Automated Debugging Method Using Data Checking Specifications

HIRONOBU NAGANO*, SHUETSU HANATA*, MUNEO TAKAHASHI* and TETSURO MIKAMI*

An Automated Debugging Method for large-scale software systems and an experiment performed to determine its effectiveness are described.

Debugging steps performed by experienced programmers, such as analyzing memory dump lists, can be done by machines even though there are a variety of target software. In paticular, checking data values, using templates of data structures declared in a program, can be performed by the machine more quickly and accurately.

The method described here consists of Data Checking Specification description Language (DCSL), Data Checking Program Generation, and Execution History Information Compression for debugging. A prototype system CHASE (CHecking and Analyzing System for program Errors), that realizes partial specification of the method, was constructed and applied to two software systems (FORTRAN and COBOL compilers) in order to perform the automatic data checking experiment. About 15% of the seeded buges, which simulate real bugs of the compilers detected after delivery, were analyzed automatically by this experiment.

1. Introduction

Automatic testing and debugging systems [1]–[10], [12] have been implemented making use of assertions for compilers or pre-processors in order to accomplish automatic checking. As for debugging of delivered large-scale software, however, these assertions have the following disadvantages:

(1) Assertions need recompiling.

(2) Run-time checking is ineffective in delivered software. Because, assertions have been eliminated, in order not to deteriorate runtime efficiency.

(3) Correction of assertions may degrade the original source codes.

(4) Assertions depends on a programming language in which source codes are written.

Our solution for these problems is to describe checking conditions independent from source codes; Data Checking Specifications, acquired by programmers during development, are accumulated in Data Base; the specifications are utilized for automatic data checking in order to reduce the cost of testing and debugging as well as to give experience to maintenance programmers via the Data Base.

After delivery, insufficient information is provided for debugging of large-scale software systems. Only a portion of the error messages from the system and memory dump lists are usually given. Moreover, system errors cannot be detected so quickly by the system after the execution of abnormal codes thus debugging is difficult to perform through simulation of execution or backward tracking with minimum information. Only experienced programmers have been able to perform such difficult debugging using the knowledge about the software which they had acquired on their own.

*Data Communication Development Division, Yokosuka Electrical Communication Laboratory, Nippon Telegraph and Telephone Public Corporation, YOKOSUKA, JAPAN.

2. Features of Automatic Debugging Method Using Data Checking Specifications

2.1 Construction of the CHASE System

Fig. 1 shows a schematic overview of the CHASE system:

(1) Describing DCSL

Source codes are statically analyzed by the compiler to acquire program structure information, like module construction of the system, path information and static data structure information, which is then accumulated in the Program Data Base (PDB).

DCSL is used to describe both dynamic data structures and checking conditions modifying data structure information in the PDB.

(2) Generating Data Checking Programs

Possible static relationship checking is performed on all information in the PDB. Data checking programs are then generated according to each data structure modified by DCSL.

(3) Collecting Runtime Memory Information

The program to be debugged is run with the information collecting routine that will collect necessary information about the program's behavior. This information is stored in the execution history file. For the interactive debugging mode this routine does not access the file, but rather communicates with the terminal indicating the current derail point.

