프로그램 디버깅을 위한 휴리스틱 지식의 응용

서 동 근[†]

요 약

프로그램을 디버깅하는 과정은 원래 사고력이 집중적으로 필요한 작업이다. 이 프로세스를 수행하는 프로그래머를 돕기 위해서는 지식기반의 도구를 개발하는 것이 적절하다고 생각된다. 이 논문은 그러한 시스템에 대한 설계를 발표하기 위한 것이다. 주로 정형적 프로그램 기술과 자동 프로그램 이해에 기반을 둔 대부분의 지식기반 디버깅 도구들과는 달리 이 도구는 디버깅 휴리스틱에 기반을 둔 것이다. 이 도구는 지식베이스에 저장된 디버깅 휴리스틱을 이용하여 프로그램을 디버깅하는 프로그래머에게 무엇을 그리고 어느 곳을 살펴봐야 하는지 제안만을 하는 보조 도구이다. 이 논문에서는 여러 가지 유용한 디버깅 휴리스틱이 설명되며 그것들의 디버깅 프로세스에서의 사용법이 기술된다. 그 다음에는 그 지식들을 지식베이스에서 체계화하는 방안과 그 지식을 이용한 프로그램 디버깅 보조도구를 제안하고 토의한다.

On the Application of Heuristic Knowledge for Program Debugging

Dong-Geun Suh[†]

ABSTRACT

The process of program debugging is essentially an intelligence intensive process. It is thought viable to develop a knowledge-based tool to help programmer perform this process. This paper presents the design of such a system. Unlike other knowledge-based debugging tools which are mostly based on formal program specification and automatic program understanding, this tool is based on debugging heuristics. This tool is a debugging assistant which only suggests the programmer in program debugging what and where to examine using the debugging heuristics stored in the knowledge base. In this paper, a number of useful heuristic debugging knowledge are explained and their usage in debugging process are described. Then, a scheme to organize the knowledge in the knowledge base and an intelligent program debugging assistant using the knowledge are proposed and discussed.

1. Introduction

Program debugging is a process of detecting and correcting an anomalous behavior of a program. It is an integral part of programming tasks. Previous studies showed that more than half of the software development cost can be attributed to program validation and debugging activities in general [12]. Though program debugging accounts for significant amount of software development cost, it has received relatively little attention from the software engineering research community. There is little formed theory or systematic approach to guiding programmers to conduct debugging process. Nevertheless, almost every software professional and every institution has accumulated a large body of debugging heuristics.

† 정 회 원 : 동명정보대학교 정보공학부 교수
논문접수 : 1998년 8월 20일, 심사완료 : 1998년 10월 28일

The process of program debugging consists of two phases. The first phase is to understand the underlying program and locate the program fault that caused an anomalous behavior whereas the second phase is to repair the fault and account for the potential ripple effect. Nevertheless, it is the first phase which generally accounts for 90 percent of a debugging effort and hence the focus of all debugging tools [15]. Although there is a variety of debugging tools, most of them proceed as a passive program tracer which basically preserves information of current program execution in memory and allows a programmer to break the execution and examine the memory [2]. The more intelligent aspect of program debugging such as determining *where* and *what* to examine is entirely left for programmers to conduct.

Many attempts have been made to explore the feasibility of developing a knowledge-based tool to help perform program debugging. One major approach to developing such a system is based on formal program specification and automatic program understanding [9,13,16]. For instance, Proust takes as input a program and a description of its requirements, and finds the most likely mapping between the requirements and the code [9]. This mapping is in essence a reconstruction of the design and implementation steps in the program development. Program fault is identified when the system can not find possible mapping for some parts of the program. This approach generally requires a very large knowledge base and excessive processing time. As a result, none of the systems based on this approach has been applied in the field though tremendous progress has been made in the past decade.

Another potential approach is to mimic common debugging practices conducted by human experts who do not usually attempt to understand the underlying program when encountered with a debugging task. Rather, human experts mostly use intuitive heuristic to detect and locate program faults based on their observation on the anomalous program behavior. A system developed using this approach will act as an assistant which provides useful advice to guiding a programmer to perform a debugging task through frequent conversation with the programmer. When information about the buggy situation is provided, the system may suggest the programmer *where* and *what* to examine using the debugging heuristics stored in its knowledge base.

