Operator Component Matrix Model for IMP Program Diagnosis

Zhao-Fu Fan Yunfei Jiang
Department of Computer Science
School of Information Science and Technology
Sun Yat-Sen University
No. 135 Xingang Xi Rd, Guangzhou, 510275, China
fanzhaofu@163.com

Abstract

This paper presents a new modeling approach for IMP programs with operator component matrix (OCM) model, which can be used in IMP program diagnosis. Using this model and model-based diagnosis method, some logic errors can be found in IMP programs. The model can also be extended to all kinds of imperative programs. The advantages of this diagnosis method lie in its simple and regular presentation, uniform diagnosed objects, usage of isomorphism assumptions in structure, and usage of assertions about the expected program. These advantages make diagnoses more accurate, and even help to correct the faults by mutation of operator components.

1 Introduction

Model-based diagnosis was first proposed in 1987 by Reiter [Reiter, 1987], this approach is mainly used to diagnose the systems composed of some physical components. Applying model-based diagnosis to program debugging is still a new field. Now, dependency model [Mateis *et al*., 1999] [Wieland, 2001] and value-base model [Mateis *et al*., 2000] are widely used in program diagnoses, both of which use program slicing technology [Wotawa, 2002]: the former considers static slices of a program by analysing the dependency between variables, and the latter in fact uses dynamic slices depending on a special test by computing the valuation trajectory. But these methods are not competent in structural errors, and always result in very coarse diagnoses, i.e., too many candidate diagnoses.

This paper uses a simple imperative programming language—IMP which helps to focus on the important issues and avoid considering unnecessary detail. The syntax of IMP is shown in [Winskel, 1993]. For IMP programs, we present a new modeling approach with OCM, which can be used in IMP program diagnosis. Using this model and model-based diagnosis method, some logic errors can be found in IMP programs. The model can also be extended to all kinds of imperative programs. The advantages of this diagnosis method lie in its simple and regular presentation, uniform diagnosed objects, the usage of isomorphism assumptions in structure, and the usage of assertions about the expected program. These advantages make diagnoses more accurate, and even help to correct the faults by mutation of operator components.

In this paper, section 2 presents the operator component matrix model for programs. Section 3 further analyses the dependencies implied in the model, including black-box dependency and white-box dependency. Section 4 is the core of the paper, it defines such terms as isomorphism, abstract program, program specification and breakpoint, and under these terms, it also defines the model-based program diagnosis problem, diagnosis and conflict etc., and gives an example to illustrate them. Section 5 introduces related works, and the last section makes a conclusion.

2 Operator Component Matrix Model

2.1 Operator Component and 1-variable

Definition 1. In an *n*-ary function $F^n(x_0,x_1,\ldots x_{n-1})$, we use a vector $(F_0^n, F_1^n, ..., F_{n-1}^n)$ to denote F^n, then F_k^n (k=0,...n-1) is called the *k*-th **operator component** of n-ary function F^n.

So, we can represent $F^n(x_0,x_1,\ldots,x_{n-1})$ as

$$F_0^n \bullet x_0 \oplus F_1^n \bullet x_1 \oplus \ldots \oplus F_{n-1}^n \bullet x_{n-1} \qquad (1)$$

Where "$\oplus$" is called as **operator-plus** , "$\bullet$" is called as **operator-times,** and we stipulate that the symbol "$\oplus$" has associativity and exchangeability, and "$\bullet$" has associativity and distributivity to "$\oplus$". So, we can also represent (1) as $[F_0^n, F_1^n, ...F_{n-1}^n][x_0, x_1, ...x_n]^T$.

We can prove that the above stipulation is reasonable, because under the stipulation, any legal expression in the form of operator components has a unique computable form, i.e. $F_0 \bullet x_0 \oplus F_1 \bullet x_1 \oplus \ldots \oplus F_{n-1} \bullet x_{n-1}$ is computable iff the sequence $(F_1, F_2, \cdots, F_n)$ is a permutation of all the operator components of an *n*-ary function.

Definition 2. If a domain is composed of a single element 1, then we call the variable in the domain as **1-variable**. i.e., we consider the constant 1 as a special variable.

Definition 3. Given two variables *x* and *y*, if the value of *y* is independent of *x*, we call *x* has a **null operator component** to y, represented as 0.

The arithmetic operators of IMP are defined as follows:

(1) $\varepsilon_n(x) \triangleq n \times x$, where n is a constant in **N.** Especially, we call ε_1 as **identical operator component,** which can also be represented as 1, but note that here ε_0 is different from null operator component 0.

(2) $\alpha(x,y) \triangleq x+y$. (3) $\beta(x,y) \triangleq x \times y$.

x-y can be derived from α and ε_{-1}, i.e.,

$x-y=x+(-1)\times y=\alpha(x, \varepsilon_{-1}(y))=\alpha_0 \cdot x \oplus (\alpha_1 \cdot \varepsilon_{-1}) \cdot y$.