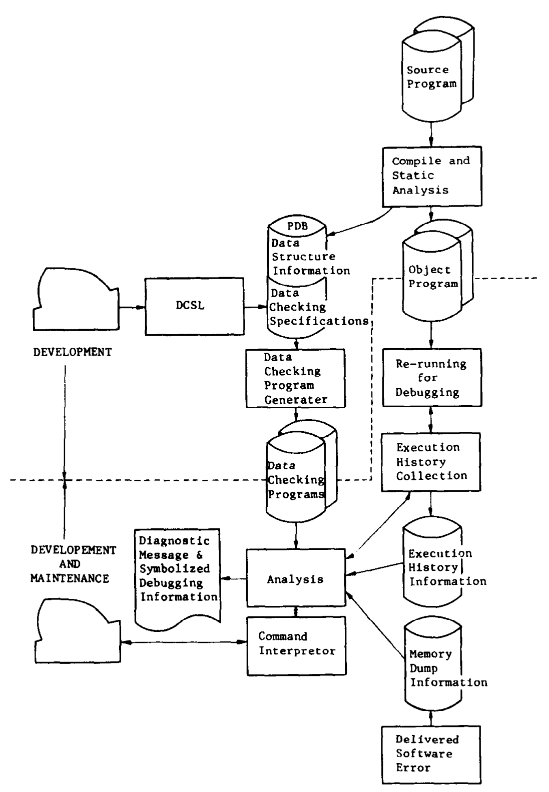

Source Program
Compile and Static Analysis
PDB
Data Structure Information
Data Checking Specifications
DCSL
DEVELOPMENT
Object Program
Data Checking Program Generater
Re-running for Debugging
Data Checking Programs
Execution History Collection
DEVELOPEMENT AND MAINTENANCE
Diagnostic Message & Symbolized Debugging Information
Analysis
Execution History Information
Command Interpreter
Memory Dump Information
Delivered Software Error

Fig. 1 Schematic overview of the CHASE system.

Derails can be set at entrance points and exit points of modules, and branch points of execution path.

(4) Checking Memory Information

Finally, generated data checking programs retrieve corresponding information from the execution history file or directly from the current memory and will automatically check each field of data structures.

A memory dump file, collected by the operating system when a delivered software system indicates a logical contradiction, is also checked in the same way.

Abnormal values will be shown with some identifications together with the symbolized data values and corresponding checking conditions.

Normal values can also be shown symbolically.

An abnormal value is traced backward in the execution history file to where the derail point was set.

Source code around the derail point is then shown in order to judge whether there is a bug or not.

2.2 Advantages

(1) Abnormal values can be checked automatically using data checking specifications described by DCSL.

(2) Data checking specification can be added after source code compiling. Therefore, it can be described and/or updated independently from source codes.

(3) The history file can be analyzed by generated programs faster than an interpretive checker, which frequently refers to the PDB.

(4) DCSL is effective in delivered software through memory dump file checking.

3. New Concepts for Debugging

3.1 DCSL

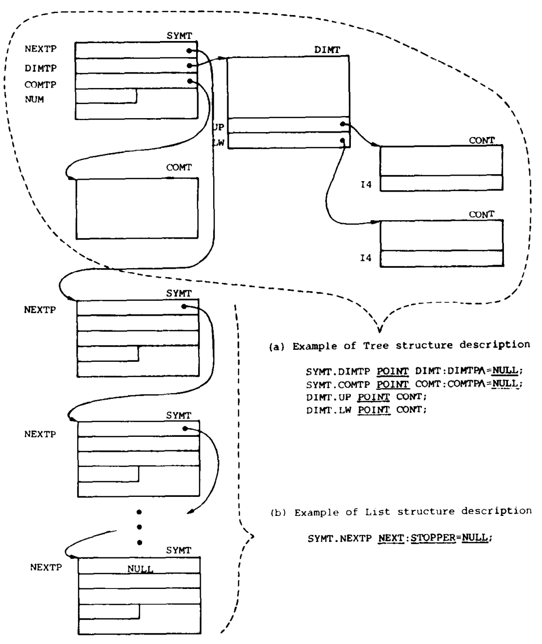
In widely used system description languages, which are subsets of PL/I, data attribute information, like BINARY or CHARACTER, and data structure template information, like ARRAY or STRUCTURE, can be collected by their compilers and accumulated in the PDB. Then, dynamic relationships between data structures in the PDB and Data Checking Specifications for each field of data structures are described in DCSL. Table 1 is an example of DCSL.

