JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2007; **19**:419–450
Published online 15 October 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.356

Research

Empirical-based recovery and maintenance of input error-correction features

Minh Ngoc Ngo*, † and Hee Beng Kuan Tan

School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 639 798, Singapore

SUMMARY

Most information systems deal with inputs submitted from their external environments. In such systems, input validation is often incorporated to reject erroneous inputs. Unfortunately, many input errors cannot be detected automatically and therefore result in errors in the effects raised by the system. Therefore, the provision of input error-correction features (IECFs) to correct these erroneous effects is critical. However, recovery and maintenance of these features are complicated, tedious and error prone because there are many possible input errors during user interaction with the system; each input error, in turn, might result in several erroneous effects. Through empirical study, we have discovered some interesting control flow graph patterns with regard to the implementation of IECFs in information systems. Motivated by these initial findings, in this paper, we propose an approach to the automated recovery of IECFs by realizing these patterns from the source code. On the basis of the recovered information, we further propose a decomposition-slicing technique to aid the maintenance of these features without interfering with other parts of the system. A case study has been conducted to show the usefulness of the proposed approach. Copyright © 2007 John Wiley & Sons, Ltd.

Received 26 October 2006; Revised 28 June 2007; Accepted 31 July 2007

KEY WORDS: input error-correction features; automated recovery; software maintenance; decomposition slicing; empirical properties; information systems

1. INTRODUCTION

Automated recovery of functional features plays an important role in many software engineering activities, especially software comprehension, testing, maintenance and evolution [1] due to the nature of software life cycle and outdated documentation. By *functional feature*, we refer to

*Correspondence to: Minh Ngoc Ngo, School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 639 798, Singapore.
†E-mail: ngom0002@ntu.edu.sg

'a coherent and identifiable bundle of system functionalities that helps characterize the system from the user perspective' [2].

Information systems constitute one of the largest and most important software domains in the world. In many information systems, a major and important component is processing inputs submitted from its external environment to update its database. Currently, input validation is the most popular means to enforce the accuracy of the user input. However, many input errors are detected only after the completion of execution. We refer to this type of input errors as *after-effect input errors*. For example, when a student submits a registration request to an online course registration system, if he/she enters and submits an extra subject by mistake, there is no means to detect such a mistake automatically and reject the extra subject. In this case, input error-correction features should be provided so that the student can delete the extra subject submitted by mistake. As after-effect input error is unavoidable, the provision of after-effect input error-correction features (IECFs) is extremely important in any information system. Any omission of the provision of these features will lead to serious adverse impact. Within the scope of this paper, we will use the term 'input error' to refer to an after-effect input error.

The first step towards maintaining a certain feature is to understand how the feature is implemented in a system. This is a critical problem in program understanding, especially when the understanding is directed to a certain goal like maintaining the feature. Before being able to understand a feature, one has to locate the implementation of the feature in the code. Unfortunately, most of the systems have a large number of components containing hundreds of line of codes. Therefore, it is not obvious which component implements a given feature. In addition, the nature of software development implies that many new or changed IECFs to be incorporated are recognized during maintenance. At this stage, the maintainers are facing the pressure to modify the application to include new features as quick as possible without introducing any adverse impact into the existing code. Unfortunately, maintenance of the IECFs is a complex, time-consuming and error-prone task as there are many types of input errors, and each of them may cause many other external erroneous effects such as updating the database with wrong information.

Recently, several researchers have developed experimental program analysis (PA) approaches [3] as a new paradigm for solving software engineering problems where traditional approaches have not succeeded. The use of empirical properties, which have been validated statistically, to solve problems is very common in the area of medicine [4]. However, the application of this method is rather unexplored in software engineering. In this paper, we present an empirical approach to the automated recovery and maintenance of IECFs from the source code. We observe that IECFs are usually implemented through a few methods. The use of these methods results in some common structural properties in the control flow graphs of (CFGs) programs which implement them. Through realizing these properties from the source code, erroneous effects resulting from IECFs provided by a system to correct these effects can be recovered. We then further propose the effect-oriented decomposition-slicing (DeSlice) technique to decompose a program with respect to the IECFs to aid software engineers in maintaining these features.

In our earlier work [5], we have reported the initial results of our approach to the recovery and maintenance of the IECFs. This paper improves the work presented in [5] in several ways. First, we present in this paper a much improved and refined set of empirical properties that enable the direct inference of erroneous effects resulting from an input error. Second, we also provide two algorithms that enable the replication and implementation of the proposed approach more easily.

Third, we enhance and extend our experiments to examine a larger variety of information systems. Last but not least, we conduct a case study to show the performance of students and programmers in manually recovering the IECFs from two web applications.

The remainder of this paper is organized as follows. Section 2 gives an overview of our approach. Section 3 establishes a theoretical background for the automated recovery of the IECFs. We present in this section our discovered structural properties of input errors and its correction features in the form of CFG properties. Section 4 proposes an approach to automatically recover IECFs based on these properties. Section 5 discusses the application of DeSlice to aid in the maintenance of the IECFs. Section 6 describes the prototype system. Section 7 validates the empirical studies and evaluates the usefulness of the proposed approach through a case study. Section 8 discusses related work. Section 9 concludes the paper.

2. OVERVIEW OF THE APPROACH

The main objective of our approach is to recover the IECFs implemented in a system and then use the recovered information to understand and aid the maintenance of these features. In this section, we present an overview of our approach and introduce the relationship between input errors, resulting effect errors (EEs) and the IECFs.

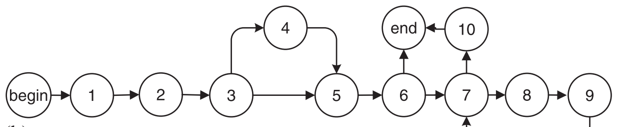
In the previous section, we have mentioned that input errors, which are detected only after the completion of the execution of the system, are referred to as after-effect input errors. In this paper, we will consider only the after-effect input errors. Next, we shall describe a real web application that will be used throughout this paper to make our approach concrete. For the ease of presentation, we simplify the application and present it in pseudo-code. ECA is an extra-curriculum activity (ECA) clubs management system developed by final-year students. The system consists of three programs: `club_registration`, `delete_club` and `delete_student`, whose pseudo-codes are given in Figures 1(a), 2(a) and 3(a), respectively. There are two database tables associated with the three programs as follows:

- STUDENT = (`student_id`, `student_name`, `ECA_points`).
- CLUB = (`club_name`, `student_id`).

The `club_registration` (Figure 1(a)) program first reads the user inputs `studentId` and `studentName` submitted through a web form by calling the function `getParameter(String param)`, which returns the value of the request parameter `param` submitted by the user. If the student is a new member, his/her record is inserted into the table STUDENT (line 4). Each time, a student can choose more than one ECA club to register. Line 5 reads all the club names that the student submits through the web form into an array by calling the function `getParameterValues(String param)`, which returns an array of values that the given parameter `param` has. For each ECA club, the student is added as the member of the club (line 8). The ECA `points` awarded by joining the club is also calculated (line 9). Function `getPoints(String clubName)` returns the `points` awarded by joining the club with the given `clubName`. Finally, his/her ECA record will be updated accordingly (line 10).

```
begin
1.  String studentId = getParameter("id"), studentName = getParameter("name");
2.  int points = 0;
3.  if (studentId does not exist in STUDENT table) then
4.   insert (studentId, studentName, points) into STUDENT table;
    endif
5.  String[] clubName = getParameterValues( "clubName");
6.  if (clubName.length > 0) then
7.   for (i = 1 to clubName.length) do
8.         insert (clubName[i], studentId) into CLUB;
9.         points = points + getPoints(clubName[i]);
     endfor
10.  update STUDENT set ECA_points = ECA_points + points where student_id = studentId;
    endif
end
```

(a)

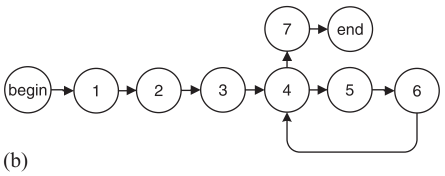


(b)

Figure 1. (a) `club_registration`'s *pseudo-code* and (b) `club_registration`'s *CFG*.

```
begin
1.  String studentId = getParameter("id");
2.  String[] clubName = getParameterValues( "clubName");
3.  int points = 0;
4.  for (i =1 to clubName.length)
5.   delete from CLUB where student_id = studentId and club_name = clubName[i]
6.   points = points + getPoints(clubName[i]);
    endfor
7.  update STUDENT set ECA_points = ECA_points - points where student_id = studentId;
end
```

(a)

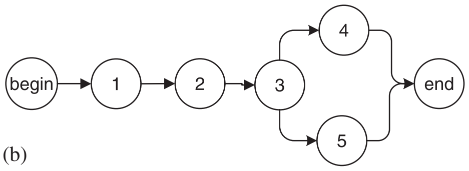


(b)

Figure 2. (a) `delete_club`'s *pseudo-code* and (b) `delete_club`'s *CFG*.

```
begin
1.  String studentId = getParameter("id");
2.  select * from CLUB where student_id = studentId;
3.  if (no record found) then
4.      delete from STUDENT where student_id = studentId;
    else
5.      print the error message "cannot delete the student"
end
```

(a)



(b)

Figure 3. (a) `delete_student`'s *pseudo-code* and (b) `delete_student`'s *CFG*.

The `delete_club` program (Figure 2(a)) removes a student from more than one ECA club (line 5) and updates his/her ECA record accordingly (line 7).

The `delete_student` program (Figure 3(a)) removes a student from the system if all the clubs registered by the student have been removed from the CLUB table (line 4).

In general, input submitted to a program can be expressed in terms of the three basic control constructs: sequence, selection and iteration. Let d and f be two elementary data items. The sequential composition of d and f, denoted by $d+f$, represents the composite data item, of which each instance (value) is an instance of d followed by an instance of f. The selection of either d or f, denoted by $[d|f]$, represents the composite data item, of which each instance is an instance of either d or f. The iteration composition of d, denoted by $\{d\}$, represents the composite data item, of which each instance is formed by multiple instances of d together. For example, the input in the `club_registration` program (Figure 1(a)) can be represented as `studentId`+`studentName`+{`clubName`}.

In a program, there are statements which when being executed will raise some external effects. In an information system, the types of effects raised vary according to its domain. Systems in one of the major domains, database applications, raise their effects by updating databases and delivering information in the form of reports, documents or interactive display to the external environment. For example, statements 4, 8 and 10 (Figure 1(a)) raise their external effect by inserting a record into table `STUDENT`, inserting a record into table `CLUB` and updating the `ECA_points` attribute of an existing record in table `STUDENT`, respectively. If these statements are influenced by some input errors, they will result in erroneous effects; we refer to these as EEs. For instance, if an extra `clubName` is submitted by mistake, it will lead to EEs in lines 8 and 10, which insert an extra record (`studentId, clubName[i]`) into the `CLUB` table and update additional ECA `points` to the student's ECA record, respectively. Correction features should be provided in this case to correct these EEs since the extra `clubName` is submitted by mistake.

