Received April 2, 2018, accepted May 6, 2018, date of publication May 15, 2018, date of current version June 26, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2836342

Verification of Program by Inspecting Internal Relations Relying on User Requirements

YUZHOU LIU[1,2], LEI LIU[1,2], HUAXIAO LIU[1,2], AND HONGJI YANG[3]
[1]College of Computer Science and Technology, Jilin University, Changchun 130012, China
[2]Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun 130012, China
[3]Department of Informatics, Leicester University, Leicester LE1 7RH, U.K.

Corresponding author: Huaxiao Liu (liuhuaxiao@jlu.edu.cn)

This work was supported by the National Key Research and Development Program of China under Grant 2017YFB1003103.

ABSTRACT Software verification can ensure the software quality by inspecting the properties of program. A key issue for software verification is to check whether the software can meet user requirements especially when the requirements change frequently. To tackle this problem, we propose an approach to verify the program by inspecting the internal relations with the user requirements. In the approach, the constraints in the requirements are represented by a concern-based model defined in our previous work by Liu *et al.* and the internal relations of program are extracted based on static analysis methods; then, a framework of verification system is defined to inspect whether the program can satisfy the constraints for discovering the errors with their locations. The main contribution of this paper includes: 1) kinds of internal relations of program are defined and their calculation methods are given to transform the source codes to a formalized model, which is taken as the object to be verified and 2) formal description of verification system framework is given to support the automation of verification process. Since the verification tasks can be set freely based on the requirements in the system, the proposed approach can help developers to cope with the change of requirements better.

INDEX TERMS Software verification, user requirements, internal relations, formal description.

I. INTRODUCTION

The essential goal of software development is to establish the system that fully satisfies user requirements. As an effective way to guarantee the achievement of this goal, software verification has been widely studied and used in practice [1], [2]. In the process of software verification, the developers must identify what the program should do, then a verifier will check whether the program actually does what the programmer wants it to do by inspecting certain properties of the codes [3], [4]. In recent years, with the vigorous development of software industry, the change of user requirements becomes more frequent and it accelerates the update speed of software. However, when requirement changes, the corresponding changed information in properties of program is hard to be inspected directly in the verification process, which may lead to deviation or lag in software verification [5]–[7]. In such conditions, how to verify a software system efficiently becomes a new research problem.

The properties/constraints contained in user requirements play an important role for solving the above problem. In requirements, there exists the goals of a software system and the properties/constraints that the system should satisfy to realize these goals. These properties/constraints can reflect the user requirements directly, and they can be used in the verification process to discover related errors in program. Based on such ideas, many methods have been proposed and achieved good performance.

In the field of dynamic verification, which takes testing as the main form, there are many researches aim at generating testing cases more reasonable according to the properties/constraints contained in user requirements, such as the cases related to the requirements boundary [8], the automate generation of cases [9]. These cases can support the testing process and improve the testing efficiency. However, as the test cases cannot cover all the running situations of program, dynamic verification can discover the existence of errors but cannot ensure the nonexistence of them. Static software verification focuses on analyzing the codes themselves without running the program, it usually selects important properties/constraints (such as scenario constraints) as the specific goals and inspects the satisfaction degree of codes on them [10], [11]. These researches indicate that it is meaningful and feasible to use the information of user requirements (especially properties/constraints) in software verification.

Software requirements engineering translates the abstract user requirements into a well-defined question, in which the properties/constraints are described more accurately and clearly. This can provide the basis for introducing requirements to software verification. However, existing approaches and technologies in verification seldom consider using the results of software requirements engineering for inspecting the codes directly. Thus, we try to construct the bridge between requirement engineering and software verification: we take the internal relations of software as the verification object, and use the model established in the phase of requirement engineering as the standards in the process of verification. As internal relations of a program are important properties of software in deciding its functionalities, the satisfaction of them on the constraints given by requirements is the precondition for a software to meet its intended goals. In this paper, we propose the approach to Verify the Internal Relations based on the constraints in Requirements (VIRR) for inspecting whether the software can satisfy the requirements. The approach mainly includes three parts as follow.

Firstly, the concern-based model defined in our previous work [12] is used to formally describe the user requirements, where the constraints on relations are defined as the properties for the subsequent verification process.

Secondly, the internal relations of program are extracted from program. The kinds of internal relations are defined and the corresponding calculation methods are given based on static analysis of program. To simplify the process of analysis, the pre-process of program is implemented and the ordered sequence of statement numbers are defined for each internal relation to record the program path it depends on. When errors are discovered, the ordered sequences can locate the errors in the statements level.

Finally, a framework of verification system is described to support the automatic verification. We give the definition of components in the system, and use semantic functions to formally describe the execution process of system. In addition, the verification tasks can be set by developers flexibly, so that the program can be verified according to the changing of requirements for helping developer locate the codes need to be modified when updating the software.

The paper is organized as follows. The introduction of requirements model is presented in Section 2. Section 3 defines the internal relations of program and proposes the method to calculate them. In Section 4, the framework of verification system is formally described. Section 5 shows an experiment to validate the effectiveness of our approach. Then, related work and discussion are separately presented in Section 6 and Section 7. Finally, conclusion is given in the last section.

II. INTRODUCTION OF A CONCERN-BASED REQUIREMENTS MODEL

Currently, many requirements models are widely studied, such as goal-oriented model [13] and scenario-based model [14]. These methods describe user requirements from different angles. 'Separate of Concern' is a fundamental principle complied in software engineering, and many requirement models have been proposed based on it [15], [16]. In our previous work [12], we have proposed a method to model the requirements with Concerns and achieve good results. In this paper, we use this method to formally describe the constraints on the internal relations of software in requirements. The definitions are introduced briefly in this section and detailed information can be found in [12].

Definition 2.1 (Requirements-Concerns Model (RCM)): RCM is the formal presentation of software requirements based on Concern, and it is specified as a 2-tuple (Cset,RL), where:

- Cset is a set of concerns;
- RL is a set of relationships between concerns.

The specific definition of each part is presented as follows.

Definition 2.2 (Concern): each concern C is specified as a 6-tuple (Id, Des, Rs, type, Vset,VL), where

- Id refers to the name of C;
- Des is a text description of C;
- Rs is a set of labels of requirements that C relates to;
- type is the category of C;
- Vset is the three variable sets in C;
- VL reflects the relationships among the three variable sets.

Concerns are classified into four types: *topic* is a theme concern. There exits one and only one *topic* in each RCM to present the overall objectives; *kinds* presents the abstract functionality; *instance* represents the concrete functionality; *property* represents the functionality attributes.

There are three kinds of variable sets that can be defined in a concern: input, user_def, output. Vset represents the concrete variable sets for a concern and contains three elements at most. For example, if a concern C contains input and output, we write C.Vset = {input, output}.

VL is a six-digit code, encoding the relationships among variable sets. We use 1 and 0 to denote the non-empty and empty states, respectively. The digits of the code, from the first to the last, stand for input→user_def, input→output, user_def→output, user_def→input output→input, output→use_def, respectively.

By defining the Vset and VL, the internal structure of each concern is described and gives constraints on the program.

Definition 2.3 (Relationship Between Concerns): The relationship between two concerns is defined as a 4-tuple (C_i, C_j, *reqls*, *type*), where C_i and C_j are two concerns and there exists the relationship from C_i to C_j, representing that there may be the relationship of calling, information transmission, functional division, etc., between the two concerns. *reqls* is a set of requirements, which is the basis of the relationship, and we have $reqls{=}C_i.Rs \cap C_j.Rs$.

The relationship is categorized into five types:

- *Kinds-of*: the two concerns have the same *kinds*.
- *Instance-of*: the two concerns have the same *instance*.
- *Specific-of*: the relationship from kinds-concern to instance-concern, represents C_j is the specific

TABLE 1. Type-kind matrix.

type-Kind	Kinds_of	Instance_of	Specific_of	Send_of	Property_of
Data	true/false	true/false	true/false	true	true
Control	true/false	true/false	true/false	false	false

Where true/false represents type-Kind could be true or false.

implementation of C_i that corresponds to the abstract functionality.

- *Send-of*: the relationship from *instance-concern* to *kinds-concern*, represents that the information produced in C_i is delivered to C_j as input.
- *Property-of*: the relationship from property-concern to kinds-concern or instance-concern.

It should be noted that the relationship between topic-concern and kinds-concern is also shown as *Kinds-of.*

Moreover, the five types of relationships above and the corresponding data/control constraints can be determined by analyzing the features of the concerns [12]. The constraints are presented in Table 1.

Based on the analysis above, RCM can be established to present software requirements in a formal way. The RCM gives the constraints on the internal relations of the program to be developed. To better support the subsequent verification process, we give the following definitions to qualify the relationship information in RCM.

Definition 2.4. (Distance Between Concerns): For the two concerns C and C', distance is the qualified value of the relationship between them, donated by D(C, C′) and calculated by the formula below:

$$D\left(\mathrm{C,C'}\right)=\begin{cases}\dfrac{1}{\sum w_{r_i}} & \{r_i|r_i\in \mathrm{reqls}, (\mathrm{C, C', reqls, type})\in \\ & \quad \mathrm{RCM.RL}\};\\ \infty & else.\end{cases}$$

Where w_{r_i} is the weight of r_i in the set of requirements and given by the developers according to the practical conditions.

As the calculation of D(C,C′) is based on the relationship (C,C′, reqls, type), D(C,C′) has its direction and D(C,C′) $\neq$ D(C′, C) in most conditions. In addition, we define that the distance between a concern and itself is zero, that is $D(\mathrm{C,C}) = 0$. Although $D\left(\mathrm{C,C'}\right)$ can reflect the degree of correlation between C and C′, we cannot understand its meaning only by itself. For example, suppose that $D\left(\mathrm{C,C'}\right) = 0.03$, we do not know whether 0.03 means the relation is close or not. Thus, the distance information in RCM should be analyzed as a whole.

Definition 2.5 (Correlation Degree Between Concerns): For the two concerns C and C', correlation degree between them is the weight of relation between them in RCM, donated by CD(C,C′) and calculated by the following formula:

$$CD\left(\mathrm{C,C'}\right)=\frac{1/D(C,C')}{\sum_{i=1}^{i=n}\sum_{j=i+1}^{j=n}1/D(C_i,C_j)}.$$

For all the relationship in RCM, CD(C,C′) $\in$ [0, 1] and the sum of them is 1. The larger value of CD(C,C′) indicates the closer relation between C and C′.

In order to reflect the mapping relationship between RCM and software, we have the following definition:

Definition 2.6 (Realization-Relation $A \Rightarrow B$): Realization-relation is a kind of abstract relations, where A is a variable in program, and B is a concern or a variable set in RCM. Its semantic interpretation is that the existing of A is for the realization of B in program. Meanwhile, we have:

$$\mathrm{Rr}\left(\mathrm{A,B}\right)=\begin{cases}true & A\Rightarrow B\text{exists;}\\ false & \text{else.}\end{cases}$$

As the requirements are the basis of program development, some requirements related information labels can be added to the source codes based on requirements. Suppose that $\mathrm{Cset}=\{C_1, C_2, \ldots, C_n$ is the set of concerns in RCM, the principle of labels is: if the existence of a program module is to achieve a functionality corresponding to C_i, the variables in this module are named by $C_{i\_}x$. In this way, the relationship between the program and requirements can be established.

III. EXTRACTING THE INTERNAL RELATIONS FROM PROGRAM

The internal relations are essential properties of a program for deciding its functionalities. In RCM, the constraints in user requirements on the internal relations of program have been defined. By inspecting whether the internal relations of program could satisfy these constraints, the software is verified to check whether it can meet user requirements. However, as the internal relations cannot be simply extracted from source codes, the program cannot be used in the process of verification directly. In this section, we give the process to extract internal relations from program based on the static analysis of source codes. The process includes three parts as is shown in Figure 1.

Firstly, the source codes are preprocessed. By the instantiation of function calls and the update of statement numbers, the relation information contained in function/procedure calls is integrated into one procedure, so that the static analysis of program can be simplified. Secondly, the internal relations are defined and the corresponding calculation methods are given to extract the variables and relations from codes. Finally, the variables are classified and redundant relations are deleted utilizing the information in RCM. We establish a Program Internal Relation Model (PIRM) to describe the information of internal relations gained in this process.

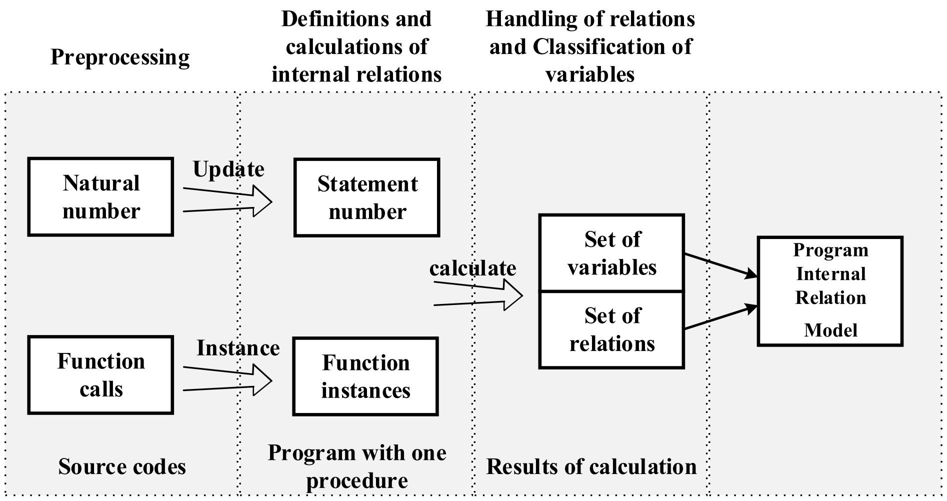


FIGURE 1. The process of acquiring the internal relations from source codes.

A. THE PREPROCESS OF SOURCE CODES

In the program, there are some kinds of statements that do not affect internal relations, such as variable declaration statements, input and output statements, etc. These statements can be ignored in the process of static analysis. Thus, we define grammars of the program to be analyzed in our approach as shown in Figure 2, where the meanings of symbols are shown in Table 2.

```
P ::= Main; D_p
Main ::= S
D_p ::= Proc id Module
Module ::= (params) S
params ::= id|id, params|ε
S ::= ass v E |  if E S S |  while E S |  Call id EL | S;  S
E ::=  n | v | id | E ⋈ E
EL ::= E | E, EL
```

FIGURE 2. Grammar of program.

TABLE 2. Type-kind matrix.

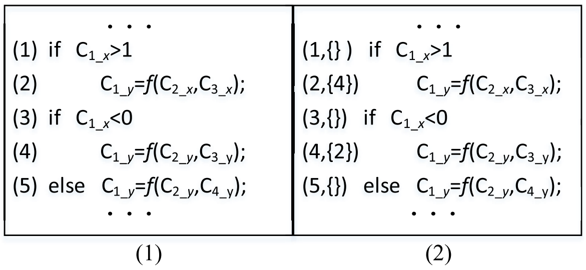
Symbol	Meaning
P	Program
Main	The main function
Params	List of parameters in function
Dp	Declaration of function
⋈	Operator between expressions

The verification object of our approach is the program conforming to the above grammar. It can be seen that there are no side effects in the program. As the statement with side effects usually could be expressed by a combination of statements (for example: z=x+y++ is the same as z=x+y and y=y+1), we do not give special analysis method for such statements. In addition, our method is for serial program, so the questions in parallel program are not considered in the grammar.