It is our observation that most experienced programmers conduct debugging process using this approach [20]: Encountering with a debugging task, an experienced programmer usually begins with analyzing the observed program failure against the intended program behavior. A number of hypotheses regarding the potential locations and possible faults can then be generated. Of all the hypotheses, the programmer will select the most promising one based on his/her experience and knowledge to further examine. When the hypothesis is verified, so is the program fault found. Otherwise, the information generated in the previous step will be used to improve the understanding of the program so as to refine the set of fault hypotheses. These steps will be repeated until the fault which caused the failure in the program is identified. Potential debugging heuristics used in these steps include: (1) effective as well as efficient approach to understand a buggy program and the observed anomalous behavior; (2) good fault hypothesis set for each specific program ; (3) potential fault locations with respect to each hypothesis; (4) strategy or intuition to select the most promising hypothesis and its associated rationale; (5) the best approach to examine and verify a chosen hypothesis.

Experts of program debugging means those who have more and effective heuristics to perform these steps. Accordingly, if a software system which possesses these debugging heuristics can provide appropriate knowledge to a programmer in a debugging session, the burden of the programmer will be reduced dramatically. The essential problem of developing such a system is what the effective knowledge is, how it can be organized and used to help a debugging programmer. The objective of this project is to develop an intelligent software *assistant* for program debugging. Presented in this paper is a proposed structure of the potential system. We will first briefly explain a number of (identified so far) useful heuristic debugging knowledge of the process. The system structure that accommodates the knowledge discussed is then proposed. Finally, an example of debugging cases that may be performed under the proposed system is presented.

2. Related Works

To ease the burden of programmers in debugging, several debugging tools were developed. In this section, what kind of debugging tools were developed and how they are related to our approach are explained.

2.1 Conventional Debugging Tools

The conventional debugging tools were designed to help mainly the tracing method in the bug locating process. They are the only debugging tools being used widely in practice. Symbolic debuggers [2] incorporate breakpoint facilities which allow the programmer to specify instruction location (breakpoint) where control is to be passed to the user's terminal during program execution. When execution is suspended and control is passed to the user, he/she can examine various components of the program state. However, the other activities, e.g., *what* and *where* to examine, which are more intelligent aspect of program debugging, are entirely left for the programmer to conduct. Our approach, presented in this paper, is trying to give advice to the programmer about what and where to examine.

Some more recent approaches are using data flow analysis: *data flow anomaly detector* [8] and *program slicer* [7,14,15,21]. The former is to check data flow anomalies such as uninitialized variables and definitions of variables which are not subsequently referenced. The latter shows the parts of a program such that "starting from a subset of program's behavior, slicing reduces that program to a minimal form which still produces that behavior [21]" , that is to address the tracing method. The limitation of the former is that the potential bugs that can be detected are minimal. The limitation of the latter is that, though it reduces the scope of the program for the user to examine in the debugging process, it is possible that the sliced program may include most of the program code within it. These two approaches give many clues to our research: mainly for generating potentially faulty locations. In addition, our approach also uses program analysis information such as the types of statements and types of variables which can be produced as a by-product by the program analyzer and program slicer.

2.2 Knowledge-based Debugging Tools

During past decade, several knowledge-based tools were proposed and developed to help the programmer in debugging more intelligently or to automate the debugging process. The knowledge used, approaches tackled, and emphases aimed by these tools are all different.

- **Program Analysis Approach**

The systems analyze the entire program to understand it and, in the consequence, find the bug [1,9,13,22]. The knowledge base of the system holds the programming models from the general design knowledge and coding technique of some programming languages. Using the programming models, a program description is constructed from the analysis of the actual program. This description will be compared with another program description which is constructed from the program specification. When the two descriptions are compared and, if there is any discrepancy, then the fault finding step starts: the part of program which is responsible for the discrepancy will be searched back through the list of reasoning. Since the computer should have every knowledge and technique covering each application domain and programming language to analyze a program and to construct a proper description of the program, the size of the knowledge base is very large. Also, there may be many possible paths in constructing those two program descriptions, so that the computation cost is very high. As a result, it is used only for debugging a toy program mainly in the education environment.