An input error is correctable if all of its resulting EEs are correctable. Therefore, the objective of providing the IECFs in a system is to correct all the EEs resulting from each possible input error. Our approach recovers these features by first identifying all the EEs resulting from each possible input error occurring during system execution. We then try to recover the implementation of the correction features for correcting these EEs by trying to detect the existence of some patterns in the source code. Basically, these patterns model some common approaches in implementing the IECFs which are discovered during our empirical study. For each IECF recovered, our approach computes a decomposition slice containing all the statements that implement the feature by using the effect-oriented DeSlice technique. Various dependency relationships between decomposition slices are also computed. On the basis of this, maintainers can inspect decomposition slices and the relationship between them to introduce changes to the system. Given that the maintainer follows some rules, our approach can automatically merge these changes to the system without affecting non-related parts.

3. PROPERTIES OF INPUT ERROR-CORRECTION FEATURES

In this section, we provide a characterization of input errors and resulting EEs in a program. We also present some common CFG properties of IECFs. For invariant properties, we will provide the proof. For empirical properties, we provide the statistical validation in Section 7.1. Before proceeding further, we shall review and define some terms that will be used throughout this paper.

A CFG is a directed graph that consists of a set of N nodes and a set $E \subseteq N \times N$ of directed edges between nodes. Each node represents a program statement. Each edge (s, t) represents the flow of control from statement s to statement t. In a CFG, there is a begin node and an end node where computation starts and finishes.

Let G be a CFG of a program. A node in G where an input from user is read in (accessed) by the program is called an *input node*. Variables defined by an input node are referred to as *input variables*. A node in G at which an external effect is raised is called an *effect node* in G. For example, in Figure 1(a), nodes 1 and 5 are input nodes and nodes 4, 8 and 10 are effect nodes.

Let v and w be nodes in a CFG such that $w \neq v$. We say v *post-dominates* w $(w \neq v)$ in the CFG if every directed path from w to the end node has to pass through v. Let x and y be nodes in a CFG. Node y is *control-dependent* [6] on node x if and only if:

- there exists a directed path p from x to y such that, apart from x and y, y post-dominates each node z in p, and
- y does not post-dominate x.

An *exemplar path* (e-path) [7] in a CFG of a program is a path in the CFG such that any loop included is iterated exactly one time. In contrast to the total number of all possible paths in a CFG, the total number of e-paths in the CFG is always finite. In our approach, we need not be concerned with the exact number of times that a loop is iterated. Therefore, we will use an e-path in a CFG to represent all the paths in the CFG of a program that are identical with the e-path, except iterating some loops included in the e-path a different positive number of times. For example, in the CFG shown in Figure 1(b), the e-path through the CFG (begin, $1, 2, 3, 5, 6, 7, 8, 9, 7, 10$, end) represents

Table I. e-Paths.

Program	*e*-Path notation	*e*-Path details
`club_registration`	x_1	(begin, $1, 2, 3, 4, 5, 6$, end)
	x_2	(begin, $1, 2, 3, 4, 5, 6, 7, 8, 9, 7, 10$, end)
	x_3	(begin, $1, 2, 3, 4, 5, 6, 7, 10$, end)
	x_4	(begin, $1, 2, 3, 5, 6, 7, 10$, end)
	x_5	(begin, $1, 2, 3, 5, 6, 7, 8, 9, 7, 10$, end)
`delete_club`	y_1	(begin, $1, 2, 3, 4, 7$, end)
	y_2	(begin, $1, 2, 3, 4, 5, 6, 4, 7$, end)
`delete_student`	z_1	(begin, $1, 2, 3, 4$, end)

all the paths through the CFG in the set

$$\{(\text{begin}, 1, 2, 3, 5, 6, 7, 8, 9, 7, 10, \text{end}),$$
$$((\text{begin}, 1, 2, 3, 5, 6, 7, 8, 9, 7, 8, 9, 7, 10, \text{end}),$$
$$(\text{begin}, 1, 2, 3, 5, 6, 7, 8, 9, 7, 8, 9, \ldots 7, 10, \text{end}), \ldots\}$$

On the other hand, the *e*-path (begin, $1, 2, 3, 5, 6$, end) represents only itself as there is no loop in the *e*-path.

Consequently, *executing a program through an e-path* refers to executing the program through any path represented by the *e*-path. Within the scope of this paper, we are concerned only about *e*-paths related to the IECFs. As such, in this paper, we use the term *e*-path to refer only to those *e*-paths that contain some input and effect nodes. Table I summarizes all the *e*-paths (which contain some input and effect nodes) in three programs: `club_registration`, `delete_club` and `delete_student`, shown in Figures 1, 2 and 3, respectively. In the rest of this paper, we will use the *e*-path notations given in Table I to refer to the respective *e*-paths.

In a program, a variable *influences* a statement if the statement references to the variable. Let v be a variable used/defined by a statement x and let y be a statement that defines v where there is a path in the CFG from y to x such that v is not redefined. Variable w *influences* variable v used in x if y references to w or is control dependent on a node that references to w. For instance, variable `clubName` influences `points` used in line 10 (Figure 1(a)) because points is defined in line 9, line 9 references `clubName`, and along the path $(9, 7, 10)$ `points` is not redefined.

If variable u influences variable v and variable v influences variable w, then variable u also influences variable w. If variable w influences variable v and v influences a statement, then w also *influences* the statement.

3.1. Input errors

A maximal sequential sub-composition [8] of input variables in the input submitted to a program is called an *input block*; we use '<>' to denote an input block. The whole input submitted to a program is also an input block by itself. For example, in Figure 1(a), both <`studentId`+`studentName`+{`clubName`}> and <`clubName`> are input blocks.

Let J be an input to a program. Adapting the terms from [8], any error in J can be expressed as one or a combination of the following types of errors in J:

- Error of commission (EC) in an input block K in J: an extra instance of K has been submitted.
- Error of omission (EO) in an input block K in J: an instance of K has been omitted.
- Value error (VE) in an input block K in J: the values of elementary data items in a sub-composition in K have been submitted incorrectly.

This is due to the fact that input blocks are the only sub-compositions that can be omitted or added independently. For example, when the user submits an extra `clubName` by mistake or forgets to submit a `clubName` that he/she is supposed to register, the user has created an EC or EO in the input block < `clubName` >, respectively.

3.2. Effect errors

External effects are raised by information systems through executing effect nodes. An execution of an effect node requires a set of values to be given to the required data items. We shall call each such data item an *attribute* for executing the effect node. For example, executing an 'insert student' query in line 4, Figure 1(a), requires a set of values to be given to the student record attributes, including `student_id`, `student_name` and `ECA_points`. Each of these values is an attribute for executing the insert student effect node. Each execution leads to the insertion of a student record with the given attribute values.

Let p be an e-path through the CFG of a program that contains input nodes. Let ξ be an input error in executing p. Each error raised during the execution of an effect node in p due to the input error ξ is called an EE resulting from ξ. Errors in executing an effect node e can also be classified into EO, EC and VE as follows:

- EC in executing e: an extra execution of e has been committed.
- EO in executing e: an execution of e has been omitted.
- VE in executing e: incorrect values in some attributes for executing e.

Let V be a variable in a program. We call V a *cumulative input variable* of an input block K if V is defined in a loop and V is influenced by some input variables in K. For example, variable `points` defined in statement 9 in Figure 1(a) is a cumulative input variable of both input blocks < `studentId+studentName+{clubName}` > and < `clubName` >.

For an input error occurring during the execution of an e-path, we can derive all possible EEs resulting from the input error. The following empirical property presents the derivation.

Property 1. *Let p be an e-path through the CFG of a program. Let K be an input block of the input submitted to the program. Let e be an effect node in p. In an execution of the program through p, an error (EC or EO or VE) in K leads to the following types of EEs in executing e:*

1. *If K is the input submitted to the program, then EO or EC in K leads to EO or EC in executing e, respectively.*
2. *Otherwise, if e is in a loop whose decision node is influenced by variables in K, then EC or EO in K leads to EC or EO in executing e, respectively.*

3. *Otherwise, if some attributes for executing e are influenced by some cumulative input variables of K which are defined in some loops in p, then EC or EO in K leads to VE in these attributes in executing e.*
4. *Otherwise, if some attributes for executing e are influenced by input variables in K, then VE in K leads to VE in these attributes.*

The first sub-property is implied directly from the fact that an EO or EC in the whole input of a program leads to an omission or commission in executing the *e*-path p. Since each execution of p includes one execution of *e*, EO or EC in the input of the program leads to EO or EC in executing *e*. If *e* is in a loop and the decision node of the loop is influenced by variables in K, then the number of iterations of the loop is influenced by variables in K. Therefore, any error EC or EO in K might lead to extra iterations of the loop or some iterations left out. These imply the second property. For example, effect node 8 in Figure 1(b) is in the loop (7–9) and node 7 is influenced by the input variable clubName. Therefore, an extra value of clubName submitted will lead to an extra execution of node 7 and an omission of a value of clubName will lead to an omission of an execution of node 7.

If some attributes in executing *e* are influenced by some cumulative input variables of K, which means either EO or EC in K will lead to incorrect values of those cumulative input variables, this will in turn lead to VE in these attributes. This implies the third sub-property. For example, attributes ECA_points in executing effect node 10 in Figure 1(b) is influenced by the cumulative input variable points. An EC or EO in < clubName > will lead to a wrong value of points computed, which will then lead to an erroneous update of ECA_points in node 10.

The fourth property reflects the fact that if some input variables in K influence the attributes for executing *e*, then obviously VE in K will lead to VE in some attributes for executing *e*. For example, studentId and studentName in the input block < studentId+studentName+ {clubName}> influence the attributes student_id and student_name, respectively, in executing the effect node 4 in Figure 1(b). Therefore, VE in studentId and/or studentName will lead to VE in the respective attributes.

3.3. Input error-correction features

The objective of providing the IECFs in a system is to correct all the errors in executing effect nodes that result from erroneous inputs submitted through input nodes. Erroneous effects are raised by executing effect nodes given with a set of erroneous values to their attributes. In many information systems, erroneous effects can be corrected by executing another effect node given with a set of attribute values that can be derived from the attribute values given to the erroneous execution of the effect node. One of the largest software domains, database applications, has this property. For example, an extra execution of an effect node that inserts an extraneous record in a database can be corrected by an effect node that deletes the record inserted. Therefore, if a type of EE (EO, EC or VE) that may occur in executing an effect node *e* can be corrected by executing another effect node f in the system, we call f an *error-correction node* [8] for correcting *e*.

Without loss of generality, in this paper, we classify effect nodes into three types: insert, modify and delete. Owing to control reasons, records in some record types might not be allowed to modify or delete once they have been inserted. As such, we classify record types in a database application into: *updatable*, *modifiable* and *non-updateable*.