For the static analysis of program, the handlings of function/procedure calls are always a difficult question to be solved [17]. A effective way is to denote the program by a dependence graph or complex network, in which nodes and edges are modeled based on functions and their calls [18], [19]. However, the characters of our verification object provide a unique condition for us to simplify this question. In our approach, we aim at extracting the internal relations generated for meeting user requirements. In other word, we want to acquire the relations that is the embodiment of requirements in program. In this condition, the relations generated during some function/procedure calls are the same, so these function/procedure calls only need to handle once in the static analysis. To analyze the influence of function/procedure calls on internal relations, the following definition is given.

Definition 3.1 (Isomorphic Function/Procedure Calls): In a function/procedure $f(v_1, v_2, \ldots, v_n)$, Suppose that $(y_1, y_2, \ldots, y_n)$ and $(y'_1, y'_2, \ldots, y'_n)$ are two actual parameter vectors of f. $\forall i \in [1, n]$, if $\exists C_j \in Cset, st.$ $\mathrm{Rr}(y_i, C_j) \wedge \mathrm{Rr}(y'_i, C_j) = true$, the two function/procedure calls are isomorphic. If there is no formal parameter defined in f, any arbitrary calls are isomorphic. In addition, if two statements S and S′ contain isomorphic function/procedure calls, the two statements are isomorphic, and we write $(S \backsimeq S') = true$.

For example, suppose that a fragment of a program is shown in Figure 3(1), the corresponding Cset $= \{C_1, C_2, C_3, C_4$ in RCM. There are three function calls of f in the statements (2), (4) and (6): the calls in (2) and (4) are isomorphic because the parameters correspond to the same concerns



```
(1)                                (2)
        . . .                              . . .
(1) if  C1_x>1                     (1,{} ) if  C1_x>1
(2)        C1_y=f(C2_x,C3_x);      (2,{4})        C1_y=f(C2_x,C3_x);
(3) if  C1_x<0                     (3,{})  if  C1_x<0
(4)        C1_y=f(C2_y,C3_y);      (4,{2})        C1_y=f(C2_y,C3_y);
(5) else  C1_y=f(C2_y,C4_y);       (5,{})  else  C1_y=f(C2_y,C4_y);
        . . .                              . . .
```

FIGURE 3. **A fragment of a program.**

C_2 and C_3; while, the call in (6) is not isomorphic with the former two because its parameters correspond to the concerns C_2 and C_4 which are different with the ones in (2) and (4).

In program, the internal relations generated by isomorphic function/procedure calls are the same. Thus, in the preprocessing of program, we can generate function instances to eliminate function/procedure calls by analyzing the isomorphic relations among function/procedure calls. In addition, in order to record the information of function/procedure calls accurately, we re-defined the form of number to annotate a statement, so that errors discovered in the verification can be located.

Definition 3.2 (Statement Number (N)): For a statement S, its N is a 2-puple and donated by (n, n_set), where:

- n: the unique natural number annotating location of S in source codes;
- n_set: a set of locations of statements.

There are three conditions for identifying n_set: 1) if S contains a function/procedure call, n_set records all the locations of the statements that are isomorphic to S; 2) if S is a statement in the function instance, n_set records the locations of the statement that calls the function contains S and the locations of the statements that are isomorphic to this function call; 3) otherwise, $n\_set = \emptyset$.

We take Figure 3(2) as example. Due to the statements (2) and (4) in Figure 3(1) are isomorphic calls, the statement number is changed to (2,{4}) and (4,{2}). As the unique identification of a statement, statement number can help developers to locate its place easily. Thus, an ordered sequence of statement numbers can be used to express the program path and record the information of function/procedure calls.

Then, two operations are given for preprocessing the source codes. Suppose that the program conforming to the grammar shown in Figure 2, and in the program, the set of statements is S and the set of declarations of functions is D_p

Operation 3.1 (Function/Procedure Instancing Operation (FucIPo)): Function/procedure Instancing Operation transfers a function/procedure call and the body of corresponding function/procedure to a new sequence of program statements, it can be described by the following semantic function:

$$\text{FucIOp}: S \times D_p \longrightarrow S.$$

FucIOp uses the actual parameters in S instead of the formal parameters in the body of function/procedure declared by D_p. Furthermore, if S is an assignment statement, the relation between the variables is assigned and the function/procedure returns the values. In this way, an instance is created and expressed as a sequence of program statements.

Operation 3.2 (Statement Number Updating Operation (StUpOp)): StUpOp can change an original natural number n to statement number N or update an existing N. It is described as follow:

$$\text{StUpOp}: S \rightarrow N \times S.$$

Based on the definitions and operations above, the program can be preprocessed. For a given program, we scan its statements in turn. Suppose that the current statement is S:

1) If S do not contain any function/procedure call, just change its number to a statement number as we defined;
2) Otherwise, there are two conditions: if there exists a function instance created for the call that is isomorphic to the call in S, the statement numbers in function instance are updated by adding the location of S to each n_set; otherwise, a function instance is created for the call in S and a statement number is added to each statement in the function instance.

To describe the above preprocess clearly, we give its formal description as below:

```
PrePro: $P^{(1)} \rightarrow P^{(2)}$;
$\forall S \in Main^{(1)}$:
1) if $\mathcal{M}^f(S) = false$,
            StUpOp($S$);
2) if $\mathcal{M}^f(S) = true$,
   if $\exists S' \in P^{(2)}, st. (S \doteq S') \wedge \mathcal{M}^{funIO}(S', P^{(2)}) = true$,
            StUpOp($S'$);
   else $S^* = \text{FuncIOp}(S, D_p)$,
            StUpOp($S^*$).
where $\mathcal{M}^f(S)$ and $\mathcal{M}^{funIO}$ are defined as:
$\mathcal{M}^f(S) = \begin{cases} true, & S \text{ contains function/procedure call;} \\ false, & otherwise. \end{cases}$
$\mathcal{M}^{funIO}(S', P^{(2)}) = \begin{cases} true, & P \text{ contains the function instance created for the call in } S'; \\ false, & otherwise. \end{cases}$
```

In the preprocess of program, function instances and re-defined statement numbers are used together to integrate the internal relation information in function/procedure calls into a main function. So that the subsequent static analysis of program can be simplified and done in one procedure.

B. THE DEFINITIONS AND CALCULATION METHOD OF INTERNAL RELATIONS

After preprocessing, the program is analyzed to extract variables and the relations among them. To simplify the description, there are some description rules in this subsection as follow:

- V is the set of all the variables in program;
- E is the set of all the expressions in program;
- Suppose that $e \in E$, $V(e)$ is the set of all the variables that appear in e.

Based on the rules above, the definitions of internal relations are given.

Definition 3.3 (Relation Ds): If S is an assignment statement for the variable v, we say that S defined v. And a two-tuple (v, N) is used as an element to record variable defined v, where N is the statement number of S. Ds is a set of all elements defined in S.

Definition 3.4 (Relation λ_s): Suppose that $v \in V$, $e{\in}E$, if the value of v takes part in the calculation of e directly or indirectly, v and e have the relation λs, written as $(v, e, N^*) \in \lambda\text{s}$. Where N^* is an ordered sequence of statement numbers to record the path which is depended by the forming of relation between v and e.

Definition 3.5 (Relation μ_s): Suppose that $v \in V$, $e{\in}E$, if the e takes part in calculating the value of v directly or indirectly, v and e have the relation μs, written as $(e, v, N^*) \in \mu\text{s}$. Where N^* is an ordered sequence of statement numbers to record the path which is depended by the forming of relation between e and v.

Definition 3.6 (Relation ρ_s): Suppose that $v \in V, v'{\in}V$, if the value of v takes part in calculating the value of v'directly or indirectly, v and v' have the relation ρs, written as $(v, v', N^*) \in \rho\text{s}$. Where N^* is an ordered sequence of statement numbers to record the path which is depended by the forming of relation between v and v'.

Definition 3.7 (Relation η_s): Suppose that $v \in V$, $e{\in}E$, if the e is judgment condition for branching or looping construct and there exists S that defines v in the body of branching or looping construct, e controls the execution of S directly or indirectly, v and e have the relation ηs, written as $(v, e, N^*) \in \eta\text{s}$. Where N^* is an ordered sequence of statement numbers to record the path which is depended by the forming of relation between v and e.

Definition 3.8. (Relation σ_s): Suppose that $v \in V, v'{\in}V$, if the value of v controls the execution of Swhich defines v'directly or indirectly, v and v' have the relation σs, written as $(v, v', N^*) \in \mu\text{s}$. Where N^* is an ordered sequence of statement numbers to record the path which is depended by the forming of relation between v and v'.

The operation between any two kinds of relations is defined. Suppose that Rset = {λs, μs, ρs, ηs, σs, for any two relations R, R′, we have:

$$(x, y, N^* \in R) \wedge ((\text{y}, z, N^{'*}) \in \text{R}') \Rightarrow$$
$$((x, z, N^* :: N^{'*}) \in \text{RR}') \wedge (\text{RR}' \in \text{Rset}),$$

Where :: is a connector which means that N^* and $N^{'*}$ are connected orderly.

From the above basic definitions, it can be seen that the relation among ρs, λs and μs is $\rho\text{s} = \lambda\text{s}\mu\text{s}$, while, the relation among σs, λs and ηs is $\sigma\text{s} = \lambda\text{s}\eta\text{s}$.

The calculation methods of Ds and the other relation types are given separately according to the three basic program structures and the assignment statement that delivers the data.

1) Assignment statement S: ass $v\,e$, (N is the statement number of S):

$$\text{Ds} = \{(v, N)\},$$
$$\lambda\text{s} = \left\{(v', e, N \mid v' \in \text{V}(e)\right\},$$
$$\mu\text{s} = \{(e, v, N)\}, \quad \eta\text{s} = \emptyset,$$
$$\rho\text{s} = \{(v', v, N) | v' \in \text{V}(e)\}, \quad \sigma_s = \emptyset.$$

2) Sequence structure S:S_1; S_2.

$$\text{Ds} = Ds_1 \cup Ds_2, \quad \lambda\text{s} = \lambda\text{s}_1 \cup \rho\text{s}_1\lambda\text{s}_2 \cup \lambda\text{s}_2,$$
$$\mu\text{s} = \mu\text{s}_1 \cup \mu\text{s}_1\rho\text{s}_2 \cup \mu\text{s}_2, \quad \eta\text{s} = \eta\text{s}_1 \cup \eta\text{s}_1\sigma\text{s}_2 \cup \eta\text{s}_2$$
$$\rho\text{s} = \rho\text{s}_1 \cup \rho\text{s}_2 \cup \rho\text{s}_1\rho\text{s}_2,$$
$$\sigma\text{s} = \sigma\text{s}_1 \cup \sigma\text{s}_2 \cup \sigma\text{s}_1\sigma\text{s}_2 \cup \rho\text{s}_1\sigma\text{s}_2.$$

3) Branching structure S: if e S_1S_2, (the statement number of e is N):

$$\text{Ds} = Ds_1 \cup Ds_2,$$
$$\lambda\text{s} = \lambda\text{s}_1 \cup \lambda\text{s}_2 \cup \{(v, e, N) | v \in V(e)\},$$
$$\mu\text{s} = \mu\text{s}_1 \cup \mu\text{s}_2, \quad \eta\text{s} = \eta\text{s}_1 \cup \eta\text{s}_2 \cup \{(e, v, N^*) | (v, N') \in Ds_1 \cup Ds_2, N^* = N :: (\text{e} \longrightarrow v)N^*\},$$

where:

$$(\text{e} \longrightarrow v')N^* = \begin{cases} N_1^* & \exists(e', v, N_1^*) \in \eta\text{s}_1; \\ N_2^* & \exists(e', v, N_2^*) \in \eta\text{s}_2; \\ N' & \text{otherwise.} \end{cases}$$
$$\rho\text{s} = \rho\text{s}_1 \cup \rho\text{s}_2,$$
$$\sigma\text{s} = \sigma\text{s}_1 \cup \sigma\text{s}_2 \cup \lambda\text{s}\eta\text{s}.$$

4) Looping structure S : whileeS_1, (the statement number of e is N):

$$\text{Ds} = Ds_1,$$
$$\lambda\text{s} = \{(v, e, N) | v \in V(e) \cup \rho\text{s}_1^* \cup (\{(v, e, N) | v \in V(e) \cup \lambda\text{s}_1) \cup \lambda\text{s}_1,$$

where $\rho s^* = \rho s \cup \rho s^2 \cup \ldots \cup \rho s^n$,

$$\mu\text{s} = \mu\text{s}_1 \cup \mu\text{s}_1\rho\text{s}_1^*,$$
$$\eta\text{s} = \eta\text{s}_1 \cup \{(e, v, N^*) | (v, N) \in Ds_1,$$
$$N^* = N :: (\text{e} \longrightarrow v)N^*\},$$

where:

$$(\text{e} \longrightarrow v')N^* = \begin{cases} N_1^* & \exists(e', v, N_1^*) \in \sigma\text{s}_1; \\ N & \textit{otherwise.} \end{cases}$$
$$\rho\text{s} = \rho\text{s}_1^*, \quad \sigma\text{s} = \sigma\text{s}_1 \cup \lambda\text{s}\eta\text{s}.$$

The ρs and σs contain the internal relations and their path information, which can support the verification of program. The roles of λs, μs and ηs are helping the calculation of ρs and σs.

Based on the above definitions and calculation methods, static analysis of program can be done to extract the internal relations. And the program can be abstractly expressed as a three-tuple $P = (\mathrm{V}, \rho\mathrm{s}, \sigma\mathrm{s})$.

C. HANDLING OF REDUNDANT RELATIONS AND CLASSIFICATION OF VARIABLES

In practice, there are variables in program that cannot be connected to the requirements, for example, a programmer can define memory management and memory assignment for implementing an algorithm which may not be linked directly to the requirement. These variables and the relations between them cannot provide useful information for the verification, so the program P is further analyzed to delete such redundant information. Furthermore, there are three elements in $C.Vset$ and constraints between them are given in VL of RCM (introduced in section 2). In order to support the verification of these constraints, the variables are classified and connected with the three kinds of elements in a Concern. To simplify the description, we define the mapping function between variables and concerns in RCM.

Definition 3.9. (Mapping Function $\mathcal{M}^{\mathbf{v}\to\mathbf{C}}$): A mapping function of $\mathcal{M}^{v\to C}$ returns the concern that has realization-relation with the given variable in RCM. The form of $\mathcal{M}^{v\to C}$ is shown as below:

$$\mathcal{M}^{v\to C} : v \times RCM \to C.$$

$$\mathcal{M}^{v\to C}(v, RCM) = \begin{cases} C_i & \exists C_i \in RCM \cdot Cset \wedge Rr(v, C_i); \\ Null & else. \end{cases}$$

1) HANDLING OF REDUNDANT RELATIONS

The process of deleting redundant relations includes two sub-process: deletion of relations and establishment of new mapping relationships. Next, the detailed introduction of each sub-process is given.