- **Using Description of Program failure**

The system, Falosy [18], searches the program to find chunks of the code in the program which are faulty. The knowledge base of the system holds the models of *stereotyped bugs* from programming language knowledge and general programming knowledge. Using the bug models, the incorrect patterns in the program is searched. Falosy takes as input the program to be debugged and a manually prepared list of output discrepancy. Using the description of output discrepancies, fault models (fault-driven, function-driven, or computation-driven) are triggered. Each fault model contains references to the expected-defects (the models of stereotyped bugs). Falosy searches where in the program the fault model and expected-defects are implemented. That is, an exhaustive search of the program for each bug model is conducted. Though understanding the entire program is not necessary in this approach, sufficient understanding of each part of the program must be achieved. This requires a large programming knowledge base and control knowledge to guide the partial understanding process and long computation time (it blindly searches the entire program to find the best match). Also, since the models should be exact (not to give false alarm), representing the types of bugs has problems: there might be large number of bug types (otherwise, very limited bugs can be detected), obtaining the exact description of output discrepancy is also not easy. And, actually, fault locating for only one type of program failure (file-read-error) was designed. However, this approach gave us a good insight of using debugging heuristics (that is, the notion of stereotyped bugs).

- **Internal Tracing Approach**

The systems analyze the program (or program behavior) and reduce the amount of information the programmer still has to examine to localize a bug [10,19]. One approach is using process-level dependency information. For example, MTA [19] requires two inputs: finite-state machine specification of the process' behavior and the interprocess message trace. Each message contains the IDs of the sending and receiving process, message type, message data, and time stamp. The system examines the message trace and outputs a list of suspicious process IDs and the anomalies seen in the message trace. The user will decide which process to focus and a process will be localized when the process seems solely responsible for the anomalies. In this approach, only process level localization is possible, if processes interact in other ways (such as through shared data) the bug localization performance will be impaired, and representing the finite-state machine description of processes will be hard for the procedure oriented program. Another approach is using the statement-level dependency information. For example, PELAS [10] requires two inputs: the *failure location* and the source code. The program is represented as a network of dependent statements from the failure location. Using the program execution information, the system shows the value of a variable in some statement which is in the scope of dependency and asks whether it is correct or not at that moment. When the user says the answer is correct, the statement is dropped from dependency list. When the answer is not correct, the fault is localized (the statement contains the fault). In this approach, the programmer should answer about the correctness of the value of a variable, the system didn't address the intermodule dependency problem, and if the error is a missing statement then it is impossible to localize a statement which does not exist. The main problem in this approach is that it uses only one heuristic: process dependency or statement dependency. Our approach will use as many heuristics as possible.

3. Debugging Heuristics

Due to the tremendous variety of programming languages, possible program errors, application domains etc., it is essentially infeasible to cover all theoretically infinite amount of debugging knowledge and/or heuristics. In our approach, we consider only domain independent debugging heuristics, and this may give a framework to further identify and represent those domain specific debugging heuristics. These debugging heuristics can be classified into three categories: those to generate fault hypotheses, those to select good hypotheses, and those to select effective examining and verifying strategies and processes. They are described in the following sections.

3.1 Heuristics of generating fault hypotheses

This type of heuristics is basically deduced from knowledge that associates program s with possible program faults. They can be further divided into four classes.

• *Programming language-based debugging heuristics*

Every programming language has its own features, limits, and constraints. Knowledge of these features and constraints often give some clues to help program debugging. For instance, when encountering with a *divide-by-zero* error in a (say, *Pascal*) program, an experienced programmer will consider some reasons such as variable aliasing, side effect of subprogram calling, real to integer type conversion that may cause a variable to become zero. However, overloaded division operator in an expression will not be included since it doesn't happen in *Pascal*. A simple representation of this knowledge is shown in (Fig. 1)(a).

• *Application domain-based debugging heuristics*

Knowledge on the possible implementations of underlying application domain is another source of debugging heuristics. Due to large variety of application domains, it is impossible to collect and store all the knowledge pertaining to all application domains. Nevertheless, human experts usually possess knowledge of common domains. For instance, when a sorting routine sorts a series of values into descending order whereas the intended behavior is to sort the list into ascending sequence, an expert would immediately check the logical operator embedded in the comparison statement (refer (Fig. 1) (b)).