We classify the error-correction node f of an effect node e that operates on record type R, according to its types as follows:

(1) e is an insert node:

  (a) EO: f is also an insert node operating on R and sets the same attributes as e.
  (b) EC: If R is updateable, then f is a delete node operating on R. Otherwise, it is an insert node operating on record type S (S can be another record type or R itself) to indicate that the wrongly inserted record has been deleted.
  (c) VE: If R is updateable or modifiable, then f is a modify node operating on R to modify the attributes set by e. Otherwise, it is an insert node operating on record type S (S can be another record type or R itself).

(2) e is a modify node: For each type of error (EC, EO or IE), f is a modify node operating on R to modify the attributes set by e.
(3) e is a delete node:

  (a) EO: f is a delete node operating on R.
  (b) EC: f is an insert node operating on R.
  (c) VE: this cannot be corrected directly.

VE in executing a delete node cannot be corrected directly. This is because we have to delete the right record that is to be deleted and insert the record that is wrongly deleted.

Let p be an e-path through a program T and ξ be an input error in executing p. Let $Q=\{q_1,\ldots,q_k\}$ be a set of e-paths through some programs (T might be one of them), where $k\geq 1$. Q is called a *basis error correction set* of ξ if the following conditions satisfy:

1. All the EEs resulting from ξ can be corrected by executing the e-paths in Q.
2. For each EE there is a unique $j, 1\leq j\leq k$, such that the EE is corrected solely by executing $q_j\in Q$.

Each e-path in the set is called a *correction path* for ξ.

Let s and t be effect nodes in the CFG of a program. If s is control dependent on a decision node d if and only if t is control dependent on d, then we say that s and t are *control isomorphic*. If s and t are always in the same loop, then we say that s and t are *iteration isomorphic*.

A *partition* of a set X is a set of non-empty subsets of X such that every element x in X is in exactly one of these subsets.

Next, we present an empirical property of a set of a basis error-correction set (BECS) of an input error in executing an e-path.

Property 2. *Let p be an e-path. Let ξ be an input error in executing p. Let $\{q_1,\ldots,q_k\}$ be a set of e-paths through some programs. Then, it is highly probable that the set is a BECS of ξ if and only if there exists a partition $\{M_1,\ldots,M_k\}$ of effect nodes in p such that for each $j, 1\leq j\leq k$, there is a one-to-one mapping θ_j from M_j onto the set of effect nodes in q_j with the following properties*:

1. *For each $e\in M_j$, $\theta_j(e)$ is an error-correction node for correcting the EE resulting from ξ in executing e.*
2. *For any e_1 and $e_2\in M_j$, e_1 and e_2 are iteration isomorphic in p if and only if $\theta_j(e_1)$ and $\theta_j(e_2)$ are iteration isomorphic in q_j.*

Take, for example, the *e*-path $x_3 = (\text{begin}, 1, 2, 3, 4, 5, 6, 7, 8, 9, 7, 10, \text{end})$ in Figure 1(b). Let $S_1 =$ $<$ `studentId` $+$ `studentName` $+$ {`clubName`} $>$ be an input block of the inputs submitted to the program in Figure 1(a). From property 1, we can infer that EC in S_1 leads to EC in executing effect nodes 4, 8 and 10. Applying property 2, we can derive the basis BECS of EC in S_1 as follows:

Consider $y_2 = (\text{begin}, 1, 2, 3, 4, 5, 6, 4, 7, \text{end})$ through the CFG shown in Figure 2(b) and $z_1 =$ $(\text{begin}, 1, 2, 3, 4, \text{end})$ through the CFG shown in Figure 3(b). We can form a partition of all the effect nodes in p as follows: $M = \{M_1, M_2\}$, where $M_1 = \{4\}$ and $M_2 = \{8, 10\}$. Furthermore, θ_1 and θ_2 can be defined as follows: $\theta_1(4) = 4$ in z_1, $\theta_2(8) = 5$ in y_2 and $\theta_2(10) = 7$ in y_2. Therefore, $\{y_2, z_1\}$ is the BECS of EC in S_1. Indeed, by executing z_1, EC in inserting a record into the `STUDENT` table raised by effect node 4 in x_3 can be corrected by deleting the record from the `STUDENT` table raised by effect node 4 in z_1. By executing y_2, EC in inserting (`studentId, clubName`) into the `CLUB` table raised by effect node 8 in x_3 can be corrected by deleting (`studentId, clubName`) from the `CLUB` table raised by effect node 5 in y_2. Similarly, EC in updating a record in `STUDENT` table in node 10 in x_3 can be corrected by updating the record with correct values in node 7 in y_2.

If there exists a BECS for correcting an input error in executing an *e*-path, we say that the input error is *correctable*. For many programs, the correctability of all its input errors in an *e*-path can be deduced from the correctability of some of these errors. This means, some input errors can be corrected indirectly through the correction features provided for some other input errors. This is presented in the next invariant property.

Property 3. *If for each e-path p through the CFG of a program, EC in any input block in executing the e-path is correctable, then any input error in executing the program is correctable.*

Proof. The proof of this property is straightforward. Assuming that any EC in executing each *e*-path can be corrected, any EO in executing the *e*-path can be corrected by executing the program through the *e*-path itself. Any VE can also be corrected by treating it as an EC followed by an EO. Therefore, we execute the correction paths for correcting EC in the *e*-path followed by executing the program through the *e*-path with the correct input values. This completes the proof. □

As an illustration of this property, consider the ECA clubs management system that consists of three programs shown in Figures 1(a), 2(a) and 3(a). Table II lists the possible BECSs of EC in each *e*-path inferred by applying Property 2.

From Table II, we can see that any EC in each *e*-path in the `club_registration` program is correctable. Therefore, according to Property 3, any after-effect input error in executing the program is also correctable. Indeed, Table CI in Appendix C shows that any other input error in executing the program is correctable.

The collection of all the BECSs for correcting all the input errors in an *e*-path forms the IECF for the *e*-path. If each *e*-path through the CFG of program S is in the IECF of an *e*-path in program T, then S is called an *error-correction program* for T. It is also possible to deduce the correctability of all input errors in a program through the correctability of all input errors of some other programs in a system. We formalized this observation in the following empirical property.

Property 4. *It is highly probable that any input error of program P is correctable if and only if one of the following conditions holds*:

- *Property* 3 *infers that any input error of P is correctable.*

Table II. Basis error-correction sets for ECs.

Program	*e*-path	Input error	Basis error-correction set
club_registration	x_1	EC in S_1	$\{z_1\}$
	x_2	EC in S_1; EC in K_1	$\{y_2, z_1\}$; $\{y_2\}$
	x_3	EC in S_1	$\{y_1, z_1\}$
	x_4	EC in S_1	$\{y_1\}$ or $\{x_3\}$
	x_5	EC in S_1; EC in K_1	$\{y_2\}$; $\{y_2\}$
delete_club	y_1	EC in S_2	$\{x_3\}$ or $\{y_1\}$
	y_2	EC in S_2; EC in K_2	$\{x_5\}$; $\{x_5\}$
delete_student	z_1	EC in S_3	$\{x_1\}$

S_1 =< studentId+studentName+{clubName} >; K_1 =< clubName >
S_2 =< studentId+{clubName} >; K_2 =< clubName>
S_3 =< studentId >

- *Program P is an error-correction program for program Q and any input error of Q is correctable.*

The rationale behind the second condition of this empirical property is as follows. If P is an error-correction program for Q, any input error of P results in erroneous execution of P, which was actually aimed to correct Q. Therefore, we can treat the external effect resulting from erroneous execution of Q and P as an EC in an *e*-path through Q and the correct execution of P as an EO in another *e*-path through Q. Since any input error of Q is correctable, any input error of P is also correctable.

Programs shown in Figures 2(a) and 3(a) are error-correction programs for the program shown in Figure 1. This is because from Table II we can see that y_1, y_2 and z_1 are all in at least one BECS of EC in *e*-paths in the program in Figure 1(a). Therefore, according to the second condition of Property 4, any input error in executing the programs in Figures 2(a) and 3(a) is also correctable because any input error in executing the program in Figure 1(a) is correctable. Indeed, Table CI in Appendix C shows that there exists at least one BECS for correcting each possible input error in executing programs in Figures 2(a) and 3(a).

4. RECOVERY OF THE INPUT ERROR-CORRECTION FEATURES

On the basis of the theory established in Section 3, we develop in this section a novel approach to the automated recovery of the IECFs from the source code. By statically analyzing the source code to search for the structural properties of IECFs, the approach automatically infers and recovers the following provision for a system:

1. All the possible EEs resulting from each input error in executing each *e*-path through each program.
2. The IECF of each *e*-path including the BECSs for correcting each input error in executing the path.

3. All programs in the system whose input errors are correctable.

The proposed recovery is carried out in four steps as follows.

Step 1 (*Program analysis*): First, each program in the system is transformed into a CFG. Then, for each program, the set of all *e*-paths through its CFG is constructed by applying a depth fist search (DFS) algorithm. Note that for looping construct, the loop is either traversed once or skipped.

Step 2 (*Input errors and effect errors inference*): For each input error in executing each *e*-path, Property 1 is then applied to derive the EEs in executing each effect node in p. This step is repeated for all programs in the system before moving to the next step. The column *Effect error* in Table CI (Appendix C) shows the EEs resulting from each input error in executing each *e*-path in the ECA system.

Step 3 (*Recovery of input error-correction features*): IECFs for an *e*-path consist of a collection of BECSs of input errors in executing the path. For each *e*-path p, for each type ξ of input errors, recovery of all the BECSs is done by algorithm recover_BECS (Appendix A). The basic idea of the algorithm is that for each *e*-path q through each program in the system such that the total number of effect nodes in q is equal to or less than the number of effect nodes in p, the algorithm finds all the possible one-to-one mappings from a subset of effect nodes in p onto the set of effect nodes in q such that an effect node e in p is mapped to an effect node f in q if f is the error-correction node for correcting the EE resulting from ξ in executing e.

For each such mapping, the algorithm includes (q, θ) in the set $\Omega_{p,\xi}$. We refer to the set $\{q|(q,\theta) \in \Omega_{p,\xi}\}$ as the *candidate set of correction paths* for correcting the input error. Once $\Omega_{p,\xi}$ is fully computed, the algorithm computes all subsets K in $\Omega_{p,\xi}$ such that each EE resulting from the input error ξ is corrected by one and only one effect node in an e-path in the set $B = \{q|(q,\theta) \in K\}$. If there is no such subset K, then there is no BECS of ξ because the first condition of Property 2 does not hold. Otherwise, for each path q_k in B, let $M_k = \bigcup_{(q_k,\theta)\in K}$ (domain of θ). Actually, M_k is the set of effect nodes in p, which are corrected by the set of effect node in q_k. Let $M = \{M_k|q_k \in B\}$, then clearly M forms a partition of a BECS of the input error ξ because each EE in p is corrected by one and only one effect node in an e-path in B. Moreover, M satisfies the first condition of Property 2. If M also satisfies the second condition of property 2, then B is a BECS of ξ and it is added to $\Im_{\text{correction}}$, the set of all BECSs for correcting ξ.