(1) Dynamic Relationships between Data Structures

In large-scale system programs, the memory work area is often used dynamically in a mixed form of list and tree structures using templates, such as based variables in PL/I. Such dynamic memory management, is left to system programmers so that detailed specifications tend to differ from program to program. For example, there are three methods of identifying the last element of a list structure which are typically found in large scale system program, a NULL next pointer, an element counter, and a last element flag. Dynamic re-

Table 1 Example of DCSL.

Classification		DCSL Statements
Dynamic Data Structure Specifications	Data Structure Chaining	POINT
	List Structure Construction	HEAD, NEXT, TAIL
	Data Structure Concatenation	CONCATENATE
	Existence Indication of Work Area	EXISTCOND
	Dynamic Data Length	LENGTH, UBV
Value Checking Specifications	Data Value Range	RANGE, ENCODE-CODED
	Logical Relation Expression	IMP, EXC
	Relation between Data Values	COND
	Selection of attributions declared for a data field	CASECOND
	Conditional Checking	CHECKCOND
	Comments for Data or Checking Spec.	NOTE



(a) Example of Tree structure description

```
SYMT.DIMTP POINT DIMT:DIMTP∧=NULL;
SYMT.COMTP POINT COMT:COMTP∧=NULL;
DIMT.UP POINT CONT;
DIMT.LW POINT CONT;
```

(b) Example of List structure description

```
SYMT.NEXTP NEXT:STOPPER=NULL;
```

Fig. 2 Example of dynamic relationships between data structures.

lationships, essential for analyzing the memory dump list, are implemented as program logics distributed in several source modules and are determined at execution time. Therefore, they are difficult to extract from source codes by static analysis.

The dynamic relationships describable in DCSL are as follows:

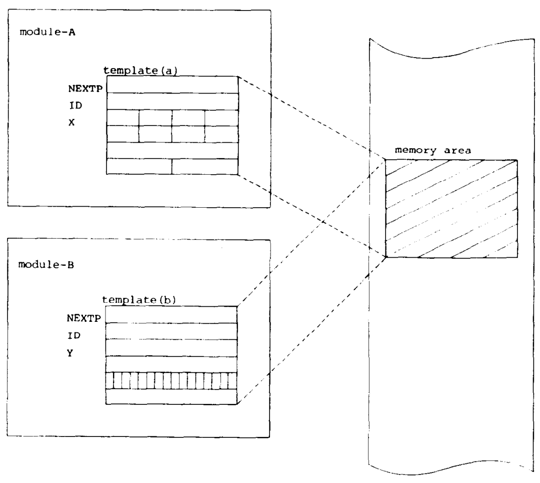
(i) Tree structure chaining among different data structure templates declared in source codes
Fig. 2(a) shows an example of a tree structure description. The first line means, for example, DIMT is pointed by SYMT.DIMTP when DIMTP is not NULL. Underlines show KEYWORDs of DCSL.

(ii) List structure chaining of the same data structure template
Fig. 2(b) shows an example of a list structure description. 'STOPPER = NULL' means the last element of SYMT is identified by a NULL value of its NEXTP.

(iii) Overlay references to the same memory work area by different data structure templates which may be declared in different source modules
Fig. 3 shows an example of overlay references in module-A and module-B using different templates (a) and (b) respectively. These templates have a common part (NEXTP, ID) and different structure parts (X, Y).

(2) Data Checking Specifications

Data Checking Specifications are classified into three groups according to their meaning. The first and the simplest one checks the value of a data structure field. The



Example of overlay reference description

```
TBL.X CASECOND:MODULE=A;
TBL.Y CASECOND:MODULE=B;
```

where declaration of the templates may be:

```
DCL 1 TBL BASED,
      2 COM,
       3 NEXTP PTR,
       3 ID    BIN,
      2 OWN   CELL,
       3 X ...,
           ...,
       3 Y ...,
           ...;
```

Fig. 3 Example of overlay reference to the same work memory area.

second checks the relationship between two or more data structure field values. The third checks the relationship between values and the excuting program codes. The first and the second extend the global assertions to all the system modules, while the third has the same meaning as local assertions.

