Reprinted from —
AFIPS — Conference Proceedings
Volume 52

Arlington, VA 22209

HITS: A symbolic testing and debugging system for multilingual microcomputer software

by TAKESHI CHUSHO,
ATSUSHI TANAKA, and
ERI OKAMOTO
Hitachi, Ltd.
Kawasaki, Japan
and
AKINORI HONDA
and TORU KUROSAKI
Hitachi, Ltd.
Yokohama, Japan

ABSTRACT

The use of a large-scale computer is the key to the development of increasingly numerous and large-scale microcomputer software programs. HITS (Highly Interactive Testing-and-debugging System) constructs an integrated programming environment for 68000 microcomputer systems on a large-scale computer in cooperation with language translators. This system supports efficient and effective software validation from module testing through system testing. Functions of HITS are provided in the test-procedure description language, in which test data, expected results and the testing environment are described and separated from the target program. The main features are (1) symbolic support of both a high-level language and an assembly language, (2) module testing facilities such as driver and stub definitions, (3) a testing coverage monitor for branch testing, (4) debugging commands added temporarily to a test procedure from a terminal, and (5) a macro definition for language extension. HITS has already been used at many sites. In our early experience of applying it to the software development of various communication systems, software productivity and reliability were considerably improved.

INTRODUCTION

Software development for microcomputer systems is entering a critical stage. This is because the programming environment is still poor, even though microcomputers have been applied extensively to various fields, many of which have required high reliability. Furthermore, large-scale software has begun to be developed as 16-bit microcomputers have come into wider use. For example, we have developed 100 ~ 200 kilo steps of software for a digital switching system using 68000 microcomputers.

To date, almost all programming environments for microcomputer-software development have been constructed on the target microcomputer or on a development support system in which a microcomputer is embedded. Such resident support systems, however, provide limited facilities. That is, the programming language is usually assembly language. Furthermore, a debugger supports only dump, patch, breakpoint, and trace on a machine-language level. There are no operating systems with various useful utilities and powerful file management for software development as there are in a large-scale computer.

There are two effective solutions to these problems:

1. Programming in a high-level language and testing and debugging on a source-program level.
2. Using a large-scale computer for developing software, from programming through validation.

These solutions have been partially adopted in previous studies. For example, a high-level language, PL/M, for Intel's microcomputer families, was early developed. However, the software development system is not sufficient for software validation, because it mainly supports debugging, not testing such as the symbolic description of a test procedure.[1] Another example is the microcomputer software engineering facility, MSEF, which uses a minicomputer.[2] Although this system is aimed at supporting a wide range of microcomputer-software development, the testing facility is limited to management of the relationship between a target program, its input data, and results under the hierarchical file system.

We have incorporated both of these solutions in an attempt to deal with the problem of developing large-scale software for digital switching systems. First, a system description language for microcomputers, S-PL/H, has been developed and its cross compiler has been available in the Hitachi M-series computer system since the end of 1980.[3] S-PL/H is a superset language of PL/M and it provides both the basic facilities of PL/I and microcomputer-oriented facilities.

Next, a testing and debugging system for microcomputer 68000 software, HITS, has been developed for efficient and effective software validation using the large-scale computer.[4] This has been available since the spring of 1982. HITS constructs an integrated programming environment for microcomputer software development in cooperation with S-PL/H.

The main requirements for HITS are a wide range of supports for various aspects as follows:

1. Support ranging from small-scale software through large-scale software,
2. Target programs in both a high-level language and assembly language,
3. Testing facilities ranging from module testing through system testing,
4. Compatibility of testing facilities and debugging facilities,
5. Executions in interactive mode and batch mode.

This paper describes the design concepts and functions of HITS and some application results.

DESIGN CONCEPTS

Many different techniques and tools for software validation have been developed, such as data-flow analysis for automatic error detection, symbolic execution for automatic test-data selection, and assertions for correctness proof.[5] Many of them, however, are not practical for large-scale software validation because they require enormous computing resources.

Therefore, we still must depend on "exhaustive testing" in which a lot of data are evaluated against the corresponding expected results. Our goal is to improve the efficiency and effectiveness of such dynamic testing. HITS was thus developed on the basis of the following design concepts:

1. *Environment:* use of a large-scale computer
2. *Coverage:* support of module testing, integration testing, and system testing
3. *Function:* support and unification of systematic testing facilities and interactive debugging facilities
4. *Object:* program modules written in a high-level language and assembly language
5. *Ease of use:* minimization of preparations and operations, such as symbolic commands and a test-procedure library.