As an illustration, we apply the algorithm recover_BECS to recover the BECS for correcting ξ (EC in <`studentId`+{`clubName`}>) in the e-path x_3 through the program in Figure 1. This input error leads to EC in effect nodes 4 and 10 in x_3. Let Q be the set of e-paths though each program in the ECA management system: $Q = \{x_1, x_2, x_3, x_4, x_5, y_1, y_2, z_1\}$. The algorithm only considers e-paths in Q whose total number of effect nodes is equal to or less than the total number of effect nodes in x_3. Therefore, the following e-paths are considered: $\{x_1, x_3, x_4, y_1, z_1\}$. E-paths x_1, x_3, x_4 do not contain any error-correction node for correcting any EE resulting from ξ in x_3. Effect node 7 in y_1 is the error-correction node for correcting EC in effect node 10 in x_3. Effect node 4 in z_1 is the error-correction node for correcting EC in the effect node 4 in x_3. As such, two possible mappings that can be formed are $\theta_1(4) = 4$ in z_1 and $\theta_2(10) = 7$ in y_1. The set $\Omega_{p,\xi}$ is equal to $\{(z_1, \theta_1), (y_1, \theta_2)\}$ and the candidate set of correction paths for correcting the input error ξ in x_3 is $\{y_1, z_1\}$.

We consider the following subsets of $\Omega_{p,\xi}$: $K_1 = \{(z_1, \theta_1)\}$, $K_2 = \{(y_1, \theta_2)\}$ and $K_3 = \{(z_1, \theta_1), (y_1, \theta_2)\}$. Let $B_i = \{q|(q,\theta) \in K_i\}$, $i = 1, 2, 3$. As $B_1 = \{z_1\}$ and $B_2 = \{y_1\}$, they cannot correct all the EEs resulting from ξ; thus, they cannot be the BECS for correcting the input error. We now consider $B_3 = \{y_1, z_1\}$. From the two mappings θ_1 and θ_2, it is obvious that all EEs in x_3 resulting from ξ can be corrected by e-paths in B_3. Moreover, each EE is corrected by one and only one effect node in an e-path in B_3. From B_3, a partition of the effect nodes in x_3 can be formed as follows: $M = \{M_1, M_2\}$, where $M_1 = \{4\}$ and $M_2 = \{10\}$. M_1 contains all the effect nodes in x_3 which are corrected by effect nodes in z_1, and M_2 contains all the effect nodes in x_3 which are corrected by effect nodes in y_1. As M_1, M_2 contain only one element, M satisfies the second condition of Property 2; hence, the BECS of ξ is $B_3 = \{y_1, z_1\}$.

The column *Correction Feature* in Table CI (Appendix C) shows the basis error-correction sets recovered for each input error in executing each e-path in the ECA system.

Step 4 (*Recovery of program input error correctability*): In this step, we recover a set of programs whose input errors are correctable. The algorithm for this step is shown in Appendix B. In summary, the algorithm has two main phases. The first phase (Phase 1) applies Property 3 to recover a set of programs whose input errors are correctable. In this step, each program T in the system is checked to determine whether EC in all inputs accessed in the program is correctable. If so, T satisfies the property stated in Property 3, and therefore it is concluded that all after-effect input errors of T are correctable. If it is uncorrectable, then the checking for T is completed with no conclusion from Property 4.

The second phase (Phase 2) applies the second sub-property in Property 4 to infer the correctability for all other programs in the system. Let Φ be the set of programs in the system whose input errors have been concluded as correctable in the earlier sub-step. Let Ψ be the set of remaining programs in the system. Ψ is iterated over until the value of Φ is stabilized (that is, its value does not change from the previous iteration) by applying Property 4. Finally, the input errors of each program in Φ are correctable and the input errors of each program in Ψ are not correctable.

According to Table II, all the EC in executing the programs in Figures 1–3 are correctable. As such, Phase 1 of this step concludes that all the input errors in executing these programs are also correctable. Note that Phase 2 of this step can also be applied to conclude on the correctability of the programs in Figures 2 and 3. As all the input errors in executing the program in Figure 1 are correctable and the programs in Figure 2 and 3 are error-correction programs for the program in Figure 1, all the input errors in executing the programs in Figures 2 and 3 are also correctable.

5. MAINTAINING THE INPUT ERROR-CORRECTION FEATURES

5.1. Decomposition slicing

The previous section describes how the provisions for the IECFs in a system can be automatically recovered from the system's source code. As input error correction is an important functional feature, clearly, the provisions recovered are useful in testing, maintenance and comprehension of software systems in general and information system in particular. When a programmer writes a new or edits an existing correction feature, she/he hopes to make this change without breaking the system. How can one check that the new feature does not interfere with existing features? In this section, we show techniques that make use of the recovered information to assist the software engineer in maintaining these features.

Program slicing [9] is a well-known method that can be used to extract from a program those statements which are relevant to a particular computation. Program slicing was first introduced by Weiser [10] and has been used extensively in many software engineering fields, such as debugging, code understanding, testing, reverse engineering, software maintenance, reuse and metrics [11–16]. A slice $S(v,n)$ of program P on variable v at statement n yields the portions of the program that contribute to the value of v just before statement n is executed. (v,n) is called a slicing criterion.

The DeSlice technique is proposed by Gallagher and Lyle [13] to assist software engineers in modifying the components of a program without introducing ripple effects. The paper introduces the decomposition slice concept for the decomposition of a program with respect to a variable. A decomposition slice with respect to a variable v captures all the computation of v and is independent of program location. Two decomposition slices are independent if they have no statements in common. A slice is strongly dependent on another slice if it is a subset of the latter. A slice that is not strongly dependent on any other slice is maximal. A statement that is shared by more than one slice is dependent; otherwise, it is independent. A variable that is defined by an independent statement is independent. The complement of a decomposition slice is defined as the original program minus all the independent statements in the decomposition slice. The following set of guidelines was proposed to ensure that modifications to the decomposition slice should not affect the complement; thus, only testing of the modified slice is necessary:

Rule 1: Independent statements may be deleted from a decomposition slice.

Rule 2: Assignment statements that target independent variables can be added anywhere in a decomposition slice.

Rule 3: Logical expression and output statements may be added anywhere in a decomposition slice.

Rule 4: New control statements that surround any dependent statement will affect the complement's behavior.

5.2. Effect-oriented decomposition slicing

As we are interested in program statements that implement error-correction features, we adopt the idea of DeSlice to decompose a program with respect to the IECFs. By doing this, we obtain a smaller view of a correction program with the unrelated statements removed and dependent statements restricted from modification. We introduce the concept of effect-oriented decomposition slice. The idea is that an effect-oriented decomposition slice of a program, taken with respect to the external effects raised during the execution of the program, should contain only statements that affect the attributes of some external effects raised during the execution of the program.

Let U be a set of some effect nodes in a program. The statements in $S_d(U)=\bigcup_{n\in U} S(\mathit{Varset}(n),n)$, where *Varset*($n$) is a set of variables used by statement n and $S(\mathit{Varset}(n),n)$ is the slice taken at statement n with respect to *Varset*(n), form the *effect-oriented decomposition slice* of the program taken with respect to U. For example, if $U=\{8\}$ (Figure 1(a)), then $S_d(U)=\{1,2,5,6,7,8\}$ is the effect-oriented decomposition slice of the program in Figure 1 taken with respect to U. Clearly, $S_d(U)$ contains only statements that (transitively) affect the attributes `studentId` and `clubName` for executing effect node 8.

Let ξ be a type of input error in an e-path p. Let q be a path in a BECS of ξ. Let E_q be the set of the effect nodes in q.We define the effect-oriented decomposition slice taken with respect to E_q,

$S_d(E_q)$, as the *error-correction decomposition slice* (EcDS) taken with respect to q for correcting the input error ξ.

As an illustration, consider the *e*-path $x_5 = (\text{begin}, 1, 2, 3, 5, 6, 7, 8, 9, 7, 10, \text{end})$ in Figure 1. Let ξ be EC in <`clubName`>, which will lead to EC in executing 8 and VE in attributes `ECA_points` in executing 10. According to Table II, this error can be corrected by path $y_2 = (\text{begin}, 1, 2, 3, 4, 5, 6, 4, 7, \text{end})$ in Figure 2(b). Thus, $E_{y2} = \{5, 7\}$ and $S_d(E_{y2}) = \{1, 2, 3, 4, 5, 6, 7\}$ is the EcDS taken with respect to y_2 for correcting the input error ξ. Note that, in this simple example, all the statements in each program are involved in some IECFs; therefore, the effect-oriented decomposition slice is not much different from the program itself. However, during our empirical studies, we find that a program might implement more than one functional feature; thus, the size of the effect-oriented decomposition slice is much smaller than the program. As such, it is useful in isolating the IECFs for understanding.

Let $S_d(E_{q1})$ and $S_d(E_{q2})$ be two EcDSs taken with respect to *e*-paths q_1 and q_2 (q_1, q_2 are in the same program) for correcting some input errors. Adopting terms from [13], statements in $S_d(E_{q1}) \cap S_d(E_{q2})$ are called *slice-dependent statements* with respect to the two EcDSs. Dependent statements are shared among several effect-oriented decomposition slices and should not be modified during the maintenance of any particular feature, else it will affect the behaviors of other correction features. On the basis of the definition of slice-dependent statements, other statement and variable dependency concepts can be inherited from DeSclice [13] as presented in Section 5.1.

For instance, *e*-paths x_1 and x_3 (Table II) are the correction paths for EC in <`studentId`> in path z_1 and EC in <`studentId+{clubName}`> in y_1 accordingly. The EcDSs taken with respect to x_1 and x_3 are $S_d(E_{x1}) = \{1, 2, 3, 4\}$ and $S_d(E_{x3}) = \{1, 2, 6, 10\}$. As such slice-dependent statements are $\{1, 2\}$ in Figure 1(a).

5.3. Removing error-correction feature

During the maintenance phase, the engineer might decide that the IECF of an *e*-path is no longer needed or some BECSs in the correction feature need to be removed. Let F be a BECS of the input error ξ in an *e*-path. F might contain paths that belong to different programs in the system. To successfully remove F from the system, at least one path in the set must be successfully removed from the containing program. The following corollary states the conditions in which a BECS cannot be removed from a system. A maintainer should check for this condition before deciding to remove a correction set from the system.