(i) Data structure field checking specifications
Range checking of a variable, such as an iteration index or array subscript, can be specified throughout the system. (Fig. 4(a))
System status fields usually have some discrete value patterns. These patterns, corresponding to the enumeration type in Pascal language, can be defined by symbolic codes for both automatic checking and symbolizing. (Fig. 4(b))

(ii) Relationship between plural data values
They are given as Boolean expressions such as assertion descriptions. For example, the system control table holds system status fields that specify other table structures and their field values. (Fig. 4(c))

(a) SYMT.NUM RANGE (1: 7);
(b) DOT.COLOR CODED COLORDEF;
COLORDEF CODE RED=1, BLUE=2, YELLOW=3;
(c) CONT.I4 POINTED DIMT.UP>CONT.I4 POINTED DIMT.LOW;
(d) TBL.ID RANGE (0: 4) :MODULE=A;
TBL.ID RANGE (5: 9) :MODULE=B;

Fig. 4 Example of data checking specifications.

(iii) Relationship between data values and execution status
This relationship specify more limited conditions compared with (i) and (ii). Relationships described in (i) and (ii) are connected with the execution points of a specified module. Execution points are specified by a combination of one or more line numbers, labels and a module name.
Fig. 4(d) shows that RANGE check conditions for TBL.ID is specified with effective module names, A and B.

3.2 Data Checking Program Generation

Data Checking Specifications in the PDB are translated into Data Checking Programs.

(1) Automatic Data Checking

Table 2 shows a comparison of three automatic data checking methods. Automatic data checking can be done interpretively by referring to PDB information whenever it is needed. For a small program with a small-scale PDB, it is feasible and efficient to use a general purpose interpreter. For large-scale software systems, however, PDB access time becomes so long that the interpretive analysis method cannot be performed efficiently. Therefore, the generation method is introduced in our experiment to solve this scale problem.

The disadvantage of the generation method is that modification of a data structure declaration necessarily results in regeneration. However, modifications of data structure declaration are, considered so rare from system integration tests that this disadvantage is assumed not to be a problem.

Generated data checking programs are called from the CHASE command interpreter according to user demands.

(2) Requirements for Generated Programs

Generated program functions consist of (i) table addressing in the memory dump file, (ii) tracing pointer relations between data structures, (iii) data value checking and (iv) printing the results.

(i) Addressing
Table names demanded should be translated into memory addresses. It is not possible to address all tables directly, but addressing common or external tables is possible. Other tables are accessed by tracing pointer relations.

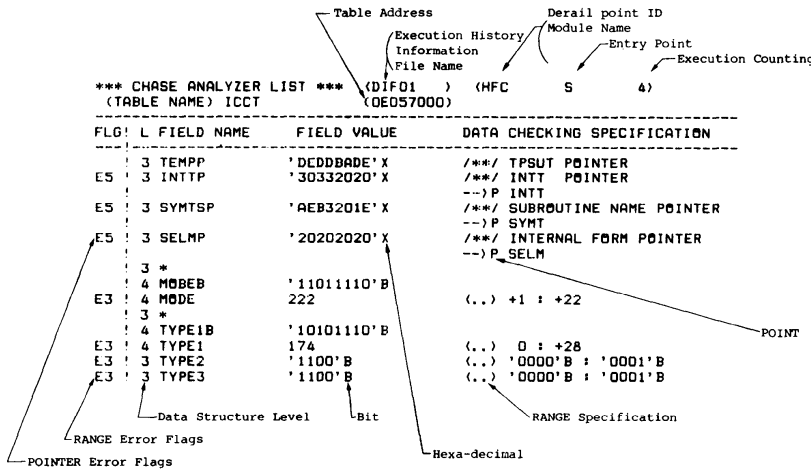
(ii) Table relation tracing
Pointer relations have been given by DCSL. They are used for table tracing from the tables described in (i).