1) In the sets of relations ρs and σs, if a relation established by two variables that cannot be connected with requirements, the relation is useless in the verification and can be deleted. Take ρs as example, the concrete rules is shown:

$\forall(v, v', N^*) \in \rho\mathrm{s}$:

- if $v = v'$, delete (v, v', N^*).
- if $\mathcal{M}^{v\to C}(v, RCM) \neq \emptyset \wedge \mathcal{M}^{v\to C}(v', RCM) \neq \emptyset$, $(v, v', N^*) \in \rho\mathrm{s}$;
- if $\mathcal{M}^{v\to C}(v, RCM) = \emptyset \wedge \mathcal{M}^{v\to C}(v', RCM) \neq \emptyset \wedge\ (\exists\, (v'', v, N^{'*}) \in \rho\mathrm{s})$,
  delete (v, v', N^*) and add $(\mathrm{v}'', \mathrm{v}', N^{'*}{::}N^*)$ to ρs;
- if $\mathcal{M}^{v\to C}(v, RCM) \neq \emptyset \wedge \mathcal{M}^{v\to C}(v', RCM) = \emptyset \wedge$ $(\exists\, (v', v'', N^{'*}) \in \rho\mathrm{s})$,
  delete (v, v', N^*) and add $(v, \mathrm{v}'', N^*{::}N^{'*})$ to ρs;
- if $\mathcal{M}^{v\to C}(v, RCM) = \emptyset \wedge M^{v\to C}(v', RCM) = \emptyset$, delete (v, v', N^*).

According to the rules above, the redundant relations in ρs are deleted. Meanwhile, σs is handled in the same way. The new ρs and σs are acquired as the basis of the following process.

2) In the new ρs and σs, if a relation still contains a variable that cannot be connected with requirements, the relation is used to send data between two functional modules corresponding to different Concerns in RCM, and the variables in this relation are mapped to the same concern. In this case, a new mapping relationship is established to alleviate the effect of information labels missing caused by non-standard programming. The rules are given by taking ρs as example.

$\forall(v, v', N^*) \in \rho\mathrm{s}$:

- if $\mathcal{M}^{v\to C}(v, RCM) \neq \emptyset \wedge \mathcal{M}^{v\to C}(v', RCM) = \emptyset$, $\mathrm{Rr}(v', \mathcal{M}^{v\to C}(v, RCM)) = true$;
- if $\mathcal{M}^{v\to C}(v, RCM) = \emptyset \wedge \mathcal{M}^{v\to C}(v', RCM) \neq \emptyset$, $\mathrm{Rr}(v, \mathcal{M}^{v\to C}(v', RCM)) = true$.

The above two sub-processes are executed in turn, that is only after 1) deletion of relations has completed, 2) the establishment of new mapping relationships can be executed. In this way, the race conditions between the two sub-processes can be avoided. After the handling of relations, ρs and σs meet the following the property:

$$\forall(v, v', N^*) \in \rho\mathrm{s} \cup \sigma\mathrm{s} \Rightarrow$$
$$\mathcal{M}^{v\to C}(v, RCM) \neq \emptyset \wedge \mathcal{M}^{v\to C}(v', RCM) \neq \emptyset.$$

That is, there are corresponding information in RCM for all the relations in ρs and σs.

2) CLASSIFICATION OF VARIABLES

According to the mapping relationship with the concern in RCM and the role it plays in the program, a variable is classified in two levels: concern level and internal variable sets level.

1) In the concern level, the variables are classified into n kinds based on the concerns in RCM:

$\forall v \in \mathrm{V}$:

$$\text{if } \mathcal{M}^{v\to C}(v, RCM) = C_i,\ V(C_i) = V(C_i) \cup v;$$
$$\text{otherwise } \mathcal{M}^{v\to C}(v, RCM) = \emptyset,\ delete\ v.$$

Thus, the set of variables V can be expressed as $\mathrm{V} = \{\mathrm{V}(C_1), \ldots, \mathrm{V}(C_n)\}$, where the variables in $\mathrm{V}(C_i)$ are related to the concern C_i in RCM.

2) Furthermore, in the internal variable sets level, the variables in a certain $\mathrm{V}(C_i)$ are further analyzed to establish mapping relationships with the three variable sets i*nput*,*output*and *user_def* of C_i. In this process, three corresponding sets of variables $\mathrm{V}(C_i)_{input}$, $\mathrm{V}(C_i)_{output}$, $\mathrm{V}(C_i)_{use\_def}$ are created based on whether relations containing the given variables can send or receive message from a variable related to another Concern in RCM:

$\forall v \in V(C_i)$:

- if $\exists(v', v, N^*) \in \rho\mathrm{s} \cup \sigma\mathrm{s} \wedge v' \notin \mathrm{V}(C_i)$, $\mathrm{V}(C_i)_{input} = \mathrm{V}(C_i)_{input} \cup \{v\}$;
- if $\exists(v, v', N^*) \in \rho\mathrm{s} \cup \sigma\mathrm{s} \wedge v' \notin \mathrm{V}(C_i)$, $\mathrm{V}(C_i)_{output} = \mathrm{V}(C_i)_{output} \cup \{v\}$;
- if $v \notin \mathrm{V}(C_i)_{input} \wedge v \notin \mathrm{V}(C_i)_{output}$, $\mathrm{V}(C_i)_{use\_def} = \mathrm{V}(C_i)_{use\_def} \cup \{v\}$.

Based on the rules above, each $\mathrm{V}(C_i)$ is further classified. $\mathrm{V}(C_i)_{input}$ contains the variables that receive data or control information from other fragments of program that correspond to the concerns different form C_i. $\mathrm{V}(C_i)_{output}$ contains the variables that send data or control information to other fragments of program. In $\mathrm{V}(C_i)_{use\_def}$, the variables are created just to realize the functionality related to C_i. It is worth to note that the sets $\mathrm{V}(C_i)_{input}$, $\mathrm{V}(C_i)_{output}$, $\mathrm{V}(C_i)_{use\_def}$ meet the following properties:

- $\mathrm{V}(C_i)_{input} \cup \mathrm{V}(C_i)_{output} \cup \mathrm{V}(C_i)_{use\_def} = \mathrm{V}(C_i)$;
- $\mathrm{V}(C_i)_{input} \cap \mathrm{V}(C_i)_{use\_def} = \emptyset$;
- $\mathrm{V}(C_i)_{output} \cap \mathrm{V}(C_i)_{use\_def} = \emptyset$.

However, $\mathrm{V}(C_i)_{input} \cap \mathrm{V}(C_i)_{output}$ can be not null. That is because a variable can go through a program fragment for both receiving and sending information.

Based on the above three parts, the internal relation of program is acquired. To express the final information, a model is established.

Definition 3.10. (Program Internal Relation Model (PIRM)): PIRM is specified as 3-tuple (V, ρs, σs) where

- $\mathrm{V} = \{\mathrm{V}(C_1), \ldots, \mathrm{V}(C_n)$, where $\mathrm{V}(C_i) = \{\mathrm{V}(C_i)_{input}, \mathrm{V}(C_i)_{output}, \mathrm{V}(C_i)_{use\_def}$;
- ρs is the set of internal relations of data in program;
- σs is the set of internal relations of control in program.

PIRM is the presentation of internal relations, it describes a program with its variables and the relations between them. Thus, PIRM can be utilized instead of the program itself as the object to be verified for inspecting whether the program can satisfy the corresponding constraints in requirements.

IV. VERIFICATION SYSTEM FOR VIRR

Internal relations of program are expressed by PIRM and its constraints in requirements are defined in RCM. Based on the PIRM and RCM, we give a framework of verification system for VIRR in this section. In order to support automatic realization of the system, formal definitions of the components in the system are given and the implementation process of the system is defined by formal semantic functions.

A. THE VERIFICATION METHOD AND THE FRAMEWORK OF VERIFICATION SYSTEM

In RCM, the relationship can be divided into two kinds: inner relationship of concerns and relationship between concerns. Correspondingly, there are two forms of the constraints on the internal relations of program: 1) inner relationship of concerns, the constraints restrict the construction of a concern by defining the relationship among the three variable sets; 2) the constraints on relationship between concerns, and this kind of information is specified by defining the type of the relationship. Furthermore, we can identify the kind of constraints that a relation in PIRM should obey according to the variables contained in the relation, that is for a given relation $\mathfrak{r} = (v, v', N^*)$ and $\mathfrak{r} \in \rho\mathrm{s} \cup \sigma\mathrm{s}$, there are two conditions:

- If $\mathcal{M}^{v\to C}(v, RCM) = \mathcal{M}^{v\to C}\left(v', RCM\right) = C$, the $\mathfrak{r}$ should satisfy the constraints on inner relationship of C;
- if $(\mathcal{M}^{v\to C}(v, RCM) = C) \wedge (\mathcal{M}^{v\to C}(v', RCM) = C') \wedge (C \neq C')$, the $\mathfrak{r}$ should obey the constraints between C and C'.

Based on the rules above, the mappings between internal relations PIRM and constraints in RCM are established. Then, the verification of software can be done also from two aspects:

1) Whether all the constraints in RCM can be realized in the program;
2) Whether all the internal relations in PIRM satisfy their corresponding constraints.

The verification process of above two aspects are driven by the requirements. Specifically speaking, the constraints in the requirements are acquired from RCM and the set S of internal relations related to the constraints is gained from PIRM, then we have the following judgements:

- If $S = \emptyset$, it presents that the program lacks of the realization to the constraints, that is the program cannot meet the requirements;
- If $S \neq \emptyset$, further judgements are given: If there exists a relation $\mathfrak{r} \in S$ and $\mathfrak{r}$ does not meet the constraints, $\mathfrak{r}$ contains errors and the program needs to be modified; otherwise, the program meets the requirements.

To support the application of above verification idea, we give a framework of verification system in Figure 4, where solid lines link the operations in the system and indicate their execution sequence in the verification process, while the dotted lines show the input and output of these operations.

Firstly, a verification task is selected. Then, the constraints and internal relations related to the task are acquired from RCM and PIRM separately according to the task. Finally, the result is generated by inspecting whether the internal relations can meet the constraints, and a signal is generated as the feedback for the system to select another task. The above process is executed in an iterative way until all the tasks have been verified, and the result is send to the developers. The detailed description of the verification system is given in the following sub-sections.

B. THE FORMAL DEFINITIONS OF THE COMPONENTS

As is shown in Figure 4, the rectangles represent the input and output of our system. The RCM and PIRM that have been defined in the Section 2 and Section 3, and the meaning and definition of verification task, result and signal are given next.

1) VERIFICATION TASK

Verification task is the input of the system, and it is set by the developers as the objects of verification. By controlling the verification task, the developers are free to select the requirements they want to verify. According to the forms of constraints in requirements, there are two types of concrete verification tasks defined as below.

Definition 4.1 (Task of Constraints Between Concerns): A task of constraints between concerns donated by *cortask* is defined as a three-tuple $(C_i, C_j, State)$, $C_i \neq C_j$, where:

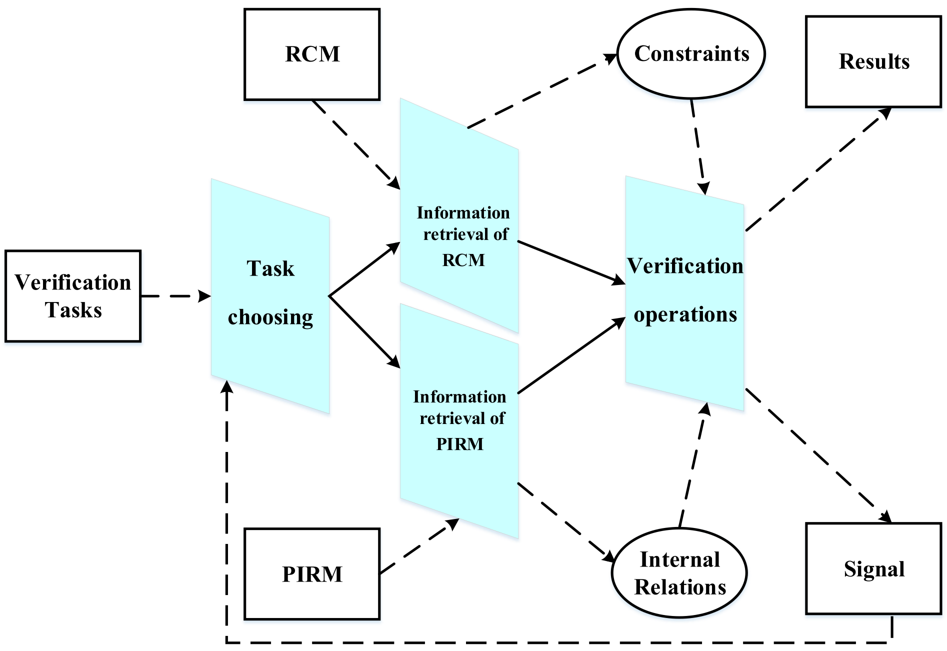


FIGURE 4. Framework of verification system.

- C_i, C_j are two different concerns, that is $C_i, C_j \in RCM.Cset$;
- *State* records whether the task has been implemented or not. If it is, $State = true$; otherwise, $State = false$.

cortask is the task that aims at inspecting whether the constraints of relationship between two concerns can be satisfied by the program.

Definition 4.2 (Task of Constraints Inner One Concern): A task of constraints inner one concern donated by *sigtask* is defined as a three-tuple $(C_i, \emptyset, State)$, where:

- C_i is a concern, that is $C_i \in RCM.Cset$;
- *State* records whether the task has been implemented or not. If it is, $State = true$; otherwise, $State = false$.

sigtask is the task that inspects whether the constraints of the inner relationship of a concern can be satisfied by the program.

Based on the definitions above, the verification task is defined.

Definition 4.3 (Verification Task): The verification task is specified as two-tuple $TASK = (\mathrm{Cor}_{\mathrm{TASK}}, \mathrm{Sig}_{\mathrm{TASK}})$, where:

- $\mathrm{Cor}_{\mathrm{TASK}}$ is the finite set of *cortask*, that is $\mathrm{Cor}_{\mathrm{TASK}} = \{cortask_1, cortask_2, ..,cortask_n\}$;
- $\mathrm{Sig}_{\mathrm{TASK}}$ is the finite set of *sigtask*, that is $\mathrm{Sig}_{\mathrm{TASK}} = \{sigtask_1, sigtask_2, \ldots, sigtask_m\}$.

There are two ways to set TASK for the developers: one is setting the TASK according to its form directly, which specifies the tasks more accurately provided by the developers who have definite verification goals and know the related information in RCM clearly; the other is giving the requirements that need to be verified and the TASK is generated automatically according to the relations between requirements and concerns in RCM. When a requirement is given, the related concerns in RCM can be identified based on the Definition 2.2. Then constraints related to these concerns are taken as the TASK.

2) RESULT

Result is the output of the system, and it records conclusion of the verification task. When errors are discovered, the result contains the type of errors and related internal relations of program to support the modification of program.

Definition 4.4 (Verification Result ($\mathfrak{Result}$)): It is defined by a two-tuple $(kind, Content)$, where

- *kind* is the category of the result;
- $Content \subset \rho s \cup \sigma s$ is the set of internal relations in the PIRM, it records the concrete information.

The $\mathfrak{Result}$ is given to developers when the verification is complete. The *kinds* of $\mathfrak{Result}$ are given in Table 3. If $\mathfrak{Result}.kind \neq Correct$, it indicates that there are errors in the program and $\mathfrak{Result}.kind$ describes the reasons of these errors; furthermore, $\mathfrak{Result}.Content$ gives the locations of errors in the statement level. For example, a $\mathfrak{Result} = (CorExtra_{Control}, (X, Y, (1, 2))\})$, it indicates that the control relation between variables X and Y makes the program cannot satisfy the constraints in the requirements, and such errors are generated from the statements 1 and 2 in the program. This information gives suggestions to the developers on modifying the errors.