(a) If Failure is Divide-By-Zero and
Language-Used is Algol-Family
Then Check-Aliasing and Check-Side-Effect and Check-Type-Conversion.

(b) If Failure is Incorrect-Output and
Order-of-Output is reverse and
Application is Sorting
Then Check-Logical-Op-in-Comparison-Stmt.

(c) If Failure occurs at A-Line with A-Variable
Then Suspicious-Loc = Dependency(A-Line, A-Variable).

(Fig. 1) Rules for Hypotheses Generation

• *Programming-based debugging heuristics*

There is no clear cut between knowledge of programming language and that of general programming expertise [4]. We regard programming knowledge as those concepts or common sense which can be applied in programming for any programming language. For instance, with the knowledge that a loop construct must contain five components, when encountering with an infinite loop error, an experienced programmer would check the use of loop control variable(s): is it appropriately used? is it incrementally updated, is it modified in a correct direction?

• *Run-time environment based debugging heuristics*

A program runs in a computing environment. Each environment has its own architecture, capability, and restrictions. A program must fit into its run-time environment in order to execute correctly. Some program s may be caused by incompatibilities to its run-time environment. Samples of run-time errors which are common to most environments include segment fault, incorrect file I/O, miscalculation of machine precision, arithmetic overflow, etc.

• *General program information*

Components of a program are generally bound together with one another through various connections. These connections are called program dependencies. Since program dependency represents semantic relationship between connected components, it has been widely suggested to be used in software maintenance tasks [3,7,14,15,21]. Moreover, it was indicated that this type of information is often used by human experts in tracing a program to find potential faults in a debugging process[14,15] (refer (Fig. 1) (c)).

3.2 Heuristics of selecting fault hypotheses

After generating a set of fault hypotheses, an experienced programmer usually uses his/her knowledge or experience with the software product to select the most promising one to pursue further. This process will be repeated until either the fault is located or the entire set is exhausted. The knowledge generally used for this purpose is described in the following.

• *Most frequently occurring program faults*

Psychological studies showed that programmers make mistakes more often in handling some programming constructs, e.g., pointer, *while* loop, etc. than others [5,17]. Therefore, it would be more promising to select a hypothesis that involves error-prone programming constructs than those with others. Additionally, it will be more productive to select those hypotheses that are associated with the errors that the original programmer committed most frequently. Of course, this type of information can only be obtained after observing the dynamic behavior of the underlying program for some period. A simple representation of this knowledge is shown in (Fig. 2)(a).

```
(a) If Fault-A and Fault-B are In(List-of-fault-Hypotheses)
       and
       Fault-A.Frequency  >  Fault-B.Frequency
    Then Check Fault-A first.

(b) If Failure-X occured and
       Fault-A and Fault-B are In(List-of-fault-Hypotheses)
       and
       (Failure-X,Fault-A).Frequency > (Failure-X,Fault-B).
       Frequency
    Then Check Fault-A first.

(c) If Suspicious-Locations is known and
       Traced-Locations is known
    Then Suspicious-Locations = Suspicious-Locations
    -Traced-Locations
```

(Fig. 2) Rules for Hypotheses Selection

• *Most frequently occurring program fault-failure pattern*

Moreover, the knowledge of most frequently occurring faults for a specific program failure provides another clue to select good fault hypotheses. However, it requires long time experience and in-depth analysis to obtain such knowledge. A way to represent this knowledge can be as the one in (Fig. 2)(b).

• *Dynamic program information*

The dynamic information of a buggy program often provides another source from which human experts draw heuristic to select good fault hypotheses. In specific, the following information is often used for this purpose.

- *Program execution profile*: Program execution profile is comprised of the information as to which modules actually executed during a particular running session. With this information in mind, an experienced programmer can effectively reduce the amount of information to examine in order to identify and locate a program bug [10,14] (refer (Fig. 2) (c)).

Program evolution information: It is known that errors are more likely to occur in those modules which have gone through recent modifications than others [5]. Accordingly, the information of recently modified modules provides useful clue to select fault hypotheses.

- *Program testing and debugging information*: Another rationale regarding program errors is that a module which was reported and detected errors recently are more likely to contain some other undetected errors [5]. Therefore, those modules with recent error report should be examined first in selecting a fault hypothesis.