Property 1. For $a \in$ **Aexp**[1], suppose x_1, ..., x_n are variables in a, then it can be represented in the form of

$a=F_0 \cdot 1 \oplus F_1 \cdot x_1 \oplus F_2 \cdot x_2 \oplus ... \oplus F_n \cdot x_n$,

where F_0, F_1, F_2, ..., F_n are operator component expressions.

For instance, $x \times (2 \times y+5)=(\beta_1 \cdot \alpha_1 \cdot \varepsilon_5) \cdot 1 \oplus \beta_0 \cdot x \oplus (\beta_1 \cdot \alpha_0 \cdot \varepsilon_2) \cdot y$

Property 2. For any $F, G, H \in \Re$, $m,n \in$ N

- $F \cdot 0=0 \cdot F=0$, $F \cdot 1=1 \cdot F=F$, $F \oplus 0=0 \oplus F=F$
- $\varepsilon_m \cdot \varepsilon_n=\varepsilon_{m \times n}$

2.2 Presentation of Commands

In IMP programs, there are five types of commands: skip statement (**skip**), assignment statement (x:=a), compound statement (c_0;c_1), branch statement (**if** b **then** c_0 **else** c_1) and loop statement (**while** b **do** c). Here we present these statements in the form of OCM.

Assignment statement

Suppose the variable set of a program is $\{x_1, ..., x_n\}$, then from **property 1** and **definition 3**, each $a \in$ Aexp takes the form of $F_0 \bullet 1 \oplus F_1 \bullet x_1 \oplus ... \oplus F_n \bullet x_n$, where F_k=0 if variable x_k does not occur in a.

So, we can denote assignment statement x_k:=a_k (k=1..n) as

$$x_k := F_{k0} \bullet 1 \oplus F_{k1} \bullet x_1 \oplus ... \oplus F_{kn} \bullet x_n$$

or x_k:=$[F_{k0}\ F_{k1}\ ...\ F_{kn}][1\ x_1\ ...\ x_n]^T$.

When the program executes the assignment statement x_k:=a_k, only the value of variable x_k will be changed. So we can consider that it implies such assignment statements as follows: x_j:=x_j (j=1,..n, $j \neq k$),

i.e., $x_j := 0 \bullet 1 \oplus 0 \bullet x_1 \oplus ... \oplus 1 \bullet x_j \oplus \cdots \oplus 0 \bullet x_n$

Let $\mathbf{X}=[1\ x_1\ x_2\ ...\ x_n]^T$, then x_k:=a_k is equivalent to

$$\mathbf{X} := \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \\ F_{k0} & F_{k1} & \cdots & F_{kn} \\ \vdots & & & \\ 0 & 0 & \cdots & 1 \end{bmatrix} \mathbf{X}$$, represented as **X**:=**CX**

C is a matrix composed of operator components, such matrix is called as **operator component matrix,** abbreviated as OCM. We can treat **C** as a **state transition matrix**.

Property 3. The first row of any state transition matrix has the form of [1 0 ... 0].

Property 4. In any row of any state transition matrix, there is at least one operator component that is not 0.

We label the commands with 1, 2, ..., n by their occurrence order in a program. Let $\mathbf{X}^{(k-1)}$ be the program state before the execution of the k-th command, and $\mathbf{X}^{(k)}$ be the program state after the execution of the k-th command, then $\mathbf{X}^{(k)}=\mathbf{C}_k\mathbf{X}^{(k-1)}$, where $\mathbf{C}_k$ is the state transition matrix of the k-th command. The value of an expression a under program state $\mathbf{X}^{(k)}$ is represented as $(a, \mathbf{X}^{(k)})$.

Skip statement

In a unit matrix, if we look number 1 as operator component 1, and number 0 as operator component 0, we call such matrix as **unit OCM**, represented as **E**. We also call the OCM composed of all 0 as **zero OCM**, represented as **O**.

So, skip statement (**skip**) can be represented as **X**:=**EX** .

Presentation of compound statement

Compound statements take the form of c_0;c_1. Suppose command c_0 can be represented as **X**:=$\mathbf{C_0X}$ and c_1 can be represented as **X**:=$\mathbf{C_1X}$, where $\mathbf{C}_0$ and $\mathbf{C}_1$ are all OCMs, then we can represent the command as **X**:= $(\mathbf{C_1C_0})\mathbf{X}$.

Presentation of branch statement

Branch statements take the form of **if** b **then** c_0 **else** c_1. Suppose command c_0 can be represented as **X**:=$\mathbf{C_0X}$ and c_1 can be represented as **X**:=$\mathbf{C_1X}$, where $\mathbf{C}_0$ and $\mathbf{C}_1$ are all OCMs. Then we can represent the command as **X**:= $(m\mathbf{C_0} \oplus (1\text{-}m)\mathbf{C_1})\mathbf{X}$, where m=1 when $(b, \mathbf{X}^{(0)})$=**true**, otherwise m=0, $\mathbf{X}^{(0)}$ is the program state before the execution of the command. Note that we look the result of m and 1-m as operator component 1 or 0.