Corollary 1. *Let F be a BECS of some input error ξ. If for each path p in F the EcDS taken with respect to p for correcting ξ is strongly dependent on the EcDS taken with respect to an e-path q for correcting some other input error, where p and q are in the same program, then F cannot be removed from the system.*

This is a direct corollary of Rule 1. Let $S_d(E_p)$ and $S_d(E_q)$ be the EcDSs taken with respect to p and q, respectively. If $S_d(E_p)$ is strongly dependent on $S_d(E_q)$, where p and q are in the same program, then no statements can be removed from p because it is essential for the correction path q for correcting some other input error. As such, p cannot be removed from the containing program. If for each such path p in F p cannot be removed from the containing program, then the BECS F cannot be removed.

Table II shows that $\{y_1\}$ is the BECS of EC in S_1 in the e-path x_4 in Figure 1. The EcDS taken with respect to y_1 for correcting the input error is $S_d(E_{y1}) = \{1, 2, 3, 7\}$. In the program containing y_1 (Figure 2), there is also another e-path y_2, which is also in some other BECSs of some input errors as given in Table II. The EcDS taken with respect to y_2 for correcting these input errors is $S_d(E_{y2}) = \{1, 2, 3, 4, 5, 6, 7\}$. Clearly, $S_d(E_{y1})$ is strongly dependent on $S_d(E_{y2})$; thus, we cannot remove y_1 from the program in Figure 2 as well as the BECS $\{y_1\}$.

Corollary 2. *Let F be a BECS of some input error. If there exists a path p in F such that the EcDS taken with respect to p for correcting the input error is maximal, then F can be completely removed from the system.*

This is another direct corollary of Rule 1. If the EcDS taken with respect to a correction path is maximal, then there are some independent statements in the slice which can be removed. It is obvious that if at least one statement is removed from a decomposition slice, then the path is considered removed from the respective program; thus, F is considered removed.

To remove a BECS F of an input error we first check whether F is removable by verifying against Corollary 1 or 2. If so, for each path p in F, we compute the EcDS taken with respect to p. If the EcDS is a maximal decomposition slice, we compute the set of independent statements in the decomposition slice. To remove p, we can remove all the independent statements. Note that when deleting a statement, all the statements that are transitively control dependent on the statement must also be deleted. As long as at least one path in the BECS can be removed, the set is considered removed.

For example, assume that table CLUB (Figure 1(a)) is non-updatable. As such, the external effect 'insert into table CLUB' (line 8, Figure 1(a)) is not reversible, i.e., once a record is inserted into the database, it is not possible to remove it. Subsequently, the software engineer wishes to remove the error-correction features of e-paths x_2 and x_5 (Table CI, Appendix C) by removing path y_2 in the program in Figure 2 from all the BECSs containing the path. The EcDS taken with respect to y_2 is $S_d(E_{y2}) = \{1, 2, 3, 4, 5, 6, 7\}$. According to Table CI, e-path y_1 in the program in Figure 2 also belongs to some BECSs. The EcDS taken with respect to y_1 is $S_d(E_{y1}) = \{1, 2, 3, 7\}$. Since y_1 and y_2 are all the e-paths through the program in Figure 2, $S_d(E_{y2})$ is a maximal decomposition slice. The set of independent statements of $S_d(E_{y2})$ is $S_d(E_{y1}) \cap S_d(E_{y2}) = \{4, 5, 6, 7\}$. To remove $\{y_2\}$, we need to remove statements 4–7 from the program in Figure 2.

5.4. Adding error-correction feature

Basically, the problem of adding a new error-correction feature to a system is equivalent to adding a code segment, which implements the new correction feature into an existing decomposition slice. This problem has been well addressed by Gallagher and Lyle [13]. A detailed example is also given in the paper. The basic idea is that the new code segment must follow modification Rules 2 and 4. To satisfy Rule 2, for each variable used in the new code segment, which is dependent on the existing slice, we create a new local variable, whose name does not conflict with any variable in the existing slice. We copy the value of the dependent variable to the new local variable and replace all the uses of the former with the latter in the new code segment. To satisfy Rule 4, the maintainer must ensure that a new control statement does not surround any dependent statements when specifying the position for it in the decomposition slice. A code segment satisfying Rules 2 and 4 can be automatically merged with the old code.

6. PROTOTYPE SYSTEM

A prototype system called RaMIC (recovery and maintenance of IECFs) has been developed for PHP and Java systems following the proposed approach presented in Sections 4 and 5. The architectural diagram is shown in Figure 4. It consists of three components: PA, IECFs recovery and DeSlice. In our prototype implementation, the Eclipse Software Development Kit is used as the environment to recover IECFs, compute effect-oriented decomposition slices and perform automated removing and adding of IECFs.

PA implements Step 1 of the approach for automated recovery of the IECFs proposed in Section 5. To analyze the source code of each program in a system and construct its CFG, PA consists of a PHP parser [17], SOOT [18], a simplified HTML parser [19] and an SQL query analyzer.

PHP parser analyzes programs written in PHP and generates a CFG for each program in a system. Each node in the CFG is attached with control and data-dependency information. SOOT is incorporated to support systems written in Java. SOOT is an advanced program analysis tool for

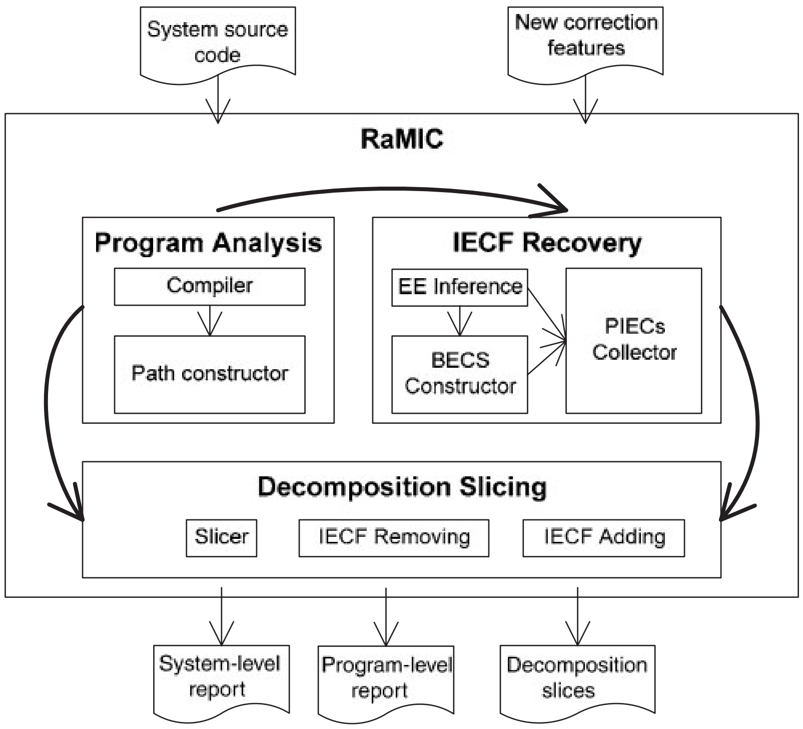


Figure 4. The prototype system.

Java programs that produces a CFG for each program in a system and gathers various control and data-dependency information.

A PHP page might contain mixtures of markup languages (HTML) and code written in PHP. The HTML parser crawls through the source code to identify code segments that contain HTML code and extract information on input variables and their values from input forms. Code segments that might contain HTML code include HTML blocks, statements that assign or define values for strings, output statements such as print and echo statements, and some other functions that produce HTML code.

The SQL query analyzer is an advanced SQL parser, which provides information about an SQL statement, such as the statement type (i.e., insert, update, delete, etc.), and about what tables and attributes are used in the statement. For systems such as web applications, some SQL statements are dynamically generated at run time. As such, the external effect raised by an SQL statement varies from execution to execution of the web application. Hence, an original SQL parser could not solve this problem. This problem, however, cannot be solved easily with dynamic analysis. Dynamic analysis is based on executing the application on particular inputs, monitoring the execution trace and analyzing the dynamic information. Normally, the code is instrumented to gather information during execution. Although dynamic analysis can gather run-time information, result from several executions cannot be generalized to all executions. Moreover, dynamic analysis relies very much on the design of inputs to force an execution trace. In [20], we have proposed an approach using symbolic execution and heuristic rules to identify all the external effects in a system automatically. Each e-path is symbolically executed by applying a set of *symbolic evaluation rules* to derive a set of symbolic expressions for the possible types of external effects raised by the path. The symbolic evaluation rules speed up and simplify the symbolic evaluation process. The types of external effects are then inferred from the symbolic expressions by applying the *Inference rules* proposed in [20]. Basically, the approach is still based on static analysis. However, the approach is facilitated by heuristic rules that make its performance much better than other approaches using symbolic evaluation. The algorithm for identifying external effects in a system has been implemented for the SQL analyzer.

IECF recovery consists of three modules: EE inference, BECS constructor and programs with input errors correctable collector, each of which implements Steps 2–4 in the proposed recovery approach in Section 4, respectively. This component produces two levels of reports:

- A system-level report lists all the programs in a system and gives overview information on the IECFs implemented in each program.
- A program-level report lists all the possible input errors in a program. For each input error, it shows all the resulting EEs as well as the basis error-correction sets of the input error. Table CI in Appendix C is a simplified program-level report for the ECA system.

The core of DeSlice is a slicer for computing the various decomposition slices for the IECFs. Slicing of Java programs was performed through the use of Kaveri [21], an Eclipse plug-in front-end for the Indus Java slicer. It utilizes the Indus Java Slicer to calculate slices of Java programs and then display the result visually in the Eclipse Java editor. We also extended Kaveri to support slicing of PHP programs. In addition, we implemented two small tools, namely *IECF Removing* and *IECF Adding*, following the approach presented in Sections 5.3 and 5.4, respectively. *IECF Removing* automatically removes a basis error-correction set specified by users from the code. It notifies the user in the case where the set cannot be removed. Currently, *IECF Adding* inputs a text file that contains the code of the new feature and specifies the program to which it will be added. The tool automatically merges this code segment with the program following the algorithm proposed by Gallagher and Lyle [13].

7. VALIDATION AND EVALUATION

In this section, we report the result of binominal testing conducted to statistically validate the empirical properties presented in Section 3. We also present a case study to evaluate the usefulness of the proposed approach.

7.1. Validation of empirical properties

We have conducted binomial tests [22] to validate each of the proposed empirical properties. For each empirical properties Q, the alternate hypothesis states that Q holds for equal to or more than 99% of the cases. Hence, the null hypothesis states that Q holds for less than 99% of the cases. That is,

$$H_0 \text{ (null hypothesis):} \quad p(Q \text{ holds}) < 0.99$$

$$H_1 \text{ (alternate hypothesis):} \quad p(Q \text{ holds}) \geq 0.99$$

The binomial test statistics z is computed as follows:

$$z = \frac{X/n - p}{\sqrt{(p(1-p)/n)}}$$

where n is the total sample size for the test and X is the number of cases that support the alternative hypothesis H_1. Taking 0.005 as the type I error, if $z > 2.408$, we reject the null hypothesis; otherwise, we accept the hypothesis.