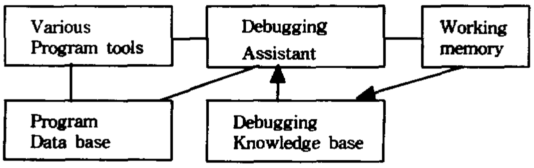
3.3 Heuristics of selecting effective strategy and examining process

Following selecting a fault hypothesis, an expert will use the most effective method to verify it. There are a variety of methods that human experts usually use to verify fault hypotheses: (1) visually reading and inspecting the code, (2) executing the program with designed inputs, (3) inserting defensive statements into the program and examining the execution result, and (4) setting up break points for an execution session and examining the values of some specific variables. It is noted, however, it is unlikely that a programmer will read and inspect the entire code when debugging a large-scale software. Rather, they usually use some other information such as dependency information to localize the scope of code that is relevant for inspection. Familiarity to a debugging tool also affects the way to choose an effective method to verify a fault hypothesis.

Some of the knowledge in this category can be represented explicitly as rules whereas others can be realized in the reasoning mechanism of the intelligent debugging assistant software. Also, this type of knowledge will affect the organization of the knowledge base and the extra service the system will render to the prospective users. In the next section, the design of a knowledge-based debugging assistant is presented and discussed that will accommodate all the above mentioned three categories of debugging heuristics.

4. Debugging Assistant System

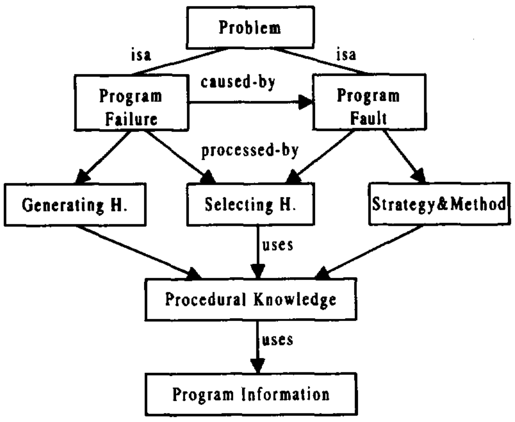
In this section, we present the design of a knowledge based debugging assistant based on the heuristics discussed in the previous section. Shown in (Fig. 3) is the architecture of the potential intelligent debugging assistant. The entire software is comprised of four components: a set of tools to analyze the underlying buggy program to collect such information as program dependence, complexity of modules, etc., and to collect various information about the buggy program such as program execution profile and program evolution information; a database to store the analyzed and collected program information; and a knowledge base to store all types of debugging heuristics. In particular, a debugging assistant component is designed to interface with the prospective users to guide them in conducting a debugging process. This assistant component will contain knowledge reasoning mechanisms and its own knowledge base in order to effectively process the heuristics stored in the knowledge base.



(Fig. 3) Program debugging environment

4.1 Organization of the knowledge base

As shown in (Fig. 4), the knowledge base designed for this intelligent debugging assistant is organized into seven components. The arrows in the figure indicate the interconnections between these components. Since a debugging process usually starts with understanding of an observed failure and proceeds toward the identification of associated program faults, these seven knowledge components are further organized into four layers to reflect the debugging process.



(Fig. 4) Debugging heuristics organization

In the top layer of the knowledge base, the *program failure* component and *program fault* component reside. *Program failure* component holds the information of all possible program failures and their respectively associated potential program faults. To facilitate its processing of the stored knowledge, this component is further organized into a hierarchy. At the top level, all program failures are classified into two categories: *system detectable failure*, and *user observed failure*. Each of these categories is classified further: *system detectable failure* can be further classified into *I/O failure*, *index related failure*, *arithmetic failure* which is again divided into *arithmetic overflow*, *divide-by-zero*, and *negative argument failure*, and so forth; *user observed failure* can also be divided into *incorrect result*, *infinite loop*, etc. Each failure in this component is associated with a set of potential faults that may cause the failure. Each fault with a failure is given a weight index to indicate the likelihood of the fault to cause the failure. Additional properties associated with each failure are its description of anomalous behavior, the location the failure occurred (it will be known during debugging), *generating heuristics*, *selecting heuristics*, etc. A subclass of a program failure will inherit the properties of its super class and may have its own properties. If more information is known for a program failure during debugging process, the more specific failure class will be matched and more specific debugging heuristics will be generated.