Presentation of loop statement

Loop statements take the form as **while** b **do** c. Suppose command c can be represented as **X**:=**CX**, where **C** is an OCM. Then we can denote the command as **X**:= $\mathbf{C}^n\mathbf{X}$, where n is the number of executions of loop body c (i.e. n= min{k|(b, $\mathbf{X}^{(k)}$) =**false**, $k \geq 0$}, where $\mathbf{X}^{(k)}$ is the program state after the k-th execution of command c).

According to structural induction, it can be proved that any IMP program can be denoted in the form of **X**:=**CX**. We call **C** as the OCM model of the program.

2.3 An example

To facilitate analysis, we use an example shown in Fig 1.

```
1(key point)    if (from ≤ to) then
1.1               {start := from ;
1.2                stop := to}
                 else
1.3               {start := to;
1.4                stop := from};
2                 i:= start;
3                 s:= 0;
4(key point)      while (i ≤ stop ) do
4.1               {s:=s+i;
4.2                i:= i+1}
5                 [end]
```

Figure 1: An IMP program

Let $\mathbf{X}=[1\ from\ to\ start\ stop\ i\ s]^T$, we can get the OCM model of the program

$\mathbf{C}=\mathbf{C_4C_3C_2C_1}=(\mathbf{C_{4.2}C_{4.1}})^n\mathbf{C_3C_2}(m\mathbf{C_{1.2}C_{1.1}} \oplus (1\text{-}m)\mathbf{C_{1.4}C_{1.3}})$

[1] **Aexp** is the set of arithmetic expressions in IMP.

$$= \begin{bmatrix} 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0 \\ \alpha_1&0&0&0&0&\alpha_0&0 \\ 0&0&0&0&0&\alpha_1&\alpha_0 \end{bmatrix}^n \begin{bmatrix} 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0 \\ 0&m&1-m&0&0&0&0 \\ 0&1-m&m&0&0&0&0 \\ 0&m&1-m&0&0&0&0 \\ \varepsilon_0&0&0&0&0&0&0 \end{bmatrix}$$

where C_l is the state transition matrix of the command labeled with l; m=1 if (*from*≤*to*, $\mathbf{X}^{(0)}$=**true** or else m=0; n=min{k|($i \le$ *stop*, $\mathbf{X}^{(4.2,k)}$=**false**, k≥0}, where $\mathbf{X}^{(0)}$ is the program state before execution of the command labeled with 1; $\mathbf{X}^{(4.2,k)}$ is the program state after the k-th execution of the command labeled 4.2.

If a program includes branch statements or loop statements, there are some unknown "constants" appearing in its OCM model, such as m, n in the above example. Though their values are dependent on a given test case, we still look them as formal constants.

3 Dependency Analysis

3.1 Black-box Dependency

Definition 4. Such an operator component λ is called a **characteristic operator component**, if it satisfies that

λ·λ=λ, λ⊕λ=λ, λ⊕0=0⊕λ=λ, λ·0=0·λ=λ.

An OCM **K** is called the **characteristic matrix** of **C**, if **K** is constructed by replacing all the operator components in **C** except 0 with λ .

Theorem 1. For any OCM $\mathbf{C}_1$, $\mathbf{C}_2$, if $\mathbf{K}_1$ and $\mathbf{K}_2$ are respectively the characteristic matrix of $\mathbf{C}_1$ and $\mathbf{C}_2$, then $\mathbf{K}_1\mathbf{K}_2$ is the characteristic matrix of $\mathbf{C}_1\mathbf{C}_2$.

Theorem 2. For any characteristic matrix **K ,** the chain **K**, $\mathbf{K}^2$, $\mathbf{K}^3$, ..., $\mathbf{K}^n$ satisfies one of the two cases as below:

(1) ∃N, ∀n>N, $\mathbf{K}^n$=$\mathbf{K}^N$;

(2) ∃N, M, ∀n> N**,** $\mathbf{K}^{n+M}$=$\mathbf{K}^n$**.**

Since the set of all characteristic matrixes of a given dimension is limited, any chain **K**, $\mathbf{K}^2$, $\mathbf{K}^3$, ..., $\mathbf{K}^n$ must reach a fixed point or enter into a ring. The theorem indicates that the dependency in a loop is definite.

Definition 5. Assume that **X** is the vector of all variables in program[2] P **,** **C** is the OCM model of P, **K** is the characteristic matrix of **C**, then **K** is called the **black-box dependency matrix** of P.