The sample for the validation was collected from the following 10 systems. Table III gives a brief description of each system.

All the open-source systems can be downloaded from [23] by searching for the systems' names. On the basis of development status, all the above projects are fully tested. Therefore, to evaluate the generality of our approach, we collected additional samples from four projects that are under other stages of the software development life cycle, including 'under-development', 'unit-testing' and 'system testing'. NCS is an industrial project developed by the National Computer of Singapore. For confidential reason, we do not disclose the project name. We purposely chose a system written in Java to show that our approach is independent of any programming language.

As Property 3 is an invariant property, we use the prototype system for the validation of Properties 1, 2 and 4. We also develop a tool called EP-Validator to provide some additional functions to assist the validation of the empirical properties. As EP-Validator is used only for the validation purpose, it is not a part of RaMIC; hence, it is not shown in Figure 4. Basically, the EP-Validator assists in generating test cases to execute programs in the samples and instruments the source code to monitor the execution and validate the outputs of each program. EP-Validator assists the test

Table III. Description of systems for hypothesis testing.

System	KLOC	Source	Status	Description
System written in PHP				
SchoolMate	11	Open source	Tested	School management system
Open Bib	58			Automated library system
WebCalendar	147			Schedule management application
CMapp	48			Contract management system
NetOffice	42			Online project management environment
Vacation-Sched	19			Online vacation scheduling system
System written in Java				
Smart_project	12	Student	Under development	Online project management system
Smart_exam	21		Unit testing	Exam assistant management system
GradeBook	8		System testing	Automatic grading system
NCS	137	Industrial	Unit testing	Health-care system

data-generation process for an e-path by first suggesting some input values taken from a data pool. The data pool is constructed as follows. EP-Validator instruments the input nodes to collect the name of the input variables, their data types and the input values submitted. Prior to the experiment, for each system, we set up and run the system. We try to interact with the system as much as possible. During this initial interaction with the system, input values submitted are stored in the data pool. To generate input values for an e-path, EP-Validator first identifies the set of input variables that affect the execution of the e-path. For each input variable, EP-Validator searches in the data pool for data items with the compatible data types. If there exist such data items, the one with the closest variable name will be recommended. If the e-path is not executed with the recommended values, we have to manually provide inputs.

The validations and their results are summarized as follows:

1. Property 1: Each *e*-path through the CFG of each program forms a case for testing Property 1. For each *e*-path, based on Property 1, RaMIC infers all possible EEs resulting from each *e*-path. The results obtained from RaMIC are then double checked with the assistance of EP-Validator. Basically, for each input error, we manually design a test case to intentionally create such an input error. EP-Validator instruments all the associated effect nodes so that we can monitor the external effects raised due to the input error created. If the results obtained from RaMIC are confirmatory for all the input errors in the *e*-path, we conclude that Property 1 holds for the case. As shown in Table IV, the total number of cases for testing this Property is 576. As the property holds for all the cases in the sample, the z-score is 2.412 ($X=576, n=576, p=0.99$).
2. Property 2: Each input error in each *e*-path through the CFG of a program in the sample forms a case for testing Property 3. For each input error, based on Property 2, RaMIC computes the basis error-correction set of the input error. For each set, we design a test case to enforce the occurrence of the input error. We then execute the paths in the basis error-correction set and check whether the input error is indeed corrected. If the result is confirmatory, then we conclude that Property 3 holds for the type of input errors. As shown in Table IV, the total

Table IV. Sample size for testing empirical properties.

Systems	Property 1	Property 2	Property 4
SchooMate	45	147	56
OpenBiblio	52	168	54
WebCalendar	47	144	60
NetOffice	78	243	198
Cmapp	54	168	90
Vacation-shed	91	279	78
Smart_project	74	234	12
Smart_exam	46	147	55
GradeBook	89	288	54
NCS	134	435	378
Total	576	1818	657

number of cases for testing this property is 1818. As Property 2 holds for all the cases in the sample, the z-score is 4.285 ($X = 1818, n = 1818, p = 0.99$).

3. Property 4: Each program in the sample to which some inputs are accessed forms a case for testing of Property 4. RaMIC infers the correctability for the after-effect input errors of each program based on Property 4. For cases that are inferred from the first condition of Property 4, no additional validation is needed since all the basis error-correction sets have been confirmed correct through the testing of Property 2. For cases that are inferred from the second condition, we manually design test cases with the assistance of EP-Validator and execute the program to check the result obtained by RaMIC. If the results are confirmatory, then we conclude that Property 4 holds for the program. As shown in Table I, the total number of cases for testing Property 4 is 657. As all the cases gave affirmative result to the property, the z-score is 2.576 ($X = 657, n = 657, p = 0.99$).

Clearly, the z-scores for testing all the above-mentioned properties are greater than 2.408. Thus, we reject all the null hypotheses and conclude that all the empirical properties hold for more than or equal to 99% of the cases at 0.5% level of significance.

7.2. A case study

At the heart of our approach is the recovery of IECFs to assist program understanding tasks and maintenance of the features. Therefore, we conduct a case study to assess the usefulness of our approach by mimicking a program-understanding task assuming no previous knowledge of the programs to identify the features.

We choose ZebraZ and NTUMatch as the subject systems for our case study. Both systems are written in PHP. ZebraZ is a membership management system for clubs and organizations which can be downloaded from [23]. The latest version of ZebraZ contains 13 688 LOC decomposing in 206 programs (files). NTUMatch is a match-making system designed to help students in a university to find their ideal partners. This is a student project with 25 452 LOC contained in 374 programs. The project is developed by a group of final-year undergraduate students. No integration testing has been performed on either system. The incompleteness of the IECFs makes them interesting for our case study.

We formulate the following program understanding task: 'identify possible input errors that occur during the execution of the system that are uncorrectable and locate the code segments that are affected by the input errors'. We tackle the programming task manually with the participation of two groups of programmers and systematically with the proposed approach using RaMIC tool. We then compare and discuss the results obtained from the two groups with the results produced by RaMIC. The first group consists of two final-year students and the second group consists of two senior programmers with two years of experience. The two groups are free to choose their own approaches to this program-understanding task. We provided them with the program analysis tool, which is a part of the RaMIC tool. This program analysis tool converts each program into a CFG, which can be displayed using Dotty [24], which assists the participants in visualizing the control flow of each program. We also provided them with the DeSlice component of the RaMIC tool to aid the participants in isolating particular code segments of their interests. To ensure that the case study environment does not favor the proposed approach, we gave the participants a tutorial on IECFs as well as the full documentation of the two systems.

Basically, there is no documentation associated with ZebraZ. Therefore, we had to thoroughly study the system and manually create the documentation for ZebraZ. We also studied the source code to manually work out the set of uncorrectable input errors and the associated code segments, which are considered as the *expected outputs* of the program-understanding task. The precision of the results obtained by each participant (including RaMIC) will be assessed by comparing with the expected outputs.

For each input error concluded uncorrectable by each group, we check whether the input error is included in the expected output. If so, we call this a true-positive case; otherwise, it is a false-positive case. Those input errors that are in the expected output (i.e., they are uncorrectable), but are, undiscovered by the participants are referred to as true-negative cases. The same checking procedure is also applied to results obtained by RaMIC.

Table V presents the results of the case study. Columns *t*, *uncorrectable*, *falsePositive* and *trueNegative* give the time spent by each group to conduct the task for both ZebraZ and NTUMatch, the total number of uncorrectable input errors produced by each group, the number of false-positive cases and the number of true-positive cases, respectively.

We compute the precision and recall of the results obtained by each approach by adopting the definition of precision and recall from Information Retrieval field as follows. The precision of the information produced by each group is measured by the percentage of the input errors that are concluded uncorrectable by the group and are also true positive. The precision is computed using

Table V. Case study results.

Participant	*t*	Uncorrectable	TrueNegative	FalsePositive	Pre (%)	Recall (%)	*f*
Student	22 h	81	70	12	85.2	49.6	0.63
Programmer	13 h	114	30	5	95.6	78.4	0.86
RaMIC	9921 ms	139	0	0	100	0	1

the following formula:

$$Precision = \frac{uncorrectable - falsePositive}{uncorrectable} \times 100\%$$

The recall (completeness) of the information produced by each group measures the ratio between the number of input errors that are concluded uncorrectable by the participant and are also true positive and the actual number of all the uncorrectable input errors in the system:

$$Recall = \frac{uncorrectable - falsePositive}{uncorrectable - falsePositive + trueNegative} \times 100\%$$

We evaluated the overall quality of information produced by each group based on the traditional f-measure as follows:

$$f\_measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

F-measure weights low values of precision and recall more heavily than higher values. It is high if both precision and recall are high.

Columns *Pre*, *Recall* and f in Table V give the precision, recall and f-measure of the result obtained by each group and RaMIC, respectively. It is not surprising that the programmers spent only 13 h, 9 h less than the student, and recovered 114 uncorrectable input errors, out of which there were five false-positive cases. As a result, the information produced by the programmers achieved 95.6% precision with 78.4% recall. Overall, the f-measure of the information produced by the programmers was 0.86. On the other hand, the information produced by the students gave only a precision of 85.2% with 49.6% recall, and overall the f-measure was 0.63.

The results show that even programmers with two years of experience need to spend much time on understanding the systems and implementing the IECFs, yet accurate information cannot be manually recovered from the source code. However, in this case, RaMIC produced 139 uncorrectable errors with no false-positive or true-negative cases. Thus, the overall f-measure of the information recovered by RaMIC is 1. More importantly, all this information is recovered automatically and the time spent by RaMIC was approximately 9921 (ms) for two systems.

7.3. Threats to validity

This study, like any other empirical study, has some limitations. In this section, we identify the primary threats to validity and explain how we tried to control unwanted sources of variation.

First, the binomial tests presented in Section 7.1 are influenced by our choice of sample sets and the subject applications. Even though we have randomly chosen systems from various application domains and different stages of the software development life cycle, the chosen systems might not be representative of the population. Moreover, the development status of each system is also not reliable and might follow different standards. However, we surmise that our conclusions will generalize since they are based on results that are consistent over a population of sample sets that are greatly varied in size.

Second, the hypothesis testing (Section 7.1) was conducted on a large sample size in which the results produced by the prototyping tool were manually validated and tested with the aid of the EP-Validator to check whether a particular property held for a case. This is a tedious and error-prone task. We have spent more than one year on the validation, with one author working in conjunction with a senior programmer to countercheck any conclusion.

Last, the validity of the case study presented in Section 7.2 is based on a subjective assessment of the IECFs recovered by a group of students and a group of senior programmers who have no prior knowledge of the two subject systems. To avoid favoring the proposed approach, we provided the participants with available program analysis tools with visualization of CFG and a PHP slicer. We also provided each participant full documentation of the systems and essential background on the IECFs.