It is noteworthy that there may be much information in *Content* and the location of errors is not accurate enough.

TABLE 3. Information of kind in result.

kind	Meaning
$\mathrm{CorExtra}_{\mathrm{Control}}$	The control relations in the program violate the constraints between concerns in RCM.
$\mathrm{CorExtra}_{\mathrm{Data}}$	The data relations in the program violate the constraints between concerns in RCM.
$\mathrm{CorLack}_{\mathrm{both}}$	The constraints between concerns in RCM require internal relations (control, data or both) which are lacked in program.
$\mathrm{CorLack}_{\mathrm{Data}}$	The constraints between concerns in RCM require data relations which are lacked in program.
SigExtra	The relations in the program violate the constraints generated by inner relationship of concerns in RCM.
SigLack	The constraints generated by inner relationship of concerns in RCM require internal relations which are lacked in program.
Correct	No error is discovered.
A special *kind*	
PLackV	For this *kind*, the $Content = (C, \emptyset, \emptyset)$ where $C \in$ Cset, which means there is no variables related to the concern C in program, that is for any variable v in program, $\mathrm{Rr}(v, C) = false$.

In such situation, *Content* needs to be further analyzed to give the probability of location, so that the developers can inspect the location with high probability first. The essential of *Content* is a set of internal relations described by variables and ordered sequence of statement numbers, so if the frequency of a statement appeared in *Content* is high, the probability of errors that exit in the statement is accordingly high. For example, if there are two internal relations (x_1,y_1,(*1*,*2*,*4*,*5*)) and (x_2,y_2,(*3*,*4*,*6*,*7*)) in *Content*, the statement *4* is the location that should be checked first because it appears in both relations.

3) SIGNAL AND VERIFICATION RULES

Signal is used for the information exchange in the system to drive the process of verification.

Definition 4.4 (Signal (SIGN)): A signal is specified as a two-tuple (C, C'), where

- $C \in Cset$ is a concern in RCM;
- $C' \in Cset \cup \{\emptyset \wedge C' \neq C$: If $C' \neq \emptyset$, the signal contains the information between C and $C\prime$; otherwise, the signal contains the information related to C.

In the verification system, there are a set of rules (denoted by *Rule*) in the system. These rules control the detailed process of verification and can be changed by the developers based on the actual situation. We use a set of preset functions to describe these rules.

Function 4.1 (Concern and Relationship Selected Function ($\mathcal{M}^{crs}$)):

$$\mathcal{M}^{crs}{:}(C, C') \times \mathrm{RCM} \longrightarrow (C_i, C_j),$$

where $C, C_i \in RCM.Cset, C', C_j \in RCM \cdot Cset \cup \{\emptyset\}$. The input of $\mathcal{M}^{crs}$ is RCM and (C, C') which can express a relationship either between concerns or inner-concern. The output of $\mathcal{M}^{crs}$ is (C_i, C_j), which also describes a relationship.

The $\mathcal{M}^{crs}$ is a traversal strategy of the information in RCM. In our previous work [12], [16], we have analyzed the relationship between requirements and given a method to quantify the affection degree between them. We introduce the core idea in Section 2 and give the correlation degree between concerns in definition 2.5. Here, we use it as the traversal strategy $\mathcal{M}^{crs}$ for scheduling our verification process. Specifically, When the system receive a signal (C, C'), the strategy decides which relationship would be inspected next. In our experiment, correlation degree between concerns *CD*(introduced in definition 2.5) is used to find the relationship that have the biggest correlation with the current one, that is: If $(C, C') = (C, \emptyset)$ and there is a C'' giving the biggest value of $CD(\mathrm{C},\mathrm{C}'')$, we have next task as $(C_i, C_j) = (C, C'')$; otherwise, if $C' \neq \emptyset$, $(C_i, C_j) = (C', \emptyset)$. Based on $\mathcal{M}^{crs}$, when an error is discovered and a corresponding SIGN is generated with the result, the verification system can use the SIGN to find other requirement most related to the error and set it as the new verification task. In this way, the system can discover new errors more efficiently.

Function 4.2. (Element Selected Function ($\mathcal{S}$)): $\mathcal{S}$ is a strategy for the set to select one element.

$$\mathcal{S} : Set \rightarrow element.$$

For the different sets, they can have different strategies and we use $\mathcal{S}^{Set}$ to present the concrete strategy for the given *Set*. In fact, $\mathcal{S}$ usually is a random selection strategy when the developers do not give any restrictions.

Function 4.3 (Element Selected Function ($\mathcal{F}$)): $\mathcal{F}$ is a strategy to judge whether an element is in the given set.

$$\mathcal{F} : Set \times element \rightarrow element,$$

$$\boxed{\begin{array}{l} \mathcal{F}\,[\![(\emptyset, element)]\!] \;=\emptyset; \\ \mathcal{F}\,[\![(\mathrm{Set}, element)]\!] \;=\; \text{let } ele = \mathcal{S}^{Set}(Set) \text{ in} \\ \qquad\qquad ele = element \rightarrow \\ (element), (\mathcal{F}\,[\![(\mathrm{Set} - \{ele\}, element)]\!]\;). \end{array}}$$

where *Set* is a given set and *element*.

C. IMPLEMENTATION PROCESS OF THE SYSTEM

Based on the definitions above, the implementation process of verification system is described in a formal way using semantic functions. The process inspects the internal relations of program to check whether they can satisfy the constraints in requirements according to the verification task, its semantic function is given.

Function 4.4 (Verification System ($\mathcal{V}\mathrm{erify}_{Sys}$)):

$\mathcal{V}\mathrm{erify}_{\mathrm{Sys}}$:

$$\mathrm{RCM} \times \mathrm{PIRM} \times \mathrm{TASK} \times \mathrm{SIGN} \xrightarrow{Rule} \mathcal{R}esult \times \mathrm{SIGN}.$$

$$\boxed{\begin{array}{l}
\mathcal{V}\mathrm{erify}_{\mathrm{Sys}} \llbracket (\mathrm{RCM}, \mathrm{PIRM}, \mathrm{TASK}, \mathrm{SIGN}) \rrbracket_{Rule} = \\
\text{let } (C, C') = \mathcal{T}askCOp \llbracket (\mathrm{RCM}, \mathrm{TASK}, \mathrm{SIGN}) \rrbracket_{Rule} \text{in} \\
\text{let } (\mathcal{C}o\mathcal{T}, \mathcal{S}i\mathcal{T}) = \mathrm{TASK} \text{ in} \\
\;C' = \emptyset \rightarrow \\
\;C = \emptyset \rightarrow (\emptyset), \\
\left(\begin{array}{c}
\text{let } (r, s) = \\
\left(\begin{array}{c}\mathrm{VerifyOP}_{\mathrm{Sig}} \llbracket \mathrm{RCM}, \mathrm{PIRM}, (C, \emptyset, false) \rrbracket_{Rule}, \\ (C, \emptyset)\end{array}\right) \text{ in} \\
(r, \mathrm{SIGN}) \cup \\
\mathcal{V}\mathrm{erify}_{\mathrm{Sys}} \llbracket \left(\begin{array}{c}\mathrm{RCM}, \mathrm{PAM}, \\ (\mathcal{C}o\mathcal{T}, \mathcal{M}^{state}(\mathcal{S}i\mathcal{T}, (C, \emptyset, false))), s\end{array}\right) \rrbracket_{\mathcal{R}ule}
\end{array}\right), \\
\left(\begin{array}{c}
\text{let } (r, s) = \\
\left(\begin{array}{c}\mathrm{VerifyOP}_{\mathrm{Cor}} \llbracket \mathrm{RCM}, \mathrm{PIRM}, (C, C', false) \rrbracket_{Rule}, \\ (C, C')\end{array}\right) \text{ in} \\
(r, \mathrm{SIGN}) \cup \\
\mathcal{V}\mathrm{erify}_{\mathrm{Sys}} \llbracket \left(\begin{array}{c}\mathrm{RCM}, \mathrm{PIRM}, \\ (\mathcal{M}^{state}(\mathcal{C}o\mathcal{T}, (C, C')), \mathcal{S}i\mathcal{T}), s\end{array}\right) \rrbracket_{Rule}
\end{array}\right).
\end{array}}$$

When a task is selected (expressed by function $\mathcal{T}askCOp$), it needs to judge the kind of the task for choosing a proper method to implement the verification task. Function $\mathrm{VerifyOP}_{\mathrm{Sig}}$ expresses the process of verifying the task belonging to $\mathrm{Sig}_{\mathrm{TASK}}$, while function $\mathrm{VerifyOP}_{\mathrm{Cor}}$ expresses the process of verifying the task in $\mathrm{Cor}_{\mathrm{TASK}}$. Then, the result and signal are generated where the state of finished task is changed (described by function $\mathcal{M}^{state}$). If the selected task is null, that is all the verification tasks are verified, the process of verification is ended. The concrete descriptions of functions in this process are shown next.

Function 4.5. (State Changed of Task ($\mathcal{M}^{state}$)):

$$\mathcal{M}^{state} : \mathrm{X}_{\mathrm{TASK}} \times (\mathrm{C}, \mathrm{C}', false) \rightarrow \mathrm{X}_{\mathrm{TASK}},$$

where $\mathrm{X}_{\mathrm{TASK}} \in \mathrm{Sig}_{\mathrm{TASK}} \cup \mathrm{Cor}_{\mathrm{TASK}}$.

$$\boxed{\begin{array}{l}
\mathcal{M}^{state} \llbracket (\mathrm{X}_{\mathrm{TASK}}, (C, C', false)) \rrbracket = \\
\text{let } x_{task} = \\
\quad \mathcal{F} \llbracket (\mathrm{X}_{\mathrm{TASK}}, (C, C', false)) \rrbracket \text{ in} \\
\; x_{task} = \emptyset \rightarrow \\
\left(\begin{array}{c}(\mathrm{X}_{\mathrm{TASK}}), \\ (\mathrm{X}_{\mathrm{TASK}} \cup \{(\mathcal{C}, \mathcal{C}', true)\} - (\mathcal{C}, \mathcal{C}', State))\end{array}\right).
\end{array}}$$

Function $\mathcal{M}^{state}$ describes that the process of changing the state of task when it has been verified.

Function 4.6. (Task Choosing Operation ($\mathcal{T}askCOp$)):

$$\mathcal{T}askCOp : \mathrm{RCM} \times \mathrm{TASK} \times \mathrm{SIGN} \xrightarrow{Rule} \mathrm{SIGN}.$$

$$\boxed{\begin{array}{l}
\mathcal{T}askCOp \llbracket (\emptyset, \mathrm{RL}), \mathrm{TASK}, \mathrm{sign} \rrbracket_{Rule} = \emptyset; \\
\mathcal{T}askCOp \llbracket \mathrm{RCM}, \mathrm{TASK}, \emptyset \rrbracket_{Rule} = \\
\quad \text{let } (Cset, RL) = \mathrm{RCM} \text{ in} \\
\quad \text{let } C = \mathcal{S}^{Cset}(Cset) \text{ in} \\
\; C = \emptyset \rightarrow (\emptyset), \\
\quad (\text{let } t = \mathcal{M}^{\mathcal{J}udge\_sig} \llbracket (\mathrm{C}, \mathrm{TASK}) \rrbracket \text{ in} \\
\quad (t = \emptyset \rightarrow \left(\begin{array}{c}\mathcal{T}askCOp \llbracket (Cset - \{C\}, RL), \\ \mathrm{TASK}, \emptyset \rrbracket_{Rule}\end{array}\right), \\
\qquad\quad (\text{let } (\mathrm{C}, \mathrm{C}', \mathrm{State}) = t \text{ in } (\mathrm{C}, \mathrm{C}')))); \\
\mathcal{T}askCOp \llbracket \mathrm{RCM}, \mathrm{TASK}, \mathrm{sign} \rrbracket_{Rule} = \\
\text{let } (C, C') = \mathrm{sign} \text{ in} \\
\text{let } (\mathcal{C}, \mathcal{C}') = \mathcal{M}^{crs}\big((C, C'), \mathrm{RCM}\big) \text{ in} \\
(\mathcal{C} = \emptyset \rightarrow (\mathcal{T}askCOp \llbracket \mathrm{RCM}, \mathrm{TASK}, \emptyset \rrbracket_{Rule}), \\
\quad (\text{let } t = \mathcal{M}^{\mathcal{J}udge\_co} \llbracket ((\mathcal{C}, \mathcal{C}'), \mathrm{TASK}) \rrbracket_{Rule} \text{ in} \\
\quad\; (t = \emptyset \\
\rightarrow (\mathcal{T}askCOp \llbracket \mathrm{RCM}, \mathrm{TASK}, (\mathcal{C}, \mathcal{C}') \rrbracket_{Rule}), \\
\qquad\qquad (\text{let } \big((\mathcal{C}, \mathcal{C}'), \mathrm{State}\big) = t \text{ in } (\mathcal{C}, \mathcal{C}')))).
\end{array}}$$

$\mathcal{T}askCOp$ is the realization of automatically choosing the next task from *TASK* for the system. When a task is finished, the RCM is traversed according to the strategy $\mathcal{M}^{crs}$ and a new constraint is chosen. Then, the system judges whether there are any unfinished tasks related to the constraint (described by $\mathcal{M}^{\mathcal{J}udge}$): if there exists, the task is chosen and returned by a signal; otherwise, another concern is selected according to the preset rule $\mathcal{S}^{Cset}$ to find related task. When all the concerns in RCM are traversed and no related task is unfinished, the function $\mathcal{T}askCOp$ returns null.

As there are two kinds of tasks, the function $\mathcal{M}^{Judge}$ has two forms $\mathcal{M}^{\mathcal{J}udge\_sig}$ and $\mathcal{M}^{\mathcal{J}udge\_co}$ respectively.

Function 4.7(1). (Task Judge ($\mathcal{M}^{\mathcal{J}udge\_sig}$)):

$$\mathcal{M}^{\mathcal{J}udge\_sig} : C \times \mathrm{TASK} \xrightarrow{Rule} \mathrm{TASK}.$$

$$\boxed{\begin{array}{l}
\mathcal{M}^{\mathcal{J}udge\_sig} \llbracket (\mathrm{C}, \mathrm{TASK}) \rrbracket_{Rule} = \\
\text{let } (\mathcal{C}o\mathcal{T}, \mathcal{S}i\mathcal{T}) = \mathrm{TASK} \text{ in} \\
\text{let } t = \mathcal{F}\big(\mathcal{S}i\mathcal{T}, (C, \emptyset, false)\big) \text{ in} \\
\;(t \neq \emptyset \rightarrow (C, \emptyset, false), \\
\;\;(\text{let } t' = \mathcal{S}^{Cor_{TASK}}(\mathcal{C}o\mathcal{T}) \text{ in} \\
\qquad\qquad\qquad (t' = \emptyset \rightarrow (\emptyset), \\
\left(\left(\begin{array}{c}\text{let } (\mathcal{C}, \mathcal{C}', State) = t' \text{ in} \\ ((C = \mathcal{C} \vee \mathrm{C}' = \mathcal{C}') \wedge State = false) \rightarrow \\ (\mathcal{C}, \mathcal{C}'), (\mathcal{M}^{\mathcal{J}udge\_sig} \llbracket (\mathrm{C}, (\mathcal{C}o\mathcal{T} - \{t'\}, \emptyset)) \rrbracket_{Rule})\end{array}\right)\right))).
\end{array}}$$

When the input of $\mathcal{M}^{\mathcal{J}udge\_sig}$ is a concern and TASK, the function judges if there is an unfinished task related to the concern in TASK. The related task means that the task is about the constraints generated from either the relationship in the concern C or the relationship between concern C and another concern. The $\mathcal{M}^{\mathcal{J}udge\_sig}$ returns the related task and the task belonging to $\mathrm{Sig}_{\mathrm{TASK}}$ has precedence.