Program fault component stores all potential program faults (which will grow along with the use of the knowledge base). To facilitate its processing of the stored knowledge, this component is also organized into a hierarchy: at the top level, all program faults are classified into four categories (*application domain dependent fault*, *programming language dependent fault*, *programming dependent fault*, *run time environment dependent fault*), and each of these categories is classified further (for example, *programming dependent fault* is classified into *control structure related fault*, *value assignment related fault*, and *user defined data structure related fault*). Combining an observed program failure and potential faults generates a set of fault hypotheses. However, there is no clear relationship between program failures and program faults [6]. Each program failure may be associated with more than one potential fault and vice versa. For instance, *uninitialized variables* and *variable type mismatch* faults are both potential causes for both *divide-by-zero* and *incorrect result* failures. Associated with each program fault is a weight to indicate how often this fault was committed and detected within this particular buggy program, and a set of strategies and methods, which may be applied on the underlying buggy program to identify potential fault locations. Each strategy and method is also given a weight index to indicate its priority of being used to verify the associated fault hypothesis. Similar to program failure component, program faults in this component is also organized into a hierarchy of classes such as programming language related faults, application domain related faults, programming related faults, etc.

Sitting in the second layer are *generating heuristics, selecting heuristics, and strategy & method*. The *generating heuristics* component contains the knowledge to generate all possible program fault hypotheses: suspicious program faults and suspicious locations at which some program faults may reside. Since the description of a program failure may need more than one information and more detailed information about program failure may generate better fault hypotheses set, this component contains those knowledge to accommodate incomplete information situation and refinement of program failure.

The *selecting heuristics* component contains all the heuristic knowledge pertaining to the selection of most promising fault hypothesis from a given set. This selecting heuristic is generally stored as meta rules that uses such information as failure-fault weight, fault occurrence weight, to select the appropriate fault hypothesis for further pursuing. In addition to selecting fault hypothesis, this component also stores heuristics to select appropriate location to examine during the hypothesis verifying process. This type of knowledge will be discussed further later on.

Strategy & method component stores knowledge pertaining to the process of verifying each fault hypothesis. Each strategy and/or method stored in this component is associated with a number of procedures or subprograms for verifying the associated fault. For example, the program fault, *input data error*, can be verified when the following three conditions are satisfied: (1) there is an input statement; (2) the value of the actual input data is wrong; and (3) the input data is used directly or indirectly that caused an observed program failure. Three different procedures are clearly needed to complete this verification process.

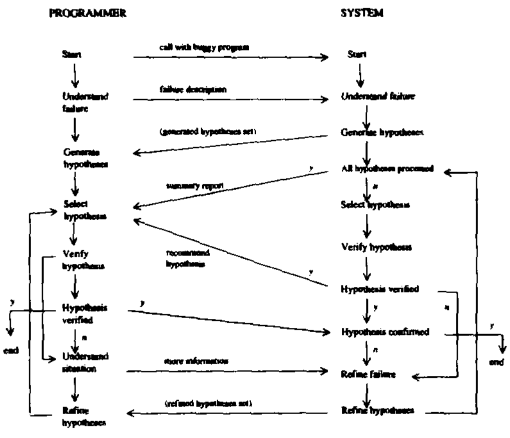
The third layer consists of *procedural knowledge* component. This component stores a number of procedures and subprograms that are triggered by the components in the second layer to generate potential fault locations using program information stored in the database and to get other program information such as the types of statements and types of variables.

The bottom layer is *program information* component which holds such information as program execution profile, debugging history that includes the types of program failures occurred and their respective faults detected, modules in which faults were detected and repaired, modules recently modified etc.

When a triggered procedure generates a number of potential fault locations, it is the meta rules stored in the selecting heuristic component to select the first location to examine using the information from the program information component as the selection criteria. The use of the knowledge described above and the debugging process that will be conducted by the software debugging assistant will be further discussed in the next section.