8. RELATED WORK

Program slicing is a method for restricting a program to a specified subset of interest, which has been used widely in numerous fields in software engineering, including debugging, program understanding, testing, reverse engineering and software maintenance [12,14–16,25,26].

Lanubile and Visaggio [27] recover reusable functions from program source code using transform-slicing techniques. The approach requires the user to identify a criterion that involves a set of input variables, a set of output variables and an initial statement. Thus, this technique involves difficulties in identifying the statements to slice upon to recover the functionality from the program source code.

Another approach to extract functionalities from the source code based on static slicing techniques augmented for handling input/output statements is proposed by Tan and Kow [28]. This approach is developed based on the hypothesis that a program delivers its functionalities through its output statements. As such, it is not useful for programs that do not satisfy the hypothesis, such as non-database or non-data-intensive programs. Moreover, it is limited by the existing techniques that deal with input/output statements in slicing. Thus, in some cases, the approach may yield a slice that is larger than what is required.

Software maintainers are faced with the problem of understanding the existing software and making changes without having negative impacts on the unchanged parts. Gallagher and Lyle [13] propose using DeSlice techniques to aid in making changes to a piece of software without unwanted side effect. A decomposition slice captures the computation performed by a program as a whole on a variable of interest. We adopt this idea to decompose a program in terms of the IECFs implemented; thus it is called effect-oriented DeSlice. While the decomposition slice [13] captures all the computations of a variable regardless of the program location, an effect-oriented decomposition slice captures all the statements involved in the implementation of an IECF.

Surgeon's Assistant [29] is a CASE tool that also uses DeSlice to analyze and limit the scope of changes to the program. The tool also incorporates a decomposition slice system that uses an acyclic graph to display slices as nodes to the maintainers. This graph is also referred to as a decomposition slice graph, which shows the computation inclusion relationship between different variables in a program. Ricca and Tonella [30] mention the use of DeSlice in understanding web applications, especially its organization, main computations and sub-computations. The use of a decomposition slice graph helps in evaluating the impact of a modification on web pages in the web application.

However, researchers [31,32] soon realized that the ‘interference’ dependency between two variables is missing in the decomposition slice graph. Tonella [32] proposes a new program representation called a concept lattice of decomposition slices, which is an extension of the decomposition slice graph, with additional nodes associated with weak interference between computations. Basically, a decomposition slice graph is not a lattice, i.e., infimum and supremum are not always unique; therefore, the decomposition graph cannot always capture all the relationships between computations. On the basis of this idea, Tonella forms a concept analysis problem in which there exists a binary relationship between a variable and a statement if the statement belongs to the decomposition slice of the variable. As such, the resulting concept ‘contains a set of variables that share some common computations’. This new program representation is more useful compared with DeSlice in terms of software maintenance, especially impact change analysis, as all the dependencies between computations are explicitly represented. Consequently, the problem of impact change analysis is equivalent to identifying a set of reachable nodes in the concept lattice.

Program analysis has been used to solve many problems (e.g., [28,33–35]). However, to the best of our knowledge, we have not seen any approach that uses any of these techniques for the automated recovery of IECFs. Moreover, our proposed approach is very different from other approaches that use program analysis. Our approach is a combination of empirical experiences obtained through the study of many software systems and formal statistical validation. Our empirical properties are constructed and refined through statistical validation and the iterative process of exploring and adjusting these properties in response to our empirical findings. The use of empirical properties enables automated recovery and makes it optimally precise and efficient.

A semi-automatic approach to locating features in the source code, which combines static and dynamic analysis and uses concept analysis, is presented by Eisenbarth *et al.* [33]. The method is capable of identifying the code segment, which is called a computational unit, which specifically implements a feature, as well as a set of jointly or distinctly required computational units for a set of features. The limitation of this approach is that it relies on domain-specific expertise, which is not always available.

Wilde *et al.* [36] and Wilde and Scully [37] develop the Software Reconnaissance method, which utilizes dynamic information to locate features in existing systems. However, the approach does not guarantee covering all the possible features or always discovering all code segments associated with a functional feature.

Antoniol and Gueheneuc [38] adopt the idea from epidemiology and propose an approach to accurately identify features in large multithreaded object-oriented programs. The approach is based on the statistical analysis of static and dynamic data. Firstly, dynamic data (traces) are collected from different scenarios during program emulation. Events in the collected traces are then ranked according to their relevance to a feature of interest through probabilistic ranking combined with epidemiological metaphor. The basic idea is that if an event (disease) occurs more frequently under certain scenarios, it is likely that the event is related to the feature. Finally, these ‘feature-relevant events’ are used to locate the source code constructs that implement the feature. The accuracy of the approach is very much dependent on the collection of dynamic data.

Chen and Rajlich [39] propose a semi-automated approach to feature location. An abstract system dependence graph (ASDG) is used as the high-level representation for a program that describes the dependencies among functions, classes and global variables. To locate a feature, the maintainer manually navigates the ASDG with the aid of a computer. The authors believe that, when dealing with feature location, a machine and a programmer are better than a machine alone since the task requires extensive knowledge that is hard to formalize. However, the scalability of the approach for large-scale systems might be problematic since the approach requires much manual effort.

Many other approaches [34,40–42] use graph-matching techniques to locate the functional features in source code by matching between predefined patterns or abstract high-level representations of the features with the structure of the source code itself. These methods require constructions of the patterns or the models representing the functional features.

The recovery framework proposed in this paper is an enhancement and extension of our initial work in [5,8]. This approach approximately recovers the provisions of IECFs from source codes. There is no precise inference on the types of EEs. In contrast to the approach proposed in [8], the approach proposed in this paper recovers accurate information, which is useful not only in the maintenance of the input error-correction features but also in the design of test cases.

9. CONCLUSIONS

We have proposed an approach for the automated recovery of after-effect IECFs from the source code of information systems and a method to aid the maintenance of these features using DeSlice.

Although the proposed approach used the basic technique of program analysis for the automated recovery of IECFs, there is a major theoretical difference between the proposed approach and other approaches that use program analysis for the automated functional features recovery. The proposed approach is developed through the integration of invariant and empirical properties. All the empirical properties have been validated statistically with samples collected from a wide range of information systems. The validation gives evidence that all the empirical properties hold for more than 99% of all the cases at 0.5 level of significance. In fact, we have not found any case in which any of our empirical properties fails.

On the basis of empirical properties, our approach automatically recovers the implementation of the IECFs from the source code in four steps. Through the use of program slicing, the recovered information aids maintainers in the maintenance of these features. In this paper, we have introduced the concept of an effect-oriented DeSlice, which is adopted from the DeSlice technique [13], to decompose a program with respect to the IECFs. As such, removing or adding a correction feature from and to a program can be done easily without introducing ripple effects to other features.

Our case studies have shown that, without using RaMIC, even a programmer with two years of experience requires to spend much time and effort on understanding the system, yet accurate information cannot be fully recovered.

The novelty of our approach lies in the use of empirical properties, which provides a simple yet efficient approach to recover the IECFs from the source code. We believe that this is a promising future direction, which opens a new avenue for the automated recovery and verification of functional features in software systems. For this purpose, more explorations on the automated recovery and maintenance of other functional features will be done in the future. Currently, we are also working on building a language-independent CFG. For each language, a parser is used to parse a system written in the language to the form of a language-independent CFG. Slicing and other algorithms will be implemented based on the new type of CFG. We are also extending the DeSlice features so that it can assist maintainers in a more interactive manner rather than using a text file.

APPENDIX A

Algorithm **recover_BECS**
Input: input error ξ, e-path p
Output: a set of all basis collections of error correction paths for correcting ξ in executing p

```
1   $Q = \{e$-path $q$ through each program each program in the system |
        the number of effect nodes in $q$ is equal to or
        less than the number of effect nodes in $p\}$
2   for each r-path $q \in Q$
        $\Psi_{eff} = \{$effect node in $q\}$, $\Im_{\text{potential}} = \emptyset$, $\Im_{\text{correction}} = \emptyset$
4       for each effect node $x \in \Psi_{eff}$
5           $\mathrm{E}_{\mathrm{q},x} = \{$effect node $e$ in $p|x$ is the correction node for correcting the effect node
            resulting from $\xi$ in executing $e$ through $p\}$
6           if ($\mathrm{E}_{\mathrm{q},x}$ is empty) then
7               continue the for loop in line 2, proceed to the next path $q$ in $Q$
8           else $\Im_{\text{potential}} = \Im_{\text{potential}} \cup \{E_{q,x}\}$
        endfor
9       compute all the possible one-to-one mappings $\theta$ from a subset of effect
        nodes in $p$ to the set of effect nodes in $q$ by assigning an element in
        $E_{q,x} \in \Im_{\text{potential}}$ to $x$ for each effect node $x \in \Psi_{eff}$
10      for each mapping $\theta: \Omega_{p,\xi} = \Omega_{p,\xi} \cup \{(q,\theta)\}$
    endfor
13  for each subset $K$ of $\Omega_{p,\xi}$
14      $isCorrectAll = false$, $isBasisCollection = $ false
15      if $\left(\bigcup_{(q,\theta)\in K} (\text{the domain of } \theta) = \text{ effect nodes in p}\right)$ then $isCorrectAll = $ true
16      if (for any $(q_1,\theta_1),(q_2,\theta_2) \in K$, (the domain of $\theta_1) \cap$ (the domain of $\theta_2) = \emptyset$)
17          then $isBasisCollection = $ true
18      if ($isCorrectAll$ and $isBasisCollection$) then
19          $\mathrm{B} = \{\mathrm{q_k} | (q_k,\theta) \in \mathrm{K}\}$ and $\mathrm{M} = \{\mathrm{M_k} | \mathrm{M_k} = \bigcup_{(qk,\theta)\in K} (\text{domain of } \theta), \mathrm{q_k} \in \mathrm{B}\}$
20          if M satisfies second condition of Property 2 then $\Im_{\text{correction}} = \Im_{\text{correction}} \cup \{\mathrm{B}\}$
        endif
    endfor
19  return $\Im_{\text{correction}}$
```

APPENDIX B

Algorithm **recover_correctable_programs**

Input: all programs in the system
Output: a set of programs whose input errors are correctable (Φ)
// Phase 1

```
1   for each program $P$ in the system
2       for EC in the input accessed in each $e$-path $p$ in $P$
3           $\Im_{\text{correction}} = \text{recover\_basis\_collection}(\text{EC}, p)$
6           if ($\Im_{\text{correction}}$ is empty) then
7               $correctability(P) = \text{UNDEF}$,
8               goto line 1 to process the next program in the system
            endfor
        endfor
9       $correctability(P) = \text{TRUE}$
10  endfor
// Phase 2
11  $\Phi = \{\text{program } P \mid correctability(P) = \text{TRUE}\}$, $\Psi = \{\text{all programs}\} \mid \Phi$
12  while ($\Phi$ is not stabilized)
13      for each $S \in \Psi$
14          for each $e$-path $q$ in $S$ which contains some effect nodes
15              for each $T \in \Phi$
16                  if NOT ($q$ is in some correction features of some $e$-paths in $T$)
17                  then goto line 13 and process the next program in $\Psi$
                endfor
            endfor
18          $\Phi = \Phi \cup \{S\}$
        endfor
    endwhile
19  return $\Phi$
```

APPENDIX C

The input error-correction features for ECA management are given in Table CI.