Function 4.7(2) (Task Judge ($\mathfrak{M}^{\mathfrak{J}udge\_co}$)):

$$\mathfrak{M}^{\mathfrak{J}udge\_co} : (C \times C) \times \text{TASK} \xrightarrow{Rule} \text{TASK}.$$

$$\boxed{\begin{array}{l}
\mathcal{M}^{\mathfrak{J}udge\_co} \llbracket (C, C', \text{TASK}) \rrbracket_{Rule} = \\
\text{let } (\mathcal{C}o\mathcal{T}, \mathcal{S}i\mathcal{T}) = \text{TASK in} \\
\text{let } t = \mathcal{F}(\mathcal{S}i\mathcal{T}, (C, C', false)) \text{ in} \\
\qquad\qquad (t \neq \emptyset \rightarrow (C, C' \ ), (\emptyset)).
\end{array}}$$

The input of $\mathfrak{M}^{\mathfrak{J}udge\_co}$ is a relationship between concerns. In this condition, $\mathfrak{M}^{\mathfrak{J}udge\_co}$ is a simple searching function that is to judge if there is unfinished task of constraint between the concerns in Cor_{TASK}.

To verify the task, its related information must be acquired from RCM and PIRM. The processes are described by functions $\mathfrak{InforRetr}^{RCM}$ and $\mathfrak{InforRetr}^{PIRM}$ separately. Similar to function $\mathfrak{M}^{Judge}$, $\mathfrak{InforRetr}^{RCM}$ and $\mathfrak{InforRetr}^{PIRM}$ also have two forms for different kinds tasks.

Function 4.8(1) (Information Retrieval of RCM ($\mathfrak{InforRetr}^{RCM\_sig}$)):

$$\mathfrak{InforRetr}^{RCM\_sig} : \text{RCM} \times (C, \emptyset) \xrightarrow{Rule} (C, \emptyset, VL).$$

$$\boxed{\begin{array}{l}
\mathfrak{InforRetr}^{RCM\_sig} \llbracket (RCM, (C, \emptyset)) \rrbracket_{Rule} = \\
\text{let } (Cset, RL) = \text{RCM in} \\
\text{let } C' = \mathcal{F}(Cset, C) \text{ in} \\
\left(\begin{array}{c} C' = \emptyset \rightarrow \\ \emptyset, (\text{let } (id, Des, Rs, type, Vset, VL) = C' \text{ in } (C, \emptyset, VL)) \end{array} \right).
\end{array}}$$

$\mathfrak{InforRetr}^{RCM\_sig}$ acquires the inner constraints of one concern from RCM for the task belonging to Sig_{TASK}, the constraints are contained in the $C.VL$ expressed as a six-digit code.

Function 4.8(2) (Information Retrieval of RCM ($\mathfrak{InforRetr}^{RCM\_co}$)):

$$\mathfrak{InforRetr}^{RCM_{co}} : \text{RCM} \times (C, C') \xrightarrow{Rule} (C, C', type), \quad \text{where } C' \neq \emptyset.$$

$$\boxed{\begin{array}{l}
\mathfrak{InforRetr}^{RCM\_co} \llbracket ((Cset, \emptyset), (C, C')) \rrbracket_{Rule} = \emptyset; \\
\mathfrak{InforRetr}^{RCM\_co} \llbracket (RCM, (C, C')) \rrbracket_{Rule} = \\
\text{let } (Cset, RL) = \text{RCM in} \\
\text{let } (\mathcal{C}, \mathcal{C}', reqls, type) = \mathcal{S}^{RL}(RL) \text{ in} \\
C = \mathcal{C} \wedge C' = \mathcal{C}' \rightarrow \\
\left(\left(\begin{array}{c} (\mathcal{C}, \mathcal{C}', type), \\ \mathfrak{InforRetr}^{RCM_{co}} \\ \llbracket ((Cset, RL - \{(\mathcal{C}, \mathcal{C}', reqls, type)\}), (C, C')) \rrbracket_{Rule} \end{array} \right) \right).
\end{array}}$$

$\mathfrak{InforRetr}^{RCM\_co}$ acquires the constraints between two concerns from RCM for the task belonging to Cor_{TASK}, the constraints are contained in the CL expressed as the type of relationship between concerns.

Function 4.9(1) (Information Retrieval of PIRM ($\mathfrak{InforRetr}^{PIRM_{sig}}$)):

$$\begin{aligned}\mathfrak{InforRetr}^{PIRM\_sig} : \text{PIRM} \times (C, \emptyset) \xrightarrow{Rule} & \ V \times V \times V \\ & \times (Content \times Content \times Content \times Content \\ & \times Content \times Content).\end{aligned}$$

$$\boxed{\begin{array}{l}
\mathfrak{InforRetr}^{PIRM\_sig} \llbracket (\text{PIRM}, (C, \emptyset)) \rrbracket_{Rule} = \\
\text{let } (\text{Vset}, \rho s, \sigma s) = \text{PIRM in} \\
\text{let } (V_{input}, V_{output}, V_{user}) = \mathcal{F}(\text{Vset}, V(C)) \text{ in} \\
\text{let } Content^{ItU}_{control} \\
= \mathcal{M}^{Control} \llbracket (V_{iutput}, V_{user}, \sigma s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{ItU}_{data} \\
= \mathcal{M}^{Data} \llbracket (V_{iutput}, V_{user}, \rho s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{ItO}_{control} \\
= \mathcal{M}^{Control} \llbracket (V_{iutput}, V_{output}, \sigma s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{ItO}_{data} \\
= \mathcal{M}^{Data} \llbracket (V_{iutput}, V_{output}, \rho s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{UtO}_{control} \\
= \mathcal{M}^{Control} \llbracket (V_{user}, V_{output}, \sigma s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{UtO}_{data} \\
= \mathcal{M}^{Data} \llbracket (V_{user}, V_{output}, \rho s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{UtI}_{control} \\
= \mathcal{M}^{Control} \llbracket (V_{user}, V_{input}, \sigma s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{UtI}_{data} \\
= \mathcal{M}^{Data} \llbracket (V_{user}, V_{input}, \rho s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{OtI}_{control} \\
= \mathcal{M}^{Control} \llbracket (V_{output}, V_{input}, \sigma s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{OtI}_{data} \\
= \mathcal{M}^{Data} \llbracket (V_{output}, V_{input}, \rho s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{OtU}_{control} \\
= \mathcal{M}^{Control} \llbracket (V_{output}, V_{user}, \sigma s) \rrbracket_{Rule} \text{in} \\
\text{let } Content^{OtU}_{data} \\
= \mathcal{M}^{Data} \llbracket (V_{output}, V_{user}, \rho s) \rrbracket_{Rule} \text{in} \\
\\
((V_{input}, V_{output}, V_{user}, \\
\qquad (Content^{ItU}_{control} \cup Content^{ItU}_{data}, \\
\qquad Content^{ItO}_{control} \cup Content^{ItO}_{data}, \\
\qquad Content^{UtO}_{control} \cup Content^{UtO}_{data}, \\
\qquad Content^{UtI}_{control} \cup Content^{UtI}_{data}, \\
\qquad Content^{OtI}_{control} \cup Content^{OtI}_{data}, \\
\qquad Content^{OtU}_{control} \cup Content^{OtU}_{data})).
\end{array}}$$

$\mathfrak{InforRetr}^{PIRM\_sig}$ corresponds to $\mathfrak{InforRetr}^{RCM\_sig}$, its input is a concern and PIRM, and it returns the sets of variables and internal relations. In the returning, the three sets of variables correspond to three variable sets of the concern, which include *input*, *output* and *user_def*; and the six sets of internal relations correspond to each digit code of VL acquired by $\mathfrak{InforRetr}^{RCM\_sig}$ separately in order.

Function 4.9(2) (Information Retrieval of PIRM ($\mathfrak{InforRetr}^{PIRM_{co}}$)):

$$\begin{aligned}\mathfrak{InforRetr}^{PIRM\_co} : PIRM \times (C, C') \xrightarrow{Rule} & \\ V \times V \times (Content \times Content)\,, & \ C' \neq \emptyset.\end{aligned}$$

$$\begin{array}{l}
\mathcal{InforRetr}^{PIRM\_co}[\![(\text{PIRM},(C,C'))]\!]_{Rule} = \\
\text{let } (\text{Vset},\rho s,\sigma s) = \text{PIRM in} \\
\text{let } (V_{input},V_{output},V_{user}) = \mathcal{F}(\text{Vset},V(C)) \text{ in}\\
\text{let } (V'_{input},V'_{output},V'_{user}) = \mathcal{F}(\text{Vset},V(C')) \text{ in}\\
V_{output} \neq \emptyset \wedge V'_{input} \neq \emptyset \rightarrow \\
\left(\begin{array}{c} \text{let } Content_{control} = \\ \mathcal{M}^{Control}[\![(V_{output},V'_{input},\sigma s)]\!]_{Rule}\text{in} \\ \text{let } Content_{data} = \\ \mathcal{M}^{Data}[\![(V_{output},V'_{input},\rho s)]\!]_{Rule}\text{in}\\ (V_{input}\cup V_{output}\cup V_{user}),\\ (V'_{input}\cup V'_{output}\cup V'_{user}),\\ (Content_{control},Content_{data})\end{array}\right),(\emptyset).
\end{array}$$

The input of $\mathcal{InforRetr}^{PIRM\_co}$ is a relationship (C, C') and PIRM. Since the *type* in RCM restricts the type of information in the program, the different types of internal relations should be separated. Thus the returning includes: two sets of variables that are corresponding to the *output* of C and *input* of C'; two sets of internal relations containing the control relations and data relations separately. To simplify the description, functions $\mathcal{M}^{Control}$ and $\mathcal{M}^{Data}$ are used to describe the acquirement of sets of control relations and data relations.

$$\mathcal{M}^{Control} : V \times V \times \sigma s \xrightarrow{Rule} Content.$$

$$\begin{array}{l}
\mathcal{M}^{Control}[\![(V,V',\emptyset)]\!]_{Rule} = \emptyset;\\
\mathcal{M}^{Control}[\![(V,V',\sigma s)]\!]_{Rule} = \\
\text{let } (x,y,N^*) = \mathcal{S}^{\sigma s}(\sigma s) \text{ in}\\
(\mathcal{F}(V,x) \neq \emptyset \wedge \mathcal{F}(V',y) \neq \emptyset) \rightarrow \\
\left(\begin{array}{c}\{(x,y,N^*)\}\\ \cup\, \mathcal{M}^{Control}[\![(V,V',\sigma s-(x,y,N^*))]\!]_{Rule}\end{array}\right),\\
\qquad (\mathcal{M}^{Control}[\![(V,V',\sigma s-(x,y,N^*))]\!]_{Rule}).
\end{array}$$

$$\mathcal{M}^{Data} : V \times \rho s \xrightarrow{Rule} Content.$$

$$\begin{array}{l}
\mathcal{M}^{Data}[\![(V,V',\emptyset)]\!]_{Rule} = \emptyset;\\
\mathcal{M}^{Data}[\![(V,V',\rho s)]\!]_{Rule} = \\
\text{let } (x,y,N^*) = \mathcal{S}^{\rho s}(\rho s) \text{ in}\\
(\mathcal{F}(V,x) \neq \emptyset \wedge \mathcal{F}(V',y) \neq \emptyset) \rightarrow \\
\left(\begin{array}{c}\{(x,y,N^*)\}\cup\\ \mathcal{M}^{Data}[\![(V,V',\rho s-(x,y,N^*))]\!]_{Rule}\end{array}\right),\\
\qquad (\mathcal{M}^{Data}[\![(V,V',\rho s-(x,y,N^*))]\!]_{Rule}).
\end{array}$$

According to the given variables, $\mathcal{M}^{Control}$ and $\mathcal{M}^{Data}$ search the internal relations from σs and ρs in PIRM, where σs is the set of control relations and ρs is the set of data relations. If the relation contains the variables same as the given ones, the relation is taken as part of returning.

Based on the $\mathcal{InforRetr}^{RCM}$ and $\mathcal{InforRetr}^{PIRM}$, the information related to a task including constraints and its corresponding internal relations are acquired from RCM and PIRM. Then, the verification can be processed and the result is generated.

Function 4.10 (Verification Operation for Task Belong to Sig_{TASK} ($VerifyOP_{Sig}$))

$$\text{VerifyOP}_{\text{Sig}} : RCM \times PIRM \times Sig_{\text{TASK}} \xrightarrow{Rule} \text{Result}.$$

$$\begin{array}{l}
\text{VerifyOP}_{\text{Sig}}[\![(\text{RCM},\text{PIRM},\text{sigtask})]\!]_{Rule} = \\
\text{let } (C,\emptyset,\text{false}) = \text{sigtask in}\\
\text{let } (C',\emptyset,VL)\\
= \mathcal{InforRetr}^{RCM\_sig}[\![(\text{RCM},(C,\emptyset))]\!]_{Rule}\text{in}\\
\left(\begin{array}{c}
C' = \emptyset \rightarrow ((PLackV,(C,\emptyset,\emptyset))),\\
\left(\begin{array}{c}
\text{let } abcdef = VL \text{ in}\\
\text{let } \left(\begin{array}{c}V_{input},V_{output},V_{user},\\ (Con^{io},Con^{iu},Con^{uo},Con^{ui},Con^{oi},Con^{ou})\end{array}\right)\\
= \mathcal{InforRetr}^{PIRM\_co}[\![(PIRM,(C,\emptyset))]\!]_{Rule}\text{in}\\
(V_{input}\cup V_{output}\cup V_{user}) = \emptyset \rightarrow\\
((PLackV,(C,\emptyset,\emptyset)),\\
(\mathcal{M}^{ver}(a,V_{input},V_{output},Con^{io})\\
\cup\,\mathcal{M}^{ver}(b,V_{input},V_{user},Con^{iu})\\
\cup\,\mathcal{M}^{ver}(c,V_{user},V_{output},Con^{uo})\\
\cup\,\mathcal{M}^{ver}(d,V_{user},V_{input},Con^{ui})\\
\cup\,\mathcal{M}^{ver}(e,V_{output},V_{input},Con^{oi})\\
\cup\,\mathcal{M}^{ver}(f,V_{output},V_{userdef},Con^{ou})))
\end{array}\right)
\end{array}\right).
\end{array}$$

$\text{VerifyOP}_{\text{Sig}}$ describes the verification process for the task in Sig_{TASK}. For such task, its related constraints are expressed in the VL as a six-digital code. For each digital, it stands for whether there can be relations between two certain variable sets of a concern (introduced in Section 2). The verification is inspecting whether the internal relations can satisfy the constraints. Because the verification of each digital is same, we use the function $\mathcal{M}^{ver}$ to express the process, where

$$\mathcal{M}^{ver}: n \times V \times V \times \text{Content} \rightarrow \text{Result}.$$

$$\begin{array}{l}
\mathcal{M}^{ver}[\![(\text{n},\text{V},\text{V}',Content)]\!] = \\
\text{n} = 0 \rightarrow \\
\left(\begin{array}{c}
\left(\begin{array}{c}Content = \emptyset \rightarrow\\ ((Correct,(\text{V},\text{V}',\emptyset))),\\ ((SigExtra,Content))\end{array}\right),\\
\left(\begin{array}{c}Content = \emptyset \rightarrow\\ ((SigLack,(\text{V},\text{V}',\emptyset))),\\ ((Correct,content))\end{array}\right)
\end{array}\right).
\end{array}$$