4.2 Debugging Assistant

Of all the components, the *debugging assistant* lies in the front-end of the system. Not only does it serve as the interface with the user, but it also triggers and coordinates the operations of other components in the *knowledge base* (KB). More importantly, it reasons the knowledge stored in different components of the KB, and provides guidance, and explanations in helping a programmer to perform a debugging process. The entire debugging process starts with an instantiation of a program failure and ends with satisfactory finding of a possible program fault. (Fig. 5) shows the interaction of a programmer with the debugging assistant during a debugging process.



(Fig. 5) Debugging process with Debugging Assistant

To begin the debugging process, the system takes a buggy program as input. Tools such as program dependency analyzer and dataflow anomaly detector are then activated to analyze the underlying program. Information including control flow, data flow, and data dependence, etc., of the program will be collected and stored in the *database* (DB) component. Should there exist program debugging history information, it will be loaded and stored into the *program information* (PINFO) component. Fault rate for each program failure will be loaded into the *program fault* (PFT) component. The *debugging assistant* (DAS) component is then activated.

Through a series of interactions with the user, the DAS obtains an observed program failure and possibly an error location depending on if the failure was detected by the system. Using *generating heuristic* (GH), the failure will be mapped into the *program failure* (PFL) component to generate a number of potential faults for this failure that forms the set of fault hypotheses. After storing the hypotheses into the working memory, the DAS then activates the *selecting heuristic* (SH) component. The SH component first prioritizes all the hypotheses in the given set using the weight indices associated with both failure-fault association and fault itself. The most promising hypothesis (or hypotheses) will then be selected. Next, the selected hypotheses will be mapped into the PFT component that generates a number of methods and procedures to be carried out to verify the selected hypothesis. If there exists more than one strategies for verifying a hypothesis, it is again up to the SH component to use its meta rules and some priority setting scheme to select the best strategy for further pursuing.

After the strategy and methods are determined, the procedures stored in the *procedural knowledge* (PK) component are activated. These procedures use the observed failure location together with the program static information, e.g., data dependence, control dependence, which is stored in the DB, to generate a set of potential fault locations. The SH component will then select and suggest the most promising location to pursue using the program debugging history information stored in the PINFO component. It is noted, however, if these exists more than one selected promising hypothesis (a tie in priority, for example), after generating locations to be examined, the SH may use the information in the PINFO to further select as to which hypothesis to pursue first. Furthermore, there may be a case in which no possibly examined locations can be generated. For example, the programmer observed an infinite loop symptom from the program execution but had no clue as to where it occurred. Upon this situation, it is the meta rule of the SH to suggest the programmer a number of methods to further examine the program in order to collect more information of the program's dynamic behavior.

During the knowledge inference process, the programmer will be notified of all the decisions made by the intelligent assistant and will have the power to intervene the operation of the DAS to select a different hypothesis or different examining location to pursue. After examining a location or verifying a hypothesis, the user should feed in the DAS of further information so that the DAS is able to mark some of the information in the working area and pursue next hypothesis or next examining location so as to generate new examining locations and/or new hypotheses. This process will continue until either the DAS is informed by the user that a possible fault and its location have been identified or the user choose to abort the process. The DAS will then store this debugging information into the PINFO and terminate the process.

5. Conclusion and Future Research

Program debugging is essentially an intelligence intensive process. It is thought viable to develop an intelligent tool to assist programmer in performing this process. Presented in this paper is the design of such a tool. This proposed intelligent debugging assistant is designed based on the process which human experts usually take and the heuristics they usually use in performing a debugging process. Several classes of debugging heuristics including those for generating fault hypotheses, those for selecting promising fault hypotheses, and those for proposing effective strategies as well as methods for verifying hypotheses, are presented. A scheme to organize all of the heuristics into a knowledge base and their respective interactions with other knowledge is also presented and discussed. This software can be extended to incorporate the formal approach to become an effective intelligent tool for high level program debugging. It can also be integrated with other maintenance tools to establish a true intelligent environment to assist in software maintenance. Currently we are in the process of implementing a prototype of this intelligent program debugging assistant.

After the prototype is implemented, we will conduct some experiments to examine the effectiveness of this approach. In the meanwhile, we will also conduct an investigation to determine the total percentage of heuristics being used in program debugging in the real world. It is also interesting to know how heuristic is mixed with formal program understanding in the debugging cases.