From the example shown in Figure 1, we can get its black-box dependency matrix

$$\mathbf{K} = \begin{bmatrix} \lambda&0&0&0&0&0&0 \\ 0&\lambda&0&0&0&0&0 \\ 0&0&\lambda&0&0&0&0 \\ 0&\lambda&\lambda&0&0&0&0 \\ 0&\lambda&\lambda&0&0&0&0 \\ \lambda&\lambda&\lambda&0&0&0&0 \\ \lambda&\lambda&\lambda&0&0&0&0 \end{bmatrix}.$$

From **K**, we can get the dependencies of the final values of variables to the initial values of variables; we represent these dependencies as $\mathbf{X}^{(n)} \xleftarrow{B} \mathbf{K}\mathbf{X}^{(0)}$, where $\mathbf{X}^{(n)}$ is the final program state, $\mathbf{X}^{(0)}$ is the initial program state. For instance, we extend the variable s, we can get

$s^{(n)} \xleftarrow{B} \lambda\cdot1\oplus\lambda\cdot from^{(0)}\oplus\lambda\cdot to^{(0)}$,

which means the final value of s is decided by the initial value of *from*, the initial value of *to*, and the constant value of s assigned in the program. Especially, if the black-box dependency of a variable includes the item λ·1, it shows the variable has been assigned with a constant in the program.

In the black-box dependency, if we find a variable does not appear in the dependencies of any other variables and its final value is not concerned about, the variable is usually considered useless, which can be deleted from the program.

3.2 White-box Dependency

Definition 6. For an OCM **C**, after replacing all its operator components except 0 and 1 with **formal operator component** represented as δ[p](p is the label set of the commands where the actual operators occur), the new components matrix **H** is called as the **formal matrix** of **C.** The properties of δ[p] are shown as follows:

$\forall p_1,p_2$, $\delta[p_1]\oplus\delta[p_2]=\delta[p_1\cup p_2]$, $\delta[p_1]\cdot\delta[p_2]=\delta[p_1\cup p_2]$,
$\delta[p_1]\oplus0=0\oplus\delta[p_1]=\delta[p_1]$, $\delta[p_1]\cdot0=0\cdot\delta[p_1]=0$,
$\delta[p_1]\oplus1=1\oplus\delta[p_1]=\delta[p_1]$, $\delta[p_1]\cdot1=1\cdot\delta[p_1]=\delta[p_1]$.

Definition 7. Assume that **X** is the vector of all variables in program P **,** **C** is the model of P, **H** is the formal matrix of **C**, then we call **H** is the **white-box dependency matrix** of P.

From the example shown in Figure 1, we can get its white-box dependency matrix

$$\mathbf{H} = \begin{bmatrix} 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0 \\ 0&\delta[1.3]&\delta[1.1]&0&0&0&0 \\ 0&\delta[1.4]&\delta[1.2]&0&0&0&0 \\ \delta[4.2]&\delta[1.1,2,4.2]&\delta[1.3,2,4.2]&0&0&0&0 \\ \delta[4.1,4.2,3]&\delta[1.1,2,4.1,4.2]&\delta[1.3,2,4.1,4.2]&0&0&0&0 \end{bmatrix}$$

From **H**, we can get the dependency of variables to commands which is represented as $\mathbf{X}^{(n)} \xleftarrow{W} \mathbf{H}\mathbf{X}^{(0)}$, where $\mathbf{X}^{(n)}$ is the final program state, $\mathbf{X}^{(0)}$ is the initial program state. For any variable in **X**, we can get its dependency to commands from **H**. For instance, we extend the variable s, and get

$s^{(n)} \xleftarrow{W} \delta[4.1,4.2,3]\cdot1\oplus\delta[1.1,2,4.1,4.2]\cdot from^{(0)}\oplus$
$\delta[1.3,2,4.1,4.2]\cdot to^{(0)}$.

Then we establish a set δ_s which is composed of all the command labels appearing in the right part, i.e.,

δ_s = {1.1,1.3,2,3,4.1,4.2}.

We call δ_s as the white-box dependency of variable s, which means the final value of s is decided by the command set {1.1,1.3,2,3,4.1,4.2}. It can be proved to be equivalent with the program slice about the slicing criterion (s, 5) [Wotawa, 2002]. Note that white-box dependencies include data dependencies and control dependencies which are implied in the formal constants related to a branch statement or a loop statement.

[2] Here 'program' indicates IMP program without special note.

3.3 Complexity analysis

In this model, the dimension of each OCM is $(n+1)\times(n+1)$ where n is the number of variables in the program. It seems that the size of the matrix model would grow quickly with the number of variables. However, it is not a big problem, because when we diagnose a program, at one time, a diagnosed object is generally a program module (a procedure or a function) where the number of variables won't be too big. Moreover, we can store all state transition matrixes in the dense mode of sparse matrix.