Function 4.11 (Verification Operation for Task Belongs to Cor_{TASK} ($VerifyOP_{Cor}$)):

$$\text{VerifyOP}_{\text{Cor}} : RCM \times PIRM \times Cor_{\text{TASK}} \xrightarrow{Rule} Result.$$

TABLE 4. Concerns in RCM of LCS.

id	Des	Rs	type	VL
LS	Lights system	{FM1-2,FM4,FM6-7,NF3-5,U13,U14}	Topic	
FM	Facility manager	{FM4,FM6-7}	Kinds	010000
USER	User related function	{ FM2,NF3,U14}	Kinds	010000
CS	Control system	{FM1-2, NF3-5,U13-14}	Kinds	111000
FMCP	Facility manager control panel	{FM4,FM6-7}	Kinds	101000
SIG	Signal	{FM1,U14}	Instance	010000
MDMAL	mdmalfunction	{FM7,NF4}	Property	010000
OLSMAL	olsmalfunction	{FM7,NF3}	Property	010000
T2i	Manager controls timing	{FM4}	Instance	010000
MD	Motion detector	{U14}	Property	010000
OLS	Outdoor light sensor	{FM1}	Property	010000
TIME	Timing	{FM2,FM4}	Instance	111100
Override	Override control	{FM6-7}	Instance	010000
LC	Light control	{FM1-2,NF3,FM6,U14}	Instance	010000
MAL	Malfunction	{FM7,NF3-4}	Instance	010000

$$
\begin{array}{l}
\mathrm{VerifyOP_{Cor}}\llbracket(\mathrm{RCM},\mathrm{PIRM},\mathrm{cortask})\rrbracket_{Rule} = \\
\text{let } (C, C', \mathrm{false}) = \mathrm{cortask} \text{ in} \\
\text{let } (C, C', type) \\
= \mathcal{I}nfor\mathcal{R}etr^{RCM\_co}\llbracket\left(\mathrm{RCM},(C,C')\right)\rrbracket_{Rule}\text{in} \\
\text{let } \left(V, V', (Content_{control}, Content_{data})\right) = \\
\mathcal{I}nfor\mathcal{R}etr^{PIRM\_co}\llbracket\left(\mathrm{PIRM},(C,C')\right)\rrbracket_{Rule}\text{in} \\
(V = \emptyset \vee V' = \emptyset) \rightarrow
\end{array}
$$

$$
\left(\begin{array}{c}
V = \emptyset \rightarrow \\
\left((PLackV, (C, \emptyset, \emptyset) \cup \left(\begin{array}{c} V' = \emptyset \rightarrow \\ ((PLackV, (C', \emptyset, \emptyset)), (\emptyset) \end{array}\right)\right), \\
\left((PLackV, (C', \emptyset, \emptyset)\right)
\end{array}\right),
$$

$$
\left(\begin{array}{c}
type = \emptyset \rightarrow \\
\left(\begin{array}{c}
Content_{control} \cup Content_{data} = \emptyset \rightarrow \\
\left(Correct, (V, V', \emptyset)\right), \\
\left(\begin{array}{c}
Content_{control} = \emptyset \rightarrow \\
\left((CorExtra_{Data}, Content_{data})\right), \\
((CorExtra_{Control}, Content_{dcontrol}) \\
\cup\ (Content_{data} = \emptyset \rightarrow \\
(\emptyset), ((CorExtra_{Data}, Content_{data}))))
\end{array}\right)
\end{array}\right), \\
\left(\begin{array}{c}
type = Kinds_{of} \vee \\
type = Instance_{of} \vee \\
type = Specific\_of
\end{array}\right) \rightarrow \\
\left(\begin{array}{c}
Content_{control} \cup Content_{data} = \emptyset \rightarrow \\
\left(CorLack_{both}, (V, V', \emptyset)\right), \\
(Correct, Content_{control} \cup Content_{data})
\end{array}\right), \\
\left(\begin{array}{c}
Content_{control} \cup Content_{data} = \emptyset \rightarrow \\
\left(CorLack_{Data}, (V, V', \emptyset)\right), \\
\left(\begin{array}{c}
Content_{control} = \emptyset \rightarrow \left((Correct, Content_{data})\right), \\
\left(\begin{array}{c}
(CorExtra_{control}, Content_{control}) \\
\cup\ (Control_{data} = \emptyset \rightarrow \\
(CorLack_{data}, (V, V', \emptyset)), (\emptyset))
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right).
$$

$\mathrm{VerifyOP_{Cor}}$ describes the verification process for the task in $\mathrm{Cor_{TASK}}$. For such task, its related constraints are expressed by the type of relationship between concerns. For a given type of relationship in RCM, we check if there are corresponding internal relations in the program and whether the internal relations can meet the constraints.

V. EXPERIMENT

To validate our approach VIRR, we conducted experiments based on the Light Control System (LCS). As the LCS is widely studied and used in our previous work [12], we choose it as our experiment object. The requirements text of LCS is given in [20]. Our experiments aim at answering the following two questions:

Q1: Can VIRR discover the errors and locate them in the program efficiently?

Q2: Can VIRR help developers to modify the program when requirements change?

The steps and results of the experiments are analyzed in this section

A. RCM OF LCS

The requirements of LCS is described by a set of numbered statements, and we choose parts of the requirements {FM1-2, FM4,FM6-7,NF3-5,U13,U14} as the requirements set because they are related to each other closely and describe the functional needs of users. In RCM, the set of concerns *Cset*and their part relationship *RL* are shown in Table 4 and Table 5 respectively.

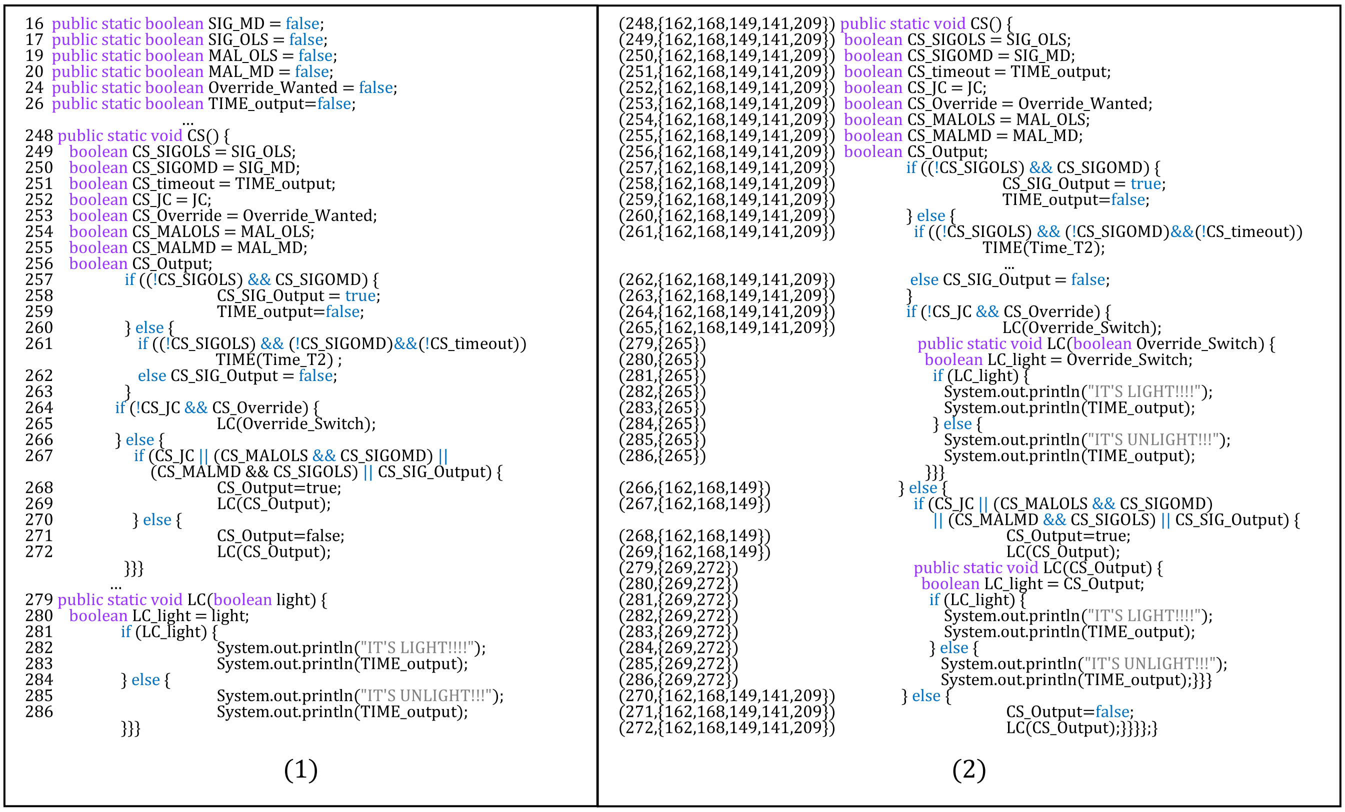
B. EXTRACTING INTERNAL RELATIONS FROM CODES AND ESTABLISHING PIRM

We invited three programmers who had more than three year's software development experience in our experiments: one of them wrote the source codes in C++ to realize the requirements above, and the other two programmers inspected the codes. As the requirements is not complicated, the program built for them was in small scale and we assumed that the three programmers could guarantee it satisfy the requirements. Then, we gained 5 different versions of the program by inserting errors and used them as the materials for answering Q1.

We established PIRM from the program for VIRR. One segmentation of program shown in Figure 5(1) is chosen

TABLE 5. Relationship between concerns in RCM of LCS.

C_i	C_j	*reqIs*	*type*	*D*	*Rp*
FM	FMCP	{FM4,FM6-7}	Kinds_of	1/3	0.048
FM	LC	{FM6}	Specific_of	1	0.016
FMCP	T2i	{FM4}	Specific_of	1	0.016
FMCP	Override	{FM6-7}	Specific_of	1/2	0.032
T2i	TIME	{FM4}	Instance_of	1	0.016
Override	LC	{FM6}	Instance_of	1	0.016
USER	LC	{FM2,NF3,U14}	Specific_of	1/3	0.048
CS	LC	{ FM1-2,NF3,U14}	Specific_of	1/4	0.065
SIG	CS	{FM1,U14}	Send_of	1/2	0.032
TIME	CS	{FM2}	Send_of	1	0.016
MAL	CS	{NF3-4}	Send_of	1/2	0.032
MD	SIG	{U14}	Property_of	1	0.016
MD	USER	{U14}	Property_of	1	0.016
OLS	SIG	{FM1}	Property_of	1	0.016
MAL	FM	{FM7}	Send_of	1	0.016
OLSMAL	MAL	{FM7,NF3}	Property_of	1/2	0.032
MDMAL	MAL	{FM7,NF4}	Property_of	1/2	0.032



```
16  public static boolean SIG_MD = false;
17  public static boolean SIG_OLS = false;
19  public static boolean MAL_OLS = false;
20  public static boolean MAL_MD = false;
24  public static boolean Override_Wanted = false;
26  public static boolean TIME_output=false;
                   ...
248 public static void CS() {
249   boolean CS_SIGOLS = SIG_OLS;
250   boolean CS_SIGOMD = SIG_MD;
251   boolean CS_timeout = TIME_output;
252   boolean CS_JC = JC;
253   boolean CS_Override = Override_Wanted;
254   boolean CS_MALOLS = MAL_OLS;
255   boolean CS_MALMD = MAL_MD;
256   boolean CS_Output;
257           if ((!CS_SIGOLS) && CS_SIGOMD) {
258                   CS_SIG_Output = true;
259                   TIME_output=false;
260           } else {
261             if ((!CS_SIGOLS) && (!CS_SIGOMD)&&(!CS_timeout))
                      TIME(Time_T2) ;
262             else CS_SIG_Output = false;
263           }
264         if (!CS_JC && CS_Override) {
265                   LC(Override_Switch);
266         } else {
267           if (CS_JC || (CS_MALOLS && CS_SIGOMD) ||
                (CS_MALMD && CS_SIGOLS) || CS_SIG_Output) {
268                   CS_Output=true;
269                   LC(CS_Output);
270           } else {
271                   CS_Output=false;
272                   LC(CS_Output);
            }}}
          ...
279 public static void LC(boolean light) {
280   boolean LC_light = light;
281         if (LC_light) {
282                   System.out.println("IT'S LIGHT!!!!");
283                   System.out.println(TIME_output);
284         } else {
285                   System.out.println("IT'S UNLIGHT!!!");
286                   System.out.println(TIME_output);
            }}}
```

(1)

```
(248,{162,168,149,141,209}) public static void CS() {
(249,{162,168,149,141,209})  boolean CS_SIGOLS = SIG_OLS;
(250,{162,168,149,141,209})  boolean CS_SIGOMD = SIG_MD;
(251,{162,168,149,141,209})  boolean CS_timeout = TIME_output;
(252,{162,168,149,141,209})  boolean CS_JC = JC;
(253,{162,168,149,141,209})  boolean CS_Override = Override_Wanted;
(254,{162,168,149,141,209})  boolean CS_MALOLS = MAL_OLS;
(255,{162,168,149,141,209})  boolean CS_MALMD = MAL_MD;
(256,{162,168,149,141,209})  boolean CS_Output;
(257,{162,168,149,141,209})          if ((!CS_SIGOLS) && CS_SIGOMD) {
(258,{162,168,149,141,209})                  CS_SIG_Output = true;
(259,{162,168,149,141,209})                  TIME_output=false;
(260,{162,168,149,141,209})          } else {
(261,{162,168,149,141,209})           if ((!CS_SIGOLS) && (!CS_SIGOMD)&&(!CS_timeout))
                                             TIME(Time_T2);
                                                 ...
(262,{162,168,149,141,209})           else CS_SIG_Output = false;
(263,{162,168,149,141,209})          }
(264,{162,168,149,141,209})          if (!CS_JC && CS_Override) {
(265,{162,168,149,141,209})                  LC(Override_Switch);
(279,{265})                            public static void LC(boolean Override_Switch) {
(280,{265})                             boolean LC_light = Override_Switch;
(281,{265})                               if (LC_light) {
(282,{265})                                 System.out.println("IT'S LIGHT!!!!");
(283,{265})                                 System.out.println(TIME_output);
(284,{265})                               } else {
(285,{265})                                 System.out.println("IT'S UNLIGHT!!!");
(286,{265})                                 System.out.println(TIME_output);
                                          }}}
(266,{162,168,149})                   } else {
(267,{162,168,149})                     if (CS_JC || (CS_MALOLS && CS_SIGOMD)
                                          || (CS_MALMD && CS_SIGOLS) || CS_SIG_Output) {
(268,{162,168,149})                               CS_Output=true;
(269,{162,168,149})                               LC(CS_Output);
(279,{269,272})                        public static void LC(CS_Output) {
(280,{269,272})                         boolean LC_light = CS_Output;
(281,{269,272})                           if (LC_light) {
(282,{269,272})                             System.out.println("IT'S LIGHT!!!!");
(283,{269,272})                             System.out.println(TIME_output);
(284,{269,272})                           } else {
(285,{269,272})                            System.out.println("IT'S UNLIGHT!!!");
(286,{269,272})                            System.out.println(TIME_output);}}}
(270,{162,168,149,141,209})           } else {
(271,{162,168,149,141,209})                       CS_Output=false;
(272,{162,168,149,141,209})                       LC(CS_Output);}}}};}
```

(2)

FIGURE 5. Preprocessing of a program segment.

as an example to illustrate this process. The major functionality of program segmentation is to receive information about light intensity and motion, etc. Based on the information, the control signal of the ceiling light is generated through logic judgment and delivered into the control segment. Since the internal structure of this program segmentation and its relationship with other programs being stable, it is representative.