References

[1] A. Adam and J. P. Laurent, "Laura, a System to Debug Student Programs," *Artificial Intelligence*, pp.75-122, Nov. 1980.

[2] A. Cargill, "Implementation of Blit Debugger," *Software: Practice and Experience*, Vol.15(2), pp.153-168, Feb. 1985.

[3] Pao S. Chang, "Some Measures for Software Maintainability," *PhD Dissertation*, Northwestern University, June 1987.

[4] M. Ducasse. A.-M. Emde, "A Review of Automated Debugging Systems: Knowledge, Strategies and Techniques," *Proc. 10th International Conf. on Software Engineering*, pp.162-171, Apr. 1988.

[5] A. Endres, "An Analysis of Errors and Their Causes in System Programs," *IEEE Trans. on Software Engineering*, pp.140-149, June 1975.

[6] W. C. Gramlich, "Debugging Methodology (Panel Session Summary)," *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pp.1-3, Aug. 1983.

[7] S. Horwitz, P. Pfeiffer and T. Reps, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, pp.26-60, Jan. 1990.

[8] J. Jachner and V. K. Ararwal, "Data Flow Anomaly Detection," *IEEE Trans. Software Engineering*, pp. 432-437, July 1984.

[9] W. L. Johnson and E. Soloway, "Proust: Knowledge-Based Program Understanding," *IEEE Trans. Software Engineering*, pp.267-275, March 1985.

[10] B. Korel, "PELAS-Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, pp.1253-1260, Sept. 1988.

[11] S. Leestma and L. Nyhoff, *Pascal: programming and problem solving*, Macmillan Publishing Co. New York, 1984.

[12] Bennet P. Lientz, "Issues in Software Maintenance," *ACM Computing Surveys*, pp.271-278, Sept. 1983.

[13] F. J. Lukey, "Understanding and Debugging Programs," *Int. Journal of Man-Machine Studies*, Vol.12, No.2, pp.189-202, Feb. 1980.

[14] J. R. Lyle, "Evaluating Variations on Program Slicing for Debugging," *PhD Dissertation*, University of Maryland, 1984.

[15] G. J. Myers, "The Art of Software Testing," Wiley-Interscience, New York, 1979.

[16] A. Podgurski and L.A. Clarke, "A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Trans. on Software Engineering*, pp.965-979, Sept. 1990.

[17] C. Rich and Y.A. Feldman, "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development," *IEEE Trans. on Software Engineering*, pp.451-469, June 1992.

[18] N. F. Schneidewind and H.-M. Hoffmann, "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions on software Engineering*, Vol.SE-5, No.3, pp.276-286, May 1979.

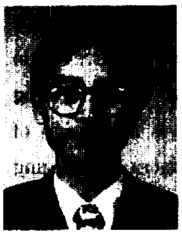
[19] R. L. Sedlmeyer, W. B. Thompson, and P. E. Johnson, "Knowledge-based Fault Localization in Debugging," Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, *SIGPLAN notices*, Vol. 18, No.8, pp.25-31, Aug. 1983.

[20] R. E. Seviora, "Knowledge-based Program Debugging Systems," *IEEE software*, pp.20-32, May 1987.

[21] M.-E. Suh, "On Knowledge Based High Level Program Debugging Using Heuristics," *PhD Dissertation*, University of Oklahoma, 1995.

[22] M. Weiser, "Program Slicing," *IEEE Tran. Software Engineering*, SE-10, pp.352-357, July 1984.

[23] H. Wertz, "Stereotyped Program Debugging: An Aid for Novice Programmers," *International Journal of Man-Machine Studies*, Vol.16, 1982.



서 동 근

e-mail : dkseo@tmic.tit.ac.kr

1979년 고려대학교 노어노문학과 졸업(학사)

1984년 고려대학교 대학원 노어노문학과(문학석사)

1990년 University of Oklahoma, Dept. of Computer Science (공학석사)

1995년 University of Oklahoma, Dept. of Computer Science(공학박사)

1995년 1995년 Knowledge Systmes Institute 교수

1995년 1997년 포스데이타(주) 선임컨설턴트

1997년~현재 동명정보대학교 전임강사

관심분야 : 소프트웨어공학, 소프트웨어품질관리, 지식표현, 분산처리