(iii) Checking
Each value in the data structure field is checked by the conditions described in DCSL according to the data structure.

(iv) Printing
Each value is symbolized and printed with error flags, if there are any. All data structures and their values can be printed if necessary. Fig. 5 shows an example of Symbolized Debugging Information.
Header lines show the information identifications: the first line shows the execution history information file name and the derail point identifications; and the second line shows the table name and its address. These are given by an operator except the table address. The table address is translated from the table name, ICCT, automatically. The fourth line shows the titles for each column. FLG and L mean flag and data structure

Table 2 Comparison of data checking methods.

Method / Item	Data Checking Program Generation	Data Checking Interpretor	Run-Time Assertion
Description of Data Checking Condition	Register the DCS by DCSL		Describe in the Source Program by Programming (Assertion) Language
The Time for Data Chceking Condition Reference	Checking Program Generation Time	Analyzing Time (Run-Time or Postmortem)	Run-Time
Effective Range of Checking	The Whole System		In the Module
Checking Condition Modification	PDB Modification and Re-Generation of Checking Program	PDB Modification	Modification of Assertion Statements and Re-Compiling
Applicability for delivered Software	Because of independency, the Efficiency does not change		Because of redundancy the Efficiency decreases
Checking Ability	The Whole Memory through the whole execution time can be checked		Only Variables, which are described by Assertion Statements, are checked



```
*** CHASE ANALYZER LIST ***  (DIF01    )  (HFC      S      4)
 (TABLE NAME) ICCT           (0E057000)
------------------------------------------------------------------
FLG! L FIELD NAME      FIELD VALUE       DATA CHECKING SPECIFICATION
---!--------------------------------------------------------------
   ! 3 TEMPP           'DCDDBADE'X       /**/ TPSUT POINTER
E5 ! 3 INTTP           '30332020'X       /**/ INTT  POINTER
   !                                     --)P INTT
E5 ! 3 SYMTSP          'AEB3201E'X       /**/ SUBROUTINE NAME POINTER
   !                                     --)P SYMT
E5 ! 3 SELMP           '20202020'X       /**/ INTERNAL FORM POINTER
   !                                     --)P SELM
   ! 3 *
   ! 4 MOBEB           '11011110'B
E3 ! 4 MODE            222               (..) +1 : +22
   ! 3 *
   ! 4 TYPE1B          '10101110'B
E3 ! 4 TYPE1           174               (..)  0 : +28
E3 ! 3 TYPE2           '1100'B           (..) '0000'B : '0001'B
E3 ! 3 TYPE3           '1100'B           (..) '0000'B : '0001'B
```

Fig. 5 Example of error flags and symbolized data values.

level.

The function of (i) as well as (ii) is realized as one generated program for each software system. These functions may be altered according to the needs of the operating systems. The functions of (iii) and (iv) are generated for the data structures; their checking specifications are described in DCSL.

The top address of the data structure and the dumped memory are input into the generated program. The symbolized data values and error diagnosis messages are output from it.

The checking and printing functions have same simple patterns for all data attributes. Taking advantage of such features of a generated program, these patterns are precoded as macro-skeltons. Generation can be done easily combing these macro-skeltons and data checking specifications according to the data structure and its attributes. An example of a macro-skelton described in SYSL-macro language [15] and a part of a generated program corresponding to it are shown in Fig. 6.

3.3 Collecting and Compressing Execution History Information

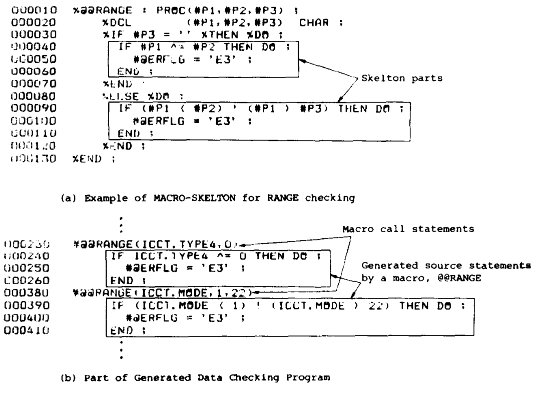
(1) Derail Points

Derail points are set at every entry and exit point of a module as well as every source code level branch point of execution paths in a module. A derail point is identified by a module name and a statement number of the source program. Derail point information is also accumulated by the compiler in the PDB in order to support source code level debugging.