Firstly, the preprocessing of program is conducted. The result of Figure 5(1) is shown in Figure 5(2). As the statements {16,17,19,20,24,26} in Figure 5(1) define the global variables and they do not reflect the internal relations of program, they are ignored in Figure 5(2). Meanwhile, since there is no formal parameter defined in function CS(), any calls of it are isomorphic. As is shown in Figure 5(2), (248,{162,168,149,141,209}) expresses that the statements

TABLE 6. Examples of Internal relations in PS.

V	V	N^*
SIG_OLS	CS_SIGOLS	((249,{162,168,149,141,209}))
SIG_MD	CS_SIGOMD	((250,{162,168,149,141,209}))
Override_Switch	LC_light	((280,{265}))
CS_Output	LC_light	((280,{269,272}))

TABLE 7. Examples of Internal relations in σs.

V	V	N^*
CS_SIGOLS	CS_SIG_Output	((257,{162,168,149,141,209}), (258,{162,168,149,141,209}))
CS_SIGOLS	TIME_output	((257,{162,168,149,141,209}), (259,{162,168,149,141,209}))
CS_JC	LC_light	((264,{162,168,149,141,209}), (280,{265}))
CS_JC	CS_Output	((264,{162,168,149,141,209}), (267,{162,168,149}),(268,{162,168,149}))

of {162,168,149,141,209} in source codes are the isomorphic. In addition, (279,{265}) and (279,{269,272}) are two function instances of LC(boolean light), that is because the parameters delivered in these two function calls are related to different concerns: *Override_Switch* $\Rightarrow$ *Override* and *CS_Output* $\Rightarrow$ *CS*.

Secondly, the sets ρs and σs of internal relations are calculated according to the methods introduced in Section 3.2.

Finally, the redundant relations are handled and variables are classified based on the rules introduced in Section 3.3. For the program segmentation shown in Figure 5(2), we have $\mathrm{V(CS)} = (\mathrm{V}(C_i)_{input}, \mathrm{V}(C_i)_{output}, \mathrm{V}(C_i)_{use\_def})$, where

$$\begin{aligned} \mathrm{V}\,(C_i)_{input} &= \left\{\begin{array}{l} \text{CS\_SIGOLS, CS\_SIGOMD,} \\ \text{CS\_timeout,} \\ \text{CS\_JC, CS\_Overrid,} \\ \text{CS\_MALOLS, CS\_MALMD} \end{array}\right\}, \\ \mathrm{V}(C_i)_{output} &= \{\text{CS\_Output}\}, \\ \mathrm{V}(C_i)_{use\_def} &= \{\text{CS\_SIG\_Output}. \end{aligned}$$

Parts of data relations and control relations are shown in Table 6 and Table 7.

C. IMPLEMENTATION OF VERIFICATION SYSTEM

In the experiment, we used the verification system to inspect the whole program. Thus, TASK of the verification included the constraints in all concerns and the constraints between concerns in RCM. For $\mathcal{M}^{crs}$, we adopted the strategy of top-down combined with biggest correlation (introduced in subsection 4.2), that is the concern in RCM was chosen according to its type in the order of *topic* $\rightarrow$ *kinds* $\rightarrow$ *instance* $\rightarrow$ *property*, and biggest correlation was used when the kind of concerns is same. For other sets, random selection strategy was used. In this way, a simply verification system was established based on our framework. Then, the system was implemented to verify the tasks.

D. THE ANALYSIS OF THE RESULTS FOR Q1

Table 8 shows an example of results of verifying one version of program by VIRR. The meaning of results is introduced in subsection 4.2. For example, in the row of $\text{CorExtra}_{\text{Control}}$, 3 means there are 3 extra control relations in the program and 14 means there may 14 paths in the program that are related to these 3 errors. Totally, our VIRR discovered 12 relation errors and located 53 related paths in this case.

TABLE 8. Examples of results.

Kind	number	The number of N^*
$\text{CorExtra}_{\text{Control}}$	3	14
$\text{CorExtra}_{\text{Data}}$	6	39
$\text{CorLack}_{\text{both}}$	1	0
$\text{CorLack}_{\text{Data}}$	1	0
SigExtra	0	0
SigLack	1	0
Total	12	53

Based on the results, three evaluation parameters are used in experiments: TP, the number of errors which are inserted by programmers and discovered by VIRR; FP, the number of errors which are discovered by VIRR but not inserted by programmers; FN, the number of errors which are inserted by programmers but not discovered by VIRR. Then we evaluate the performance of VIRR for Q1 by calculating: Precision, Recall, and F-measure, which are defined as follows:

$$\text{Precision} = \frac{TP}{TP+FP}; \text{Recall} = \frac{TP}{TP+FN};$$
$$\text{F}-\text{measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Table 9 summarizes the results of the evolution. It can be seen that the precision and recall of VIRR are 89.47% and 90.88% on average respectively. The values of F-measure for all program are above 85%, and the highest one can reach 93.75%. This indicates that our method can achieve a good performance for discovering and locating the internal relations of program that do not satisfy the requirements.

TABLE 9. Results for *Q*1.

Version	Precision	Recall	F-measure
1	85.71%	92.31%	88.89%
2	90.91%	83.33%	86.96%
3	93.75%	93.75%	93.75%
4	89.47%	85.00%	87.18%
5	87.50%	100.00%	93.33%
Average	89.47%	90.88%	90.02%

E. THE EXPERIMENT FOR ANSWERING Q2

Evaluating the usefulness of VIRR when requirements change is a nontrivial task. To answer Q2, we designed the experiment with following steps.

We firstly changed/added one requirement of LCS and used VIRR to locate the place that needs to be modified. In such condition, the verification TASK was set based on the changed requirement, so the VIRR did not need to verify the whole program again. For example, when requirement *FM6* was changed and a new requirement that "the facility manager can achieve the supreme authority to control the light" was added, the VIRR inspected related parts and gave 2 relation errors with 17 program paths that might need to be modified.

Secondly, we gave the changed requirements and information gained in the first step to the three programmers, and asked them to update the program. Meanwhile, we required the programmers to respond whether such information was useful in the updating process on a scale of 1 to 5, where 1 indicates 'totally no' and 5, 'totally yes'.

We compared the modification part of a program by programmers with the information provided by VIRR, and define the Coverage to evaluate the performance of VIRR. For each case, suppose that the statements in source codes modified by programmers for meeting the changed/new requirement established a set ChangS (sourcecodes), while the statements in the information provided by VIRR were a set VIRR*Set*, the Coverage was calculated by the formula:

$$\text{Coverage} = \frac{|\text{ChangS (sourcecodes)} \cap \text{VIRRSet}|}{|\text{ChangS (sourcecodes)}\ |}.$$

That is how many statements modified by programmers are also given by VIRR. Note that the new added statements are not considered in this process.

TABLE 10. Results for *Q*2.

Programmer	Coverage (Changes)	Coverage (New)	Score
1	79.34%	89.71%	3.80
2	81.23%	86.50%	4.00
3	82.16%	87.19%	4.40
Average	80.91%	87.80%	4.07

The experiment included 5 cases, 3 changes of requirements and 2 new requirements. The results gained from three programmers are shown in Table 10. It can be seen that the Coverage is 80.91% for the cases that changed requirements and 87.80% for the cases that added new requirements on average. Furthermore, the scores given by the programmers are all above 3 and the average is 4.07. This indicates that VIRR can help programmers modify the codes to cope with the changed/added of requirements.

After the experiment, we had an open discussion with the programmers for getting their suggestions. All of them emphasized the usefulness of VIRR in modifying the codes that related to the changes of requirements indirectly. As there are relationships between requirements, when changing a requirement or adding a new one, the statements in the codes for connecting it with other requirements need to be considered besides the part for realizing the requirement itself. VIRR can provide valuable information for locating these statements, which is beneficial for reducing the time-cost of updating the software.

F. THREATS AND LIMITATIONS OF THE EXPERIMENT

Although the results were good in our experiments, the validity of our study suffers several threats and limitations. We analyze them from two aspects.

1) THREATS TO THE VALIDITY OF EXPERIMENTS

Firstly, the scale of the dataset in our experiments is not very large. Due to our approach relies on both requirements and corresponding codes, we could not evaluate our approach with additional datasets in a short time. Thus, it is unclear whether our approach can achieve similar performance when it is applied to large-scale software. However, LCS is a typical object used in many studies, and our experiments based on it can guarantee the effectiveness of our conclusions to some extent. In addition, the discussion provides an open way for programmers to evaluate our approach.

Secondly, there lacks of comparisons between VIRR and existing methods. There are mainly three reasons. a) With respect to the execution time: the RCM is established in the phase of requirements engineering and the construction of PIRM only needs to scan the program once, meanwhile, the verification process can be taken as searching the graph of PIRM according to the information in RCM, so the time complexity of the proposed approach is $O(n^2)$. In our experiment, the time of verification process is very short as the scale of codes is small, so we could not compare the time with other methods. b) In addition, our approach requires the program containing the labels related to requirements, this means we need additional information and it would be not fair for other methods in the comparisons. c) Furthermore, there are no methods in software verification that help programmers to cope with the change of requirements to our knowledge, so we could not give the comparison from this point.

G. LIMITATIONS OF OUR APPROACH

Firstly, the kind of errors can be discovered in our approach is limited. As we focus on inspecting the internal relations of program with the constraints in requirements, only related errors can be identified. Thus, we evaluate the performance of VIRR by only inserting this kind of errors in our experiments. In the further, we will mine more kinds of information from requirements and introduce them into software verification, so that our approach can be extended to cope with different kinds of errors in program.

Secondly, our methods could not correct the errors automatically. As verification system inspect the software based on the requirements, the results can provide the information for locating the codes that may be modified, while the decisions still need to be made by programmers manually. In addition, an important goal of our approach is supporting the update of software according to the changes of requirements, and these changed requirements must be realized by developers. However, by accumulating the errors and their corresponding modification strategies in one particular domain, we may provide a list of common errors with the solutions. In this way, the system could modify some errors automatically. We wish to study this problem by cooperating with companies to apply our approach in practice.

VI. RELATED WORK

Software verification as a hot research topic in the field of computer science, it often includes two aspects, practice studies and theory researches. On one hand, some practice studies rely on verification projects, such as the certified compiler CompCert [21] and the microkernel seL4 [22], while others give verification system for specific target, Linux's USB BP Keyboard Driver [23] and event-condition-action systems [24]; meanwhile, the properties of the verification project are also studied [25], [26]. On the other hand, there are many researches on the theory of software verification, they are not limited by actual projects and have a broader research space. Due to lack of projects resource, our work pays more attention to the theoretical study and we discuss the related work from this aspect. Our work focuses on using requirements in the verification process, and tries to provide theoretical basis for bridging the gap between software verification and requirements engineering. Thus, we divide the related work from the perspective of requirements, into non-functional requirements and functional requirements.

Non-functional requirements evaluate the operations of the system and its features (such as safety, stability, etc.). Many researches are given to verify these properties of program. Alastair F. Donaldson utilizes a new k-induction rule for verifying software programs, which allows program verification using weaker loop invariants [27]. In [28], a bounded time safety verification technique for periodically actuated linear control systems is presented, by using matrix exponentiation, and symbolic evaluation of inputs, the paper gives a transformation procedure to reduce the verification problem of such systems into software verification with computation over reals. In [29], a controller is synthesized for the meshing process of Motor-Transmission Drive System with uncertain initial states and its safety property is verified with respect to the CHA model. Reference [30] presents a methodology and a tool to perform automated static analysis of embedded controller code to verify the stability of the controlled physical system, which can guarantee that the physical plant converges to a desired behavior under the actions of the controller. In [31], a new method for automatically inferring loop invariants is presented to help make extended static checking more acceptable. Reference [32] establishes the PolyPaver tool which can naturally deal with the integral operator and interval expressions and inclusions, and the paper gives a formal account of the main concepts behind it. Reference [33] proposes a novel approach for static software verification of embedded C sensor applications to verify the program and report error traces and gives a support tool. These researches focus on discovering the errors caused by coding, such as out-of-bounds arrays, null-pointer dereferences, etc. However, they do not use the information in the requirements essentially, so some high-level non-functional requirements, such as the structure of program, cannot be inspected. Different from them, our approach establishes the relationship between requirements and verification, and uses the requirement model directly in the process of verification, so it can help developers analyze software based on high-level non-functional requirements as well as functional requirements, we give a detailed discussion in Section 7.

Functional requirements define the functionality of a software system and its components. Many researches of software verification take the functionalities or behaviors of software as the objects to inspect the satisfaction degree of software on functional requirements. S.Liu proposes a method by inspecting whether every functional scenario defined in the specification is implemented correctly by a set of program paths and whether every program path of the program contributes to the implementation of some functional scenario in the specification to complete the verification of program [11]. Reference [34] reports a case on static analysis of critical C code, establishing a property on functional behavior of this code to verify the programs involving floating-point computations. In [35], the verification of parallel programs correctness is based on the axiomatic approach. In [36], a simulation framework is described to verify the real-time embedded control applications by simulating functional behavior of operating system services and hardware components at a level of abstraction. In [37], a simulation-based verification framework is firstly presented for nonlinear switched systems, in which users are required to annotate the dynamics in each control mode by a discrepancy function that formally measures the nature of trajectory convergence/divergence of the system. Reference [38] models Behaviors of the vehicle in Markov Decision Processes (MDPs) and verifies them by a probabilistic model checker PRISM. Reference [39] proposes a novel approach focusing on a process algebra structure that captures behavior-based programming to verification of performance guarantees for behavior-based robot programs. Reference [40] proposes an approach based on model checking to verify the satisfiability of behavior-aware privacy requirements in services composition. Similar to these researches, our work also extracts functionality related information from user requirements for software verification. But our approach introduces the formal model of requirements into the verification process, which makes our work different from them significantly. As the model describes the functionalities and their relationship more accurately, developers can verify part of codes based on the requirements they concerned without inspecting the whole program. Moreover, the changes of requirements are reflected by the model better, so that our approach can locate statements that need to be modified by verifying the program according to the model, this can help developers update the software to meet the ever-changing requirements efficiently.

VII. DISCUSSION

The RCM and PIRM can help establish the relations between user requirements and software, and provide the basis for inspecting whether the software can meet the requirements. Although VIRR is the verification of software from perspective of functional requirements, the information in RCM and PIRM can also help the developers to analyze and verify the program from the aspect of non-functional requirements to some extent. In this section, we propose a simply discussion on this issue.