Table CI. Input error-correction features for ECA management system.

Program	E-path	Input error	Effect error	Correction feature
Figure 1	x_1	EC in S_1	EC in node 4	$\{z_1\}$
		EO in S_1	EO in node 4	$\{x_1\}$
		VE in S_1	VE in node 4	$\{z_1, x_1\}$
	x_2	EC in S_1	EC in nodes 4, 8, 10	$\{y_2, z_1\}$
		EO in S_1	EO in nodes 4, 8, 10	$\{x_2\}$
		VE in S_1	VE in nodes 4, 8, 10	$\{y_2, z_1, x_2\}$
		EC in K_1	EC in node 8, VE in node 10	$\{y_2\}$
		EO in K_1	EO in node 8, VE in node 10	$\{x_5\}$
		VE in K_1	VE in nodes 8, 10	$\{y_2, x_5\}$
	x_3	EC in S_1	EC in nodes 4, 10	$\{y_1, z_1\}$
		EO in S_1	EO in nodes 4, 10	$\{x_3\}$
		VE in S_1	VE in nodes 4, 10	$\{y_1, z_1, x_3\}$

Table CI. *Continued*.

Program	*E*-path	Input error	Effect error	Correction feature
	x_4	EC in S_1	EC in node 10	$\{x_4\}$
		EO in S_1	EO in node 10	$\{x_4\}$
		VE in S_1	VE in node 10	$\{x_4\}$
	x_5	EC in S_1	EC in nodes 8, 10	$\{y_2\}$
		EO in S_1	EO in nodes 8, 10	$\{x_5\}$
		VE in S_1	VE in nodes 8, 10	$\{y_2, x_5\}$
		EC in K_1	EC in node 8, VE in node 10	$\{y_2\}$
		EO in K_1	EO in node 8, VE in node 10	$\{x_5\}$
		VE in K_1	VE in nodes 8, 10	$\{y_2, x_5\}$
Figure 2	y_1	EC in S_2	EC in node 7	$\{y_1\}$ or $\{x_4\}$
		EO in S_2	EO in node 7	$\{y_1\}$ or $\{x_4\}$
		VE in S_2	VE in node 7	$\{y_1\}$ or $\{x_4\}$
	y_2	EC in S_2	EC in nodes 5, 7	$\{x_5\}$
		EO in S_2	EO in node 5, 7	$\{y_2\}$
		VE in S_2	VE in nodes 5, 7	$\{x_5, y_2\}$
		EC in K_2	EC in node 5, VE in node 7	$\{x_5\}$
		EO in K_2	EO in node 5, VE in node 7	$\{y_2\}$
		VE in K_2	VE in nodes 5, 7	$\{x_5, y_2\}$
Figure 3	z_1	EC in S_3	EC in node 4	$\{x_1\}$
		EO in S_3	EO in node 4	$\{z_1\}$
		VE in S_3	VE in node 4	$\{x_1, z_1\}$

S_1 = `<studentId+studentName+{clubName}>`; K_1 = `<clubName>`
S_2 = `<studentId+{clubName}>`; K_2 = `<clubName>`
S_3 = `<studentId>`

REFERENCES

1. Beizer B. *Software Testing Techniques* (2nd edn). Van Nostrand Reinhold: New York NY, 1990.
2. Turner CR, Fuggetta A, Lavazza L, Wolf AL. A conceptual basis for feature engineering. *Journal of Systems and Software* 1999; **49**:3–15.
3. Ruthruff JR, Elbaum AS, Rothermel AG. Experimental program analysis: A new program analysis paradigm. Presented at *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, 2006.
4. Basili VR. The role of experimentation in software engineering: Past, current, and future. *Proceedings IEEE 18th International Conference on Software Engineering*, 25–30 March 1996.
5. Ngo MN, Tan HBK. A method to aid recovery and maintenance of the input error correction features. *22nd IEEE International Conference on Software Maintenance*, 24–27 September 2006.
6. Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 1987; **9**:319–349.
7. Fasolino AR, Natale D, Poli A, Quaranta AA. Metrics in the development and maintenance of software: An application in a large scale environment. *Journal of Software Maintenance* 2000; **12**:343–355.
8. Tan HBK, Thein NL. Recovery of PTUIE handling from source codes through recognizing its probable properties. *IEEE Transactions on Knowledge and Data Engineering* 2004; **16**:1217–1231.
9. Tip F. A survey of program slicing techniques. *Journal of Programming Languages* 1995; **3**:121–189.
10. Weiser M. Program slicing. *Fifth International Conference on Software Engineering*, 9–12 March 1981.
11. Beck J, Eichmann D. Program and interface slicing for reverse engineering. *Proceedings of the 15th International Conference on Software Engineering*, 17–21 May 1993.

12. Binkley D, Harman M. Results from a large-scale study of performance optimization techniques for source code analyses based on graph reachability algorithms. *Third IEEE International Workshop on Source Code Analysis and Manipulation. Proceedings*, 26–27 September 2003.
13. Gallagher KB, Lyle JR. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering* 1991; **17**:751–761.
14. Kamkar M, Fritzson P, Shahmehri N. Interprocedural dynamic slicing applied to interprocedural data flow testing. *Proceedings of the Conference on Software Maintenance*, 27–30 September 1993.
15. Krinke J. Advanced slicing of sequential and concurrent programs. *Proceedings—20th IEEE International Conference on Software Maintenance*, *ICSM 2004*, 11–14 September 2004.
16. Laski J, Szermer W. Identification of program modifications and its applications in software maintenance. *Conference on Software Maintenance 1992* (*Cat. No. 92CH3206-0*), 9–12 November 1992.
17. PHPEclipse. http://sourceforge.net/projects/phpeclipse/ [25 September 2007].
18. SOOT. A Java bytecode optimization framework. http://www.sable.mcgill.ca/soot/ [12 August 2007].
19. HTMLParser. A HTML Parser. http://htmlparser.sourceforge.net/ [12 August 2007].
20. Ngo MN, Tan HBK. Applying static analysis for automated extraction of database interactions in web applications. *Information and Software Technology* 2007; DOI: 10.1016/j.infsof.2006.11.005.
21. Jayaraman G, Ranganath VP, Hatcliff J. Kaveri: Delivering the Indus Java program slicer to Eclipse. *Fundamental Approaches to Software Engineering. Eighth International Conference*, *FASE 2005 Held as Part of the Joint European Conference on Theory and Practice of Software*, *ETAPS Proceedings*, 4–8 April 2005.
22. Gravetter FJ, Wallnau LB. *Statistics for the Behavioral Sciences* (5th edn). Wadsworth Pub Co.: Belmont, CA, 2000.
23. Sourceforge. Open-source website. http://sourceforge.net/ [12 August 2007].
24. GraphViz. Dotty. http://www.graphviz.org/ [12 August 2007].
25. Jhala R, Majumdar R. Path slicing. *SIGPLAN Notices* 2005; **40**:38–47.
26. Korel B, Laski J. STAD—A system for testing and debugging: user perspective. *Proceedings of the Second Workshop on Software Testing*, *Verification*, *and Analysis* (*Cat. No. 88TH0225-3*), 19–21 July 1988.
27. Lanubile F, Visaggio G. Extracting reusable functions by flow graph based program slicing. *IEEE Transactions on Software Engineering* 1997; **23**:246–259.
28. Tan HBK, Kow JT. An approach for extracting code fragments that implement functionality from source programs. *Journal of Software Maintenance and Evolution* 2001; **13**:53–75.
29. Gallagher KB. Visual impact analysis. *Proceedings of International Conference on Software Maintenance*, 4–8 November 1996.
30. Ricca F, Tonella P. Web application slicing. *Proceedings IEEE International Conference on Software Maintenance*, *ICSM 2001*, 7–9 November 2001.
31. Hutchins M, Gallagher K. Improving visual impact analysis. *Proceedings 1998 IEEE International Conference on Software Maintenance*, *ICSM*, 16–20 November 1998.
32. Tonella P. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering* 2003; **29**:495–509.
33. Eisenbarth T, Koschke R, Simon D. Locating features in source code. *IEEE Transactions on Software Engineering* 2003; **29**:210–224.
34. Sartipi K, Kontogiannis K. On modeling software architecture recovery as graph matching. *International Conference on Software Maintenance*, 22–26 September 2003.
35. Zhang X, Young M, Lasseter JHEF. Refining code-design mapping with flow analysis. *Twelfth ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, *SIGSOFT 2004/FSE-12*, 31 October–5 November 2004.
36. Wilde N, Gomez JA, Gust T, Strasburg D. Locating user functionality in old code. *Conference on Software Maintenance 1992* (*Cat. No. 92CH3206-0*), 9–12 November 1992.
37. Wilde N, Scully MC. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance*: *Research and Practice* 1995; **7**:49–62.
38. Antoniol G, Gueheneuc Y-G. Feature identification: an epidemiological metaphor. *IEEE Transactions on Software Engineering* 2006; **32**:627–641.
39. Chen K, Rajlich V. Case study of feature location using dependence graph. *Proceedings IWPC 2000. Eighth International Workshop on Program Comprehension*, 10–11 June 2000.
40. Devanbu P. On ‘A framework for source code search using program patterns’. *IEEE Transactions on Software Engineering* 1995; **21**:1009–1010.
41. Kontogiannis K, DeMori R, Bernstein M, Galler M, Merlo E. Pattern matching for design concept localization. *Proceedings of 2nd Working Conference on Reverse Engineering*, 14–16 July 1995.
42. Kontogiannis KA, Demori R, Merlo E, Galler M, Bernstein M. Pattern matching for clone and concept detection. *Automated Software Engineering* 1996; **3**:77–108.

AUTHORS' BIOGRAPHIES

Minh Ngoc Ngo received her BSc in Computer Sciences in 2004 from Nanyang Technological University, Singapore. She is currently a PhD student in the School of Electrical and Electronic Engineering, Nanyang Technological University. Her research interests include the developments of techniques and tools to automate or partially automate software testing and analysis. She is a member of the ACM SIGSOFT and IEEE.

Hee Beng Kuan Tan received his BSc (1 Hons) in Mathematics in 1974 from the Nanyang University, Singapore. He received his MSc and PhD degrees in Computer Science from the National University of Singapore in 1989 and 1996, respectively. He is currently an Associate Professor with the Information Communication Institute of Singapore (ICIS) in the School of Electrical and Electronic Engineering, Nanyang Technological University. His current research interest is in Software Testing and Analysis.