4 Model-based Program Diagnosis

4.1 Isomorphism Assumption and Abstract Program

Definition 8. Two commands c and c' are **isomorphic,** represented as Isomorph(c, c'), iff one of the following cases is satisfied:

1) c::=**skip** $\wedge$ c'::=**skip;**
2) c::= x:=a_0 $\wedge$ c'::= y:=a_1 $\wedge$ $x \equiv y$ $\wedge$ VS(a_0) = VS(a_1) where $x \equiv y$ means that x and y are the same variable, VS(a) means the set of all the variables in a;
3) c::= c_0; c_1 $\wedge$ c'::= c_0'; c_1' $\wedge$ Isomorph(c_0, c_0') $\wedge$ Isomorph(c_1, c_1');
4) c::= **if** b **then** c_0 **else** c_1 $\wedge$ c'::= **if** b' **then** c_0' **else** c_1' $\wedge$ Isomorph(c_0, c_0') $\wedge$ Isomorph(c_1, c_1');
5) c::= **while** b **do** c_0 $\wedge$ c'::= **while** b' **do** c_0' $\wedge$ Isomorph(c_0, c_0').

Generally, a diagnosed program should be close to an expected program. That is to say, it is possible that minor revisions will make the diagnosed program correct. So, we always suppose that for any diagnosed program, there exists a "correct" program which is isomorphic with the diagnosed program. This assumption is called **isomorphism assumption**.

In some diagnosed programs, it is possible that some variable is lost in the right-side expression of an assignment statement, such that the isomorphism assumption cannot hold any more. But fortunately, we can extend the expression to a new equivalent expression which includes the lost variable by adding a null operator component. For instance, if x:=x+i is written as x:=x+1 by mistake, we can extend the expression x+1 to x+1$\oplus$0$\cdot i$. We call this case as **extendable isomorphism**, so it assures that many diagnosed programs are at least extendable isomorphic with a correct program.

Definition 9. For a given program (OCM model), if we replace all the operator components of the program with abstract operator components, the new program is called an **abstract program**. An abstract operator component is only a marker which does not refer to any concrete operator component.

For example, there is a command l: s:=s+i in a program (where l is the label), whose model is represented as s:= $\alpha_0^{(l)}\cdot s\oplus\alpha_1^{(l)}\cdot i$, and whose abstract model is represented as s:= $f_0^{(l)}\cdot s\oplus f_1^{(l)}\cdot i$, where $f_0^{(l)}$ and $f_1^{(l)}$ are abstract operator components. Suppose the mapping from the abstract model to the actual model is MAP, then MAP=$\{\{(f_0^{(l)}, \alpha_0^{(l)}), (f_1^{(l)}, \alpha_1^{(l)})\}\}$. In order to identify the operator components which have the same form, we add the command label to them. Suppose the command l: s:=s+i should be l: s:= $s\times i$, it is known that the abstract model of the "correct" command has the same form of the abstract model of the "wrong" command. Given a mapping MAP′=$\{(f_0^{(l)}, \beta_0^{(l)}), (f_1^{(l)}, \beta_1^{(l)})\}$, we can get the correct command from the abstract model.

Under the isomorphism assumption, we can construct the abstract model of expected program from the concrete model of the diagnosed program by replacing components in the concrete model with abstract components.

Note that each extended null operator component must be replaced by an abstract operator component in the abstract model.

4.2 Program Specification and Breakpoint

Program specification

To further describe the abstract program, we need offer some assertions about the static properties of it, each assertion is called a program specification [Morgan, 1998].

Definition 10. A **program specification** has the form as below:

(*precond*(**X**), *frame*(**X**), *postcond*(**X**)),

where **X** is the set of variables, *precond*(**X**) is a description of the initial state of **X**, *frame*(**X**) indicates a piece of the abstract program including **X**, and *postcond*(**X**) gives a description of the final state of **X** after the "execution " of the piece of abstract program.

Note that *frame*(**X**) cannot be executed until a concrete implementation is given. If *slice*(**X**) is a concrete implementation of *frame*(**X**), *postcond*(X) must come true after execution of *slice*(**X**) under the condition *precond*(**X**), and we denote it as {*precond*(**X**)} *slice*(**X**){ *postcond*(**X**)}.

For example, to such a specification,

precond(s, i)={s>0, i>0},
frame(s, i)={ s:= $f_0\cdot s\oplus f_1\cdot i$},
postcond(s, i)={s>1},

slice {s:=s+i} is an implementation of *frame*(s,i), but slice {s:=$s\times i$} is not .

Breakpoint

The position of a program is called a **breakpoint,** if where there is an assertion. The assertions about the variables in a breakpoint are called **intermediate results**. The value of variable x in a breakpoint B is represented as x(B).

In program diagnosis, a kind of special position is often used as breakpoints, and those positions are closely followed with a boolean expression, we call them as **key points.** For example, in Figure 1, there are two key points in the positions of *label* 1 and *label* 4.

Note that in a runtime of a program, a breakpoint may be passed by several times or zero times, but it corresponds to a unique position in a program.