We divide the non-functional requirements into two levels from the angle of program: one level is about the concrete requirements to the codes, such as safety; the other level is the requirements about the structure of program, such as maintainability, easy to update, etc. For the first level, many existed researches are introduced in Section 6. However, it is difficult to inspect the second level just depending on the existed methods. During our research, we discover that the relations established between RCM and PIRM are benefit for the verification and analysis of the non-functional requirements of the second level. We shows the benefits from the following three aspects.

A. PARTITION OF PROGRAM MODULES

To analyze the structure of program, the program must be partitioned into modules. Since the structured development approaches of software are mostly used recent years, there usually exist natural program modules. And many analysis methods are based on such program modules. However, the relations between natural modules and user requirements are unclear. When requirements changed, it is hard to judge the modules that need to be modified correspondingly. Thus, the analysis of structure based on the natural modules cannot directly reflect the complexity of software updating and is not adapt to verify the related non-functional requirements. In our approach, the elements in PIRM.V $= \{\mathrm{V}(C_1), \ldots, \mathrm{V}(C_n)\}$ are corresponding to the concerns in RCM, which means that they are related to the requirements. Based on the set PIRM.V, the program statements containing the variables in one element of the set are taken as a program module. The modules partitioned in this way are closer to the software requirements, so the relations between such modules can better reflect the structure of software under the constraints of requirements.

B. DESCRIPTION OF PROGRAM STRUCTURE

After the program modules have been acquired, the next question is how to describe the structure of program based on these modules. The information in ρs and σs can describe the relations inside and among these modules helping to solve this problem. Since ρs and σs contain almost all the relations in the program but not all the relations can describe the structure, we need to analyze these relations to extract useful ones. For a given relation $\mathrm{r} \in \mathrm{PIRM}.\rho\mathrm{s} \cup \mathrm{PIRM}.\sigma\mathrm{s}$, it can be a backbone relation if it can satisfy the following conditions:

- $\nexists \mathrm{r}', \mathrm{r}'' \in \mathrm{PARM} \cdot \rho\mathrm{s} \cup \mathrm{PARM} \cdot \sigma\mathrm{s}, \mathrm{st} \cdot \mathrm{r} = \mathrm{r}' + \mathrm{r}''$;
- $\nexists \mathrm{r}' \in \mathrm{PARM} \cdot \rho\mathrm{s} \cup \mathrm{PARM} \cdot \sigma\mathrm{s}, \mathrm{st} \cdot \mathrm{r}' \subset \mathrm{r}$.

The backbone relation is the embodiment of information direct transmission in the program and reflects the program intuitively. By acquiring the backbone relations in ρs and σs, we establish the new set $\rho\mathrm{s}'$ and $\sigma\mathrm{s}'$. And we define DPIRM= $(\mathrm{PIRM.V}, \rho\mathrm{s}', \sigma\mathrm{s}')$, which is the basis of describing the structure of program. In DPIRM, the relations can be divided into two type: if $\mathrm{r} = (v, v', N^*), \mathcal{M}^{v\to C}(v, RCM) = \mathcal{M}^{v\to C}(v', RCM)$, r can be used to describe the structure inside the module corresponding to concern $\mathcal{M}^{v\to C}(v, RCM)$; if $\mathrm{r} = (v, v', N^*), \mathcal{M}^{v\to C}(v, RCM) \neq \mathcal{M}^{v\to C}(v', RCM)$, r can be used to describe the structure between modules that are corresponding to $\mathcal{M}^{v\to C}(v, RCM)$ and $\mathcal{M}^{v\to C}(v', RCM)$.

C. ANALYSIS OF PROGRAM STRUCTURE

Based on the description of program structure, the properties of program can be analyzed to inspect whether they can meet non-functional requirements. To achieve this goal, the weight of a relation in the program structure should be identified. For a given relation $\mathrm{r} = (v, v', N^*)$, the number of nodes in N^* can measure the correlation degree between v and v'. And we use this number to quantify the relation r, which provides basis for the next analysis. Then, two important structure properties, coupling and cohesion, are taken as examples to discuss the approaches of analyzing program structure.

Coupling represents the overall loose degree of program structure. In general, the lower coupling is, the more easily program updates. The sum of relations among modules has the similar feature, that is, the more relations among modules, the more complex program changes. However, since the scale of program is different, only the above sum cannot be used as coupling. For instance, such sum of a large program is bigger than the one of a small program, but we cannot say that the coupling of the large program is also bigger. Thus, we think that the percent of relations among modules in $\rho\mathrm{s}'$ and $\sigma\mathrm{s}'$ can be defined as the coupling of program.

Cohesion is defined in a module to represent its independence degree. The relations related to a module can help the developers to analyze its cohesion. For a module, the relations inside it embody its inner structure and the relations between it and other modules embody its dependence or effect with other modules, so we think that the ratio of the above two kinds relations can be used to define cohesion of the module.

The coupling and cohesion defined above can be easily calculated based on DPIRM, and they can help the developers to analyze the structure of program. In this way, whether the program meets related non-functional requirements can be inspected.

Based on the discussion above, we believe that the application scope of the information in our method VIRR can be expanded. And the relations established between program and user requirements can make the verification of software more comprehensive.

VIII. CONCLUSION AND FUTURE WORK

An important goal for software verification is to inspect whether the program can meet user requirements, where the frequent change of requirements proposes new challenges. This paper proposes an approach called VIRR, that verifies the software by inspecting the internal relations of program with the user requirements. In VIRR, the requirements are presented by a concern-based model RCM, which defines the constraints that the program should satisfy; the internal relations of program are defined asλs, μs, ηs, ρs and σs, and the program is transformed to PIRM by calculating these relations; then, a verification system framework is formally described using semantic functions to support the automation of the proposed approach. Furthermore, the verification task can be set to focus on inspecting part of a program with the changing requirements. Our work shows that the verification process can be implemented to discover the errors related to internal relations at statement level, and to locate or to inspect the changed parts in the program to help developers cope better with the ever-changing requirements.

Our future work will focus on two aspects. First, as the framework of verification system has not defined the concrete structure, we will compare the efficiency of different structures to find a best one for realizing the system. Second, there are other properties/constraints defined in user requirements, and it will be studied to find whether they can be used in the software verification.

REFERENCES

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, p. 19, 2009.

[2] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking," in *Proc. Haifa Verification Conf.*, 2017, pp. 99–114.

[3] D. Matichuk, T. Murray, J. Andronick, R. Jeffery, G. Klein, and M. Staples, "Empirical study towards a leading indicator for cost of formal software verification," in *Proc. IEEE/ACM, IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 722–732.

[4] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens, "Software verification with VeriFast: Industrial case studies" *Sci. Comput. Program.*, vol. 82, no. 2, pp. 77–97, 2014.

[5] D. Beyer, "Second competition on software verification," in *Proc. Int. Conf. TOOLS Algorithms Construction Anal. Syst.*, 2013, pp. 594–609.

[6] H. Post, C. Sinz, F. Merz, T. Gorges, and T. Kropf, "Linking functional requirements and software verification," in *Proc. 17th IEEE Int. Requirements Eng. Conf. (RE)*, Aug. 2009, pp. 295–302.

[7] T. Yamaguchi, T. Kaga, and S. A. Seshia, "Combining requirement mining, software model checking and simulation-based verification for industrial automotive systems," in *Proc. IEEE Formal Methods Comput.-Aided Design*, Oct. 2017, pp. 201–204.

[8] C. Pfaller, "Requirements-based test case specification by using information from model construction," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 7–16.

[9] R. Gao, J. S. Eo, W. E. Wong, X. Gao, and S. Y. Lee, "An empirical study of requirements-based test generation on an automobile control system," in *Proc. ACM Symp. Appl. Comput.*, 2014, pp. 1094–1099.

[10] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of PLC software," *Softw. Syst. Model.*, vol. 15, no. 4, pp. 937–960, 2016.

[11] S. Liu, Y. Chen, F. Nagoya, and J. A. McDermid, "Formal specification-based inspection for verification of programs," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1100–1122, Sep. 2012.

[12] H. Liu, Y. Liu, and L. Liu, "The verification of program relationships in the context of software cybernetics," *J. Syst. Softw.*, vol. 124, pp. 212–227, Feb. 2017.

[13] M. A. Teruel, E. Navarro, V. López-Jaquero, F. Montero, and P. González, "CSRML: A goal-oriented approach to model requirements for collaborative systems," in *Proc. Int. Conf. Conceptual Modeling*, 2011, pp. 33–46.

[14] G. Babin and F. Lustman, "Application of formal methods to scenario-based requirements engineering," *Int. J. Comput. Appl.*, vol. 23, no. 12, pp. 141–151, 2015.

[15] A. Rashid and Y. Yu, "Aspect-oriented requirements engineering: An introduction," in *Proc. 16th IEEE Int. Requirements Eng. Conf. (RE)*, Sep. 2008, pp. 306–309.

[16] Y. Jin, H.-X. Liu, and P. Zhang, "An approach to analysing and verifying aspect-oriented requirements model," *Chin. J. Comput.*, vol. 36, no. 1, pp. 63–73, 2013.

[17] L. Wenhui, Z. Kuanjiu, F. Jinjin, and C. Zongzheng, "Research on software cascading failures," in *Proc. Int. Conf. Multimedia Inf. Netw. Secur. (MINES)*, 2010, pp. 310–314.

[18] D. Kang, B. Xu, J. Lu, and W. C. Chu, "A complexity measure for ontology based on UML," in *Proc. 10th IEEE Int. Workshop Future Trends Distrib. Comput. Syst. (FTDCS)*, May 2004, pp. 222–228.

[19] C. Y. Chong and S. P. Lee, "Analyzing maintainability and reliability of object-oriented software using weighted complex network," *J. Syst. Softw.*, vol. 110, pp. 28–53, Dec. 2015.

[20] S. Queins, G. Zimmermann, M. Kronenburg, C. Peper, R. Merz, and J. Schäfer, "The light control case study: Problem description," *J. Universal Comput. Sci.*, vol. 6, no. 7, pp. 586–596, 2000.

[21] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[22] G. Klein *et al.*, "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1-2:70, Feb. 2014.

[23] W. Penninckx, J. T. Müehlberg, J. Smans, B. Jacobs, and F. Piessens, "Sound formal verification of Linux's USB BP keyboard driver," in *Proc. Int. Conf. NASA Formal Methods*, 2012, pp. 210–215.

[24] M. Schordan and A. Prantl, "Combining static analysis and state transition graphs for verification of event-condition-action systems in the RERS 2012 and 2013 challenges," *Int. J. Softw. Tools Technol. Transf.*, vol. 16, no. 5, pp. 493–505, 2014.

[25] P. Philippaerts, F. Vogels, J. Smans, B. Jacobs, and F. Piessens, "The Belgian electronic identity card: A verification case study," in *Proc. 11th Int. Workshop Automated Verification Crit. Syst. (AVoCS)*, vol. 46, 2011, pp. 1–16.

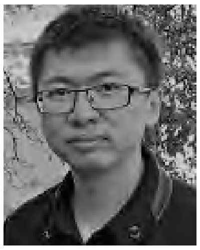
[26] F. Ivančić *et al.*, "Scalable and scope-bounded software verification in VARVEL," *Automated Softw. Eng.*, vol. 22, no. 4, pp. 517–559, 2015.

[27] A. F. Donaldson, L. Haller, and D. Kroening, "Software verification using k-induction," in *Proc. Int. Symp. Static Anal. (SAS)*, Venice, Italy, Sep. 2011, pp. 351–368.

[28] P. S. Duggirala and M. Viswanathan, "Analyzing real time linear control systems using software verification," in *Proc. Real-Time Syst. Symp.*, 2016, pp. 216–226.

[29] H. Chen and S. Mitra, "Synthesis and verification of motor-transmission shift controller for electric vehicles," in *Proc. ACM/IEEE Int. Conf. Cyber-Phys. Syst. (ICCPS)*, Apr. 2014, pp. 25–35.

[30] A. A. Martinez, R. Majumdar, I. Saha, and P. Tabuada, "Automatic verification of control system implementations," in *Proc. EMSOFT*, 2010, pp. 9–18.
[31] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 191–202, 2002.
[32] J. Duracz and M. Konečný, "Polynomial function intervals for floating-point software verification," *Ann. Math. Artif. Intell.*, vol. 70, no. 4, pp. 351–398, 2014.
[33] D. Bucur and M. Kwiatkowska, "On software verification for sensor nodes," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1693–1707, 2011.
[34] C. Marché, "Verification of the functional behavior of a floating-point program: An industrial case study," *Sci. Comput. Program.*, vol. 96, pp. 279–296, Dec. 2014.
[35] S. M. Kropacheva and I. A. Legalov, "Formal verification of programs in the functional data-flow parallel language," *Autom. Control Comput. Sci.*, vol. 47, no. 7, pp. 373–384, 2013.
[36] S. Resmerita and W. Pree, "Verification of embedded control systems by simulation and program execution control," in *Proc. Amer. Control Conf. (ACC)*, Jun. 2012, pp. 3581–3586.
[37] P. S. Duggirala, S. Mitra, and M. Viswanathan, "Verification of annotated models from executions," in *Proc. Int. Conf. Embedded Softw.*, Sep. 2013, pp. 1–10.
[38] T. Sekizawa, F. Otsuki, K. Ito, and K. Okano, "Behavior verification of autonomous robot vehicle in consideration of errors and disturbances," in *Proc. IEEE Comput. Softw. Appl. Conf.*, Jul. 2015, pp. 550–555.
[39] D. M. Lyons, R. C. Arkin, S. Jiang, T. M. Liu, and P. Nirmal, "Performance verification for behavior-based robot missions," *IEEE Trans. Robot.*, vol. 31, no. 3, pp. 619–636, Jun. 2015.
[40] J. Lu, Z. Huang, and C. Ke, "Verification of behavior-aware privacy requirements in Web services composition," *J. Softw.*, vol. 9, no. 4, pp. 944–951, 2014.



YUZHOU LIU received the bachelor's degree in optical information science and technology from the Beijing Institute of Technology, China, in 2010. He is currently pursuing the Ph.D. degree with the College of Computer Science and Technology, Jilin University, China. He was a Software Engineer with China Unicom until 2014. During the period of work, he is mainly engaged in data extraction and data analysis. His current research concerns on requirements engineering.

LEI LIU received the master's degree in computer science from Jilin University, China, in 1985. He is currently a Doctoral Supervisor with the College of Computer Science and Technology, Jilin University. At Jilin University, he has held responsibilities for over 30 projects as a lead person in the area of computer science. He has authored numerous papers and technical reports on various international journals and conferences. The central themes of his research are programming language and its realization technology, software security and cloud computing, the semantic Web and ontology engineering, and knowledge representation and reasoning.

HUAXIAO LIU received the Ph.D. degree in computer science from Jilin University, China, in 2013. He is currently an Assistant Professor with the College of Computer Science and Technology, Jilin University. The central theme of his research is improving software quality, and his recent research concerns the software requirements engineering, software cybernetics, and formal methods of software development. More specifically, he develops techniques to verify aspect-oriented requirements model based on ontology.

HONGJI YANG is currently a Professor with the Department of Informatics, Leicester University, U.K. He has published five books and over 400 research papers in the area of software engineering, distributed computing, and creative computing. He is a Golden Core Member of the IEEE CS. He served as a Program Chair for the IEEE International Conference on Software Maintenance in 1999 and the IEEE International Computer Software and Application Conference in 2002. He is editing the *International Journal of Creative Computing*.