As for the proto-type CHASE, the compiler generates routine calling codes for the information collecting rou tine with a parameter set of the module name and the statement number.

(2) Postmotem Debugging Mode

Usually, a module or function of a module changes a small part of the whole work area. Therefore, it is not necessary to collect the whole memory dump every time the memory collecting routine is called. Only all the differences in the whole work area between now and the last time it was called are needed, with the initial whole



```
%@@RANGE : PROC(#P1,#P2,#P3) ;
    %DCL        (#P1,#P2,#P3)  CHAR ;
    %IF #P3 = '' %THEN %DO ;
     IF #P1 ^= #P2 THEN DO ;
       #@ERFLG = 'E3' ;
     END ;
    %END
    %ELSE %DO ;
     IF (#P1 ( #P2) ! (#P1 ) #P3) THEN DO ;
       #@ERFLG = 'E3' ;
     END ;
    %END ;
%END ;
```

(a) Example of MACRO-SKELTON for RANGE checking

```
          :
¥@@RANGE(ICCT.TYPE4,0)
     IF ICCT.TYPE4 ^= 0 THEN DO ;
       #@ERFLG = 'E3' ;
     END ;
¥@@RANGE(ICCT.MODE,1,22)
     IF (ICCT.MODE ( 1) ! (ICCT.MODE ) 22) THEN DO ;
       #@ERFLG = 'E3' ;
     END ;
          :
```

(b) Part of Generated Data Checking Program

Skelton parts
Macro call statements
Generated source statements by a macro, @@RANGE

Fig. 6 Example of data checking program generation.

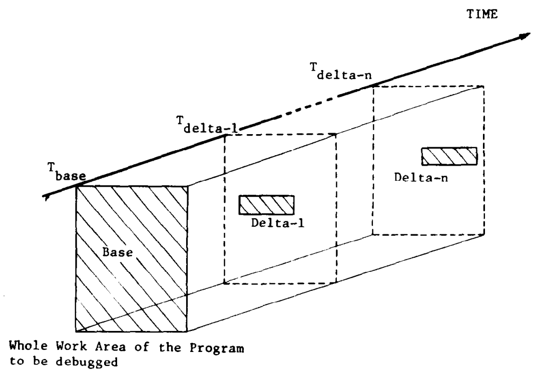


Fig. 7 Image of execution history information compression.

memory as the basement. Fig. 7 shows the image of the compression principle.

Crosshatched sections contain all the memory information to be collected: Delta-1 shows changed memory area at time T-delta-1 compared with that at T-base. Base memory information and Delta-1 are sufficient complete information for T-delta-1. Similarly complete memory information at T-delta-*n* is reconstructed using all Deltas $1 \sim n$.

Compressed Execution History Information consist of Base memory information and all Delta-*n* information including control information such as derail point identification, data addresses and lengths. The information collecting and referring routine keeps the current work area in a work file for efficient processing. To extract Delta-*n* information, the routine compares the current memory information with the work file. For forward data flow tracing of the execution history file, Delta-*n* information is overwritten on the work file at T-delta-$n+1$. Backward tracing can be realized easily in the same way: Delta-$n-1$ information is overwritten on the work file at T-delta-*n* in this case.

An experiment performed on a compiler shows that the execution history file can be compressed by 1/2000.

It is not possible to collect only changed variables described in source codes [14], when pointer variables are used for memory management. Because an incorrect data setting using an incorrect pointer value may result in destruction anywhere in the work area, in this case the whole work area should be collected for efficient debugging.