For a given breakpoint, if all test cases should pass it, we call the assertions in this breakpoint as **invariant ("always") assertion**, otherwise, we call them as **intermittent ("sometime") assertion [** Mayer and Stumptner, 2003**]**.

So, it is known that any program specification must relate to two breakpoints, and the program code between the two breakpoints must be a legal (compound) command. For example, in Figure 1, the codes between breakpoint 1.3 and breakpoint 3 are not legal commands. If B_1 and B_2 are the breakpoints related to frame(**X**), we also represent frame(**X**) as frame(B_1,B_2, **X**). If **X** includes all the variables of the program, **X** can be omitted.

4.3 Program Diagnosis

Definition 11. A **program diagnosis problem** is a triple (**SD**, **COMPS**, **SPECS**) where **SD** is the OCM model of the actual program to be diagnosed, **COMPS** is the set of operator components in **SD**, **SPECS** is the set of program specifications.

Definition 12. Suppose (**SD**, **COMPS**, **SPECS**) is a program diagnosis problem. Let **AD** be the abstract program of **SD**, **ACOMPS** be the abstract operator components in **AD**, and **MAP** be the mapping from **COMPS** to **ACOMPS**. Then $\Delta \subseteq$ **COMPS** is a **diagnosis** iff there exists **MAP′**: SA$\rightarrow\sum$ where $\sum$ is the set of all operator components and SA is a subset of **ACOMPS**, such that

- **AD** $\cup$ **MAP′** $\cup$ **SPECS** is consistent, and
- $\forall C \in$ **COMPS**$\backslash\Delta$, **MAP′** (**MAP**$^{-1}$(C))=C.

Under isomorphism assumption, **AD** $\cup$ **SPECS** is always consistent, that is, the expected program is an interpretation of **AD**.

Since any operator component corresponds to a command in the program, components in Δ indicate the possible faults in the program.

Definition 13. Let Δ be a diagnosis. If there exists no $\Delta' \subset \Delta$ such that Δ' is also a diagnosis, then Δ is a **minimal diagnosis**.

Definition 14. Suppose (**SD**, **COMPS**, **SPECS**) is a program diagnosis problem. Let **AD** be the abstract program of **SD**, **ACOMPS** be the abstract operator components in **AD**, and **MAP** be the mapping from **COMPS** to **ACOMPS**. Then **C**$\subseteq$**COMPS** is a **conflict set** iff for any **MAP′**: **ACOMPS**$\rightarrow\sum$ where **MAP′**(**C**)=**MAP**(**C**), **AD** $\cup$ **MAP′** $\cup$ **SPECS** is inconsistent.

Definition 15. Let **C** be a conflict set. If there exists no **C'**$\subset$**C** such that **C'** is also a conflict set, then **C** is a **minimal conflict set**.

Theorem 3. If $\mathbf{C}_1$, $\mathbf{C}_2$,... $\mathbf{C}_n$ are all minimal conflict sets of program *P*, then all the least hitting sets of {$\mathbf{C}_1$, $\mathbf{C}_2$,... $\mathbf{C}_n$} are minimal diagnoses of *P*.

The definitions about hitting set can be found in [Lin *et al*., 2003].

Theorem 4. Suppose *PS*=(*precond*(**X**), *frame*(B_1,B_2), *postcond*(**X**)) is a program specification. Let *slice*(B_1,B_2) be the implementation of *frame*(B_1,B_2) in the diagnosed program *P*. Given a program input, if (1) the program state in B_1 is consistent with *precond*(**X**), (2) **Y** is the set of variables whose states in B_2 are inconsistent with *postcond*(**X**) , and (3) **C** is the white-box dependency set of **Y** in *slice*(B_1,B_2), then **C** is a conflict set.

Theorem 5. Suppose *PS*=(*precond*(**X**), *frame*(B_1,B_2), *postcond*(**X**)) be a program specification. Let *slice*(B_1,B_2) be the implementation of *frame*(B_1,B_2) in the diagnosed program *P*. Suppose there is an equation $y=f(\mathbf{W}^{(0)})$ in *postcond*(**X**), where $y \in \mathbf{X}$, $\mathbf{W} \subseteq \mathbf{X}$, and $\mathbf{W}^{(0)}$ denotes the state of **W** in B_1. If the black-box dependency set of *y* in *slice*(B_1,B_2) is not equal to **W,** then the white-box dependency set of *y* in *slice*(B_1,B_2) is a conflict set.

The conclusions of theorem 4 and 5 are obvious; theorem 5 is a special case of theorem 4.

4.4 A Diagnosis Example

Program diagnosis problem

Consider the program shown in Figure 1. Suppose command 4.1 is written as **s**:=**s**×**i** by mistake. Then the program diagnosis problem is given as below:

SD=(X,C),
where $\mathbf{X}=[1\ \textit{from}\ \textit{to}\ \textit{start}\ \textit{stop}\ i\ s]^T$
$\mathbf{C}=\mathbf{C}_4\mathbf{C}_3\mathbf{C}_2\mathbf{C}_1$
$= (\mathbf{C}_{4.2}\mathbf{C}_{4.1})^n\mathbf{C}_3\mathbf{C}_2(m\mathbf{C}_{1.2}\mathbf{C}_{1.1} \oplus (1-m)\mathbf{C}_{1.4}\mathbf{C}_{1.3})$
where C_l denotes the OCM model of the command labeled with *l* in the diagnosed program;

COMPS=$\{1^{(1.1)}, 1^{(1.2)}, 1^{(1.3)}, 1^{(1.4)}, 1^{(2)}, \varepsilon_0^{(3)}, \beta_0^{(4.1)}, \beta_1^{(4.1)}, \alpha_0^{(4.2)}, \alpha_1^{(4.2)}\}$;

SPECS={PS_1**,** PS_2**,** PS_3**},** where
PS_1=({*from*≤*to*}, frame(1,3), {*i*=*from*}),
PS_2=({*from*>*to*}, frame(1,3), {*i*=*to*}),
PS_3=({*i*≤*stop*}, frame(3,5),
{$i=stop^{(0)}+1$, $s=(i^{(0)}+stop^{(0)})\times(i-i^{(0)}/2)$})
where $i^{(0)}$ and $stop^{(0)}$ denote respectively the initial values of *i* and *stop* before the execution of the frame.

Diagnosis

AD=(X,F),
where $\mathbf{X}=[1\ \textit{from}\ \textit{to}\ \textit{start}\ \textit{stop}\ i\ s]^T$
$\mathbf{F}=\mathbf{F}_4\mathbf{F}_3\mathbf{F}_2\mathbf{F}_1$
$=(\mathbf{F}_{4.2}\mathbf{F}_{4.1})^n\mathbf{F}_3\mathbf{F}_2(m\mathbf{F}_{1.2}\mathbf{F}_{1.1} \oplus (1-m)\mathbf{F}_{1.4}\mathbf{F}_{1.3})$
where F_i denotes the abstract OCM model of the command labeled with *l* in the expected program;

ACOMPS=$\{f^{(1.1)}, f^{(1.2)}, f^{(1.3)}, f^{(1.4)}, f^{(2)}, f^{(3)}, f_0^{(4.1)}, f_1^{(4.1)}, f_0^{(4.2)}, f_1^{(4.2)}\}$;

MAP={$f^{(1.1)} \rightarrow 1^{(1.1)}$, $f^{(1.2)} \rightarrow 1^{(1.2)}$, $f^{(1.3)} \rightarrow 1^{(1.4)}$, $f^{(2)} \rightarrow 1^{(2)}$, $f^{(3)} \rightarrow \varepsilon_0^{(3)}$, $f_0^{(4.1)} \rightarrow \beta_0^{(4.1)}$, $f_1^{(4.1)} \rightarrow \beta_1^{(4.1)}$, $f_0^{(4.2)} \rightarrow \alpha_0^{(4.2)}$, $f_1^{(4.2)} \rightarrow \alpha_1^{(4.2)}$ **}**

Given a set of inputs of the diagnosed program **INPUT**={*from*=6, *to*=3} (i.e., *from*(1)=6 and *to*(1)=3) which satisfy the precondition of PS_2, by running the diagnosed program with the inputs, we can get the results of *i*(3)=3 and *to*(3)=3, which are consistent with the postcond of PS_2. In this case, we cannot get a conflict set.

We go on checking the other program specifications. By observing the values of the variables of the precondition of PS_3 in breakpoint 3, we get *i*(3)=3 and *stop*(3)=6, which satisfy the precondition of PS_3. And after applying PS_3, *s*(5) should be (3+6)×(7-3)/2=18. But *s*(5) equals to 360 actually, so there is a conflict. In *frame*(3,5), the white-box dependency of *s* is {3, 4.1, 4.2} which makes a conflict set. If we go on observing *i*, *i*(5) is always consistent with the postcondition of PS_3, so the white-box dependency of *i* (i.e., {4.2}) must not be included in a conflict set. If we have another

assertion to assure that statement 3 is right, we can finally get the diagnosis {4.1}.

We can correct the faults by mutating some operator components in the diagnoses, because the number of operator components is very limited except the operator components ε_m ($m \in N$). Generally, we can assume that $Max \geq m \geq 0$ where *Max* is a finite number given in advance. When the mutations $f_0^{(4.1)} \rightarrow \alpha_0^{(4.1)}$, $f_1^{(4.1)} \rightarrow \alpha_1^{(4.1)}$ take place, we can find the correct program.

5. Related Work

There are a lot of works developed in model-based program diagnosis during the years, among which the most important work is about program slicing [Wotawa, 2002]. Our model gives a new and simpler method of computing program slices (i.e. white-box dependency).