(3) Interactive Debugging Mode

The information collecting routine communicates with the terminal at the first derail point. If the interactive debugging mode is selected by the operater, it requests debugging points as a set of derail points. At the derail point, current memory is analyzed by data checking programs by order of the operater.

4. Experiment and Evaluation

An experiment on our method has been carried out. A prototype CHASE system was constructed and applied to two compilers (FORTRAN and COBOL) in order to evaluate the Automated Data Checking Method. The two compilers used were from a Commercial Time Sharing Service.

4.1 Scope of the Experiment

(1) DCSL

Relationships between different data structures were not supported, mainly because of generation difficulties. Internal data in each module were not able to give checking specifications, because the PDB size would have become too large.

(2) Generation

Generation was achieved to the macro facility in the system description language. Therefore, generated checking programs should be compiled into object programs. Source code generation has the advantage of being capable of being modified before being compiled. However, throughout the experiment, no modifications were required. Therefore, it is better to generate object programs directly in order to reduce handling.

(3) Execution History Information Handling

The differential memory dump collecting routine is implemented to compress the file size. Any data values at any derail points can be restored to the original state. It is possible, of cource, to trace differences which show the data flow from the original value to the last value with the execution points where the value is changed. Derail points were inserted as parts of the object code by the compiler.

4.2 Experiment

(1) Evaluation Criteria

The evaluation criteria for the experiment were as follows:

(i) Preparation costs for automatic debugging
(ii) Automatic detection ability of abnormal data value

(2) Seeding and Detecting

Fifty bugs simulating real bugs were seeded into three modules of the FORTRAN compiler and two of the COBOL compiler. Simulation was done so as to make the seeding easy. Bugs were seeded to cause abnormal compiler endings (20), incorrect compiler messages (17), and abnormal object code production (13). The simulated bugs were selected from the bugs that were analyzed to cause abnormal data value somewhere in the program. Such bugs constitute 44% of all bugs which have been reported after delivery for the sample systems.

Module names were known to two debuggers who had no special knowledge of these compilers. Debuggers were given the source code lists and the corresponding table specifications for the modules. They collected the execution history file and called the generated data checking programs interacting with the CHASE command interpreter in order to detect abnormal data values.

(3) Preparation Costs for Automatic Debugging

About 1% of the total development cost of the sample compilers was needed for development of the automatic debugging by the prototype CHASE system. Development costs may be neglected, because they are lower than the costs of making dump routines by hand. For the sample compilers, dump routines for external tables were made by hand in their development phase. They are generated automatically in the CHASE system.

(4) Ability of Detecting Abnormal Data Values

Abnormal flags were shown by the system for 17 bugs out of 50. These 17 bugs constitutes 15% of all reported bugs of the sample systems. Another 29 bugs would have been flaged if a full-set DCSL was implemented and enough specifications were given. It means that detection ability of the CHASE will highten to 40% for the sample compilers by the full specification of this method.

Because a large amount of detailed information is required to find the remaining four bugs, it is th ought to be difficult to do this automatically. It is unaccep table to describe DCSL for every source statement about every data structure.

5. Conclusions and Future Projects

An automated debugging method for large-scale software systems based on the Data Checking Specifications, as well as an experiment conducted on the method, are described. Data Checking Specifications are given to the Data Structure in order to provide the experience and knowledge of development programmers. Data Checking Programs are then generated for each Data Structure. Three types of memory information are applicable to these automatic Data Checking Programs: Compressed Execution History File, Current Memory at an execution derail point and Memory Dump File produced by an operating software system. The method is effective for both development and Maintenance Phases.

The Experiment, performed on two software systems (compilers), shows that the method is effective for bugs that cause abnormal data values.

Future projects are as follows:

(1) DCSL Extension: Relationships between plural data structures will be achieved combining Generation and Interpretive Methods.