In the presentation of the abstract model of the expected program, in order to improve the diagnosis capabilities, [Chen *et al*., 2005] also uses assertions to specify an abstract model. But it does not deal with abstraction in program structure and it only works on the value-based model. Our model has stronger presentation capability, which uses more information about the existing program structure under the isomorphism assumption.

In correcting the faults, [Wotawa, 2001] also discusses a different mutation method, which cannot assure us of finding the correct program by mutation since the mutation scope is too large.

We have also developed the presentation of an array in the form of operator components. For example, array **x** with $\mathbf{x}[0]=a$, $\mathbf{x}[1]=b$, $\mathbf{x}[2]=c$ can be represented as $\mathbf{x}=[0]\cdot a \oplus [1]\cdot b \oplus [2]\cdot c$ where [0], [1] and [2] are all operator components; $\mathbf{x}[0]:=a$ can be represented as $\mathbf{x} := [0]\bullet a \oplus [\bar{0}]\bullet \mathbf{x}$ under the property $[\bar{n}]\bullet[n] = 0$, $[\bar{n}]\bullet[m] = [m]$ $(n \neq m)$; expression $\mathbf{x}[0]$ can be represented as $\{0\}\cdot\mathbf{x}$ where $\{0\}$ is an operator component under the property $\{n\}\bullet[n] = 1$, $\{n\}\bullet[m] = 0$ $(n \neq m)$. Due to the space limitation, we don't discuss it in detail.

6. Conclusion

This paper presents a new model for IMP programs with OCM. OCM model and isomorphism assumption are the basis of abstract programs, and we use the assertions of program specifications to describe the abstract programs. Acquisition of program specifications is a key problem, which needs more knowledge about the expected program. Obviously, the more specifications are achieved, the more accurate diagnoses are.

OCM model has a uniform presentation, which comply with the rules for matrix operations. We can got the dependency easily from the OCM model by using characteristic operator component and formal operator component.

Due to the space limitation, we cannot discuss more diagnosis cases in detail. For example, we do not consider the diagnosis for the condition parts of branch statements and loop statements. Generally, we put these conditions into the program specifications.

Moreover, by using black-box dependency and extending null operator components, we can find some structural faults such as variable missing. In dependency models, structural faults always result in the mistakes of program slicing [Wotawa, 2002], thereby result in wrong diagnoses.

References

[Chen *et al*., 2005] Rong Chen, Daniel Koeb and Franz Wotawa. Abstract Model Refinement for Model-Based Program Debugging.16th International Workshop on Principles of Diagnosis (DX 05), Monterey, California, USA, June 1-3, 2005.

[Console *et al*., 1993] L.Console, G.Friedrich, D.Theseider Dupre. Model-based Diagnosis Meets Error Diagnosis in Logic Programs, in Proc. 13th International Joint Conference on Artificial Intelligence (IJCAI) Chambery, M.Kaufmann, pp. 1494-1499, 1993.

[Lin *et al*., 2003] L. Lin, Y.F. Jiang. The computation of hitting sets: review and new algorithms. Information Processing Letters, 86 (4): 177-184, 2003.

[Mateis *et al*., 1999] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java Programs using a Model-Based Approach. In Proc. 10th Int'l Workshop on Principles of Diagnosis, Loch Awe, Scotland, 1999.

[Mateis *et al.*, 2000] Crisinel Mateis, Markus Stumptner, Franz Wotawa. A Value-Based Diagnosis Model for Java Programs. In Proceedings of the Eleventh International Workshop on Principles of Diagnosis, Morelia, Michoacen, Mexico, June 8-10, 2000.

[Mayer and Stumptner, 2003] Wolfgang Mayer and Markus Stumptner. Model-Based Debugging using Multiple Abstract Models. In Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging, AADEBUG '03, pages 55-70, Ghent, September 2003.

[Morgan, 1998] Carroll Morgan. Programming from Sepcification, Second Edition (ISBN: 7-111-10847-7). Prentice Hall International, Hempstead, UK, 1994.

[Reiter, 1987] R Reiter. A Theory of Diagnosis from First Principles. Artificial Intelligence, 1987, 32(1): 57-95.

[Wieland, 2001] Dominik Wieland. Model-Based Debugging of Java Programs Using Dependencies. PhD thesis, Technische Universitat Wien, November 2001.

[Winskel, 1993] Glynn Winskel, The Formal of Programming Languages: An Introduction (ISBN:0-262-23169-7). Massachusatts Institute of Technology, 1993.

[Wotawa, 2001] Frantz Wotawa. On the Relationship between Model-based Debugging and Program Mutation, Proceedings of the Twelfth International Workshop on Principles of Diagnosis, Sansicario, Italy, 2001.

[Wotawa, 2002] Franz Wotawa. On the Relationship between Model-Based Debugging and Program Slicing. Artificial Intelligence, 135(1-2):124-143, 2002.