(2) Application to Ada Programming Support Environment: Some DCSL descriptions for dynamic data structures may not be necessary, because the Ada language specification makes it possible to collect this information through the static source code analysis. This improvement will cause the PDB to increase in size, where nested-type information will accumulate. A size reduction should be realized soon thereafter.

(3) Man-Machine Interface improvement: Input-Output operation of the CHASE system will be refined using CRT-display terminals and other hardware.

Operation steps such as Generation handling will be simplified.

(4) Derail point setting methods using hardware interruption without object code insertion will be implemented soon.

Acknowledgement

The authors wish to thank Dr. Kimio Ibuki for giving them a chance to carry out this research, and also Dr. Iwao Toda for his helpful discussions and suggestions. Appreciation is also due to Shinji Takamura for encouraging them to undertake the challenge of this research.

References

1. BALZER, R. M. EXDAMS—EXtendable Debugging and Monitoring System, *AFIPS Conf. Proc.* **34**, (1969), 567–580.
2. FAIRLEY, R. E. An Experimental Program-Testing Facility, *IEEE Trans. Software Eng.*, SE-1, 4 (1975), 350–357.
3. FAIRLEY, R. E. ALADDIN: Assembly Language Assertion Driven Debugging Interpreter, *IEEE Trans. Software Eng.* SE-5, 4 (1979), 426–428.
4. HOWDEN, W. E. Symbolic Testing and the DISSECT Symbolic Evaluation System, *IEEE Trans. Software Eng.* SE-3, 4 (1977), 266–278.
5. HOWDEN, W. E. DISSECT—A Symbolic Evaluation and Program Testing System, *IEEE Trans. Software Eng.* SE-4, 1 (1978), 70–73.
6. HOWDEN, W. E. Theoretical and Empirical Studies of Program Testing, *IEEE Trans. Software Eng.* SE-4, 4 (1978), 293–298.
7. HOWDEN, W. E. Completeness Criteria for Testing Elementary Program Functions, *Proc. 5th ICSE* (1981), 235–243.
8. GANNON, C. Error Detection Using Path Testing and Static Analysis, *IEEE COMPUTER* (Aug. 1979), 26–31.
9. GANNON, C. and MEESON, R. AN EMPIRICAL EVALUATION OF STATIC ANALYSIS AND PATH TESTING, *Proc. AIAA Computers in Aerospace Conference II* (1979), 309–314.
10. RAMAMOORTHY, C. V. and HO, S. F. Testing Large Software with Automated Software Evaluation Systems, *IEEE Trans. Software Eng.* SE-1, 1 (1975), 46–58.
11. GOULD, J. D. and DRONGOWSKI, P. An Exploratory Study of Computer Program Debugging, *HUMAN FACTORS*, **1**, 6 (1974), 258–277.
12. ANDREWS, D. M. and BENSON, J. P. An Automated Program Testing Methodology and Its Implementation, *Proc. of 5th ICSE* (1981), 254–261.
13. DOD "STONEMAN" Requirements for Ada Programming Support Environment, 1980.
14. McGREGOR, D. R. and MALONE, J. R. Stabdump—A Dump Interpreter Program to Assist Debugging, *SOFTWARE—PRACTICE AND EXPERIENCE* **10**, (1980), 329–332.
15. TERASHIMA, N. SYSL—System Description Language, *ACM SIGPLAN notices* **9**, 12 (1974).
16. HANATA, S., TAKAHASHI, M., NAGANO, H., TANOMI, H. and MIKAMI, T. A method of softwere maintenance, using Program Data Base, *Proc. SIG Computer Inst. Electronics Comm. Engrs. Japan* EC81-8, (1981), 11–22 (in Japanese).
17. NAGANO, H., HANATA, S., TAKAHASHI, M. and TANOMI, H. Automated Debugging Method Using Data Checking Specifications, *Proc. WG Software Engineering IPSJ* **27**-1, (Nov. 1982) (in Japanese).

(Received Aug. 7, 1982)