First, the use of a large-scale computer provides the following advantages:

- integrated file management for source programs, object programs, test data, test results, and path-coverage data,

- parallel processing of both module testing and integration testing under a time-sharing system.

Figure 1 shows the system configuration of HITS. The second item, systematic testing from module testing through system testing in this configuration, will be described in the next chapter.

The third item is based on the idea that testing and debugging cannot be separated. Of the conventional tools for software validation in practical use, there are many that provide only debugging facilities. The others provide only testing facilities. For example, although MTS[6] and TPL[7] are excellent tools for module testing, they do not support debugging. Furthermore, the former requires much preparation time because of target-language independence. The latter is limited to tests having only Fortran subroutine parameters. In HITS, when an error is detected by the execution of a test procedure that includes test data and the expected results, the test procedure can be executed again interactively while adding temporary commands for debugging.

The fourth item, support of both a high-level language and assembly language, is necessary for the development of system software because assembly language is used for the description of modules requiring device control or critical response time. For example, in the aforementioned digital switching system, 70% of all modules are described in a high-level language, S-PL/H, and 30% in assembly language. Therefore, these two languages are supported so that HITS may be available not only for module testing but also for integration testing and system testing. The fifth item, ease of use, is indispensable to support tools. A test procedure description language for HITS was designed taking this policy into consideration.

SYSTEMATIC TESTING

Software testing is performed in the following steps:

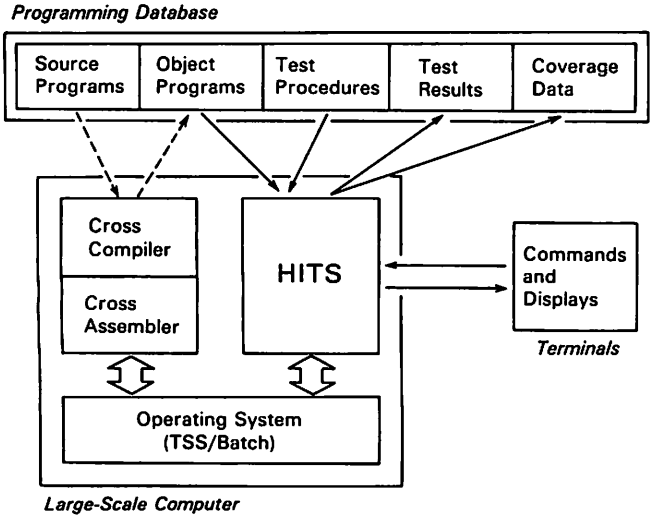

Figure 1—System configuration of HITS

1. module testing for validation of each module function,
2. integration testing for validation of interfaces between related modules,
3. system testing for validation of system function.

To be applied as widely as possible, a test system should systematically and uniformly support all of these steps and not depend on any one particular testing strategy such as bottom-up testing or top-down testing.[8]

HITS provides the following features for systematic testing:

1. All testing steps are supported by providing testing-environment simulation facilities for module and integration testing and a module-binding facility for integration and system testing.
2. Test data can be shared among all testing steps by using a test procedure that includes the test data.
3. Testing-coverage data for effective test-data selection and quality assurance are collected throughout all testing steps.

We will now look at these features in a little more detail.

Module/Integration Testing

Module and integration testing should be performed as thoroughly as possible, considering the following two axioms of productivity and reliability:

1. The later an error is detected, the more it costs to correct it.[9]
2. It is difficult to get a high testing-coverage rate at a later step.[10]

These testing steps, however, require a testing-environment construction that is complicated and troublesome. That is, an upper module, lower modules, global data, and input/output devices for the target module must be simulated. HITS reduces this work with testing-environment support facilities as shown in Figure 2.

Test Procedure

A test procedure includes test data, expected results, and testing-environment simulation, and is described in the test procedure description language that will be discussed later. This procedure is separated from a target module and can be shared throughout all testing steps by eliminating the simulation part, which integration of modules makes unnecessary.

Branch testing

Test-data selection methods are classified into functional testing, based on function specification, and structural testing based on program structure.[11] Branch testing is typical of the latter methods and is supported by HITS.[12] That is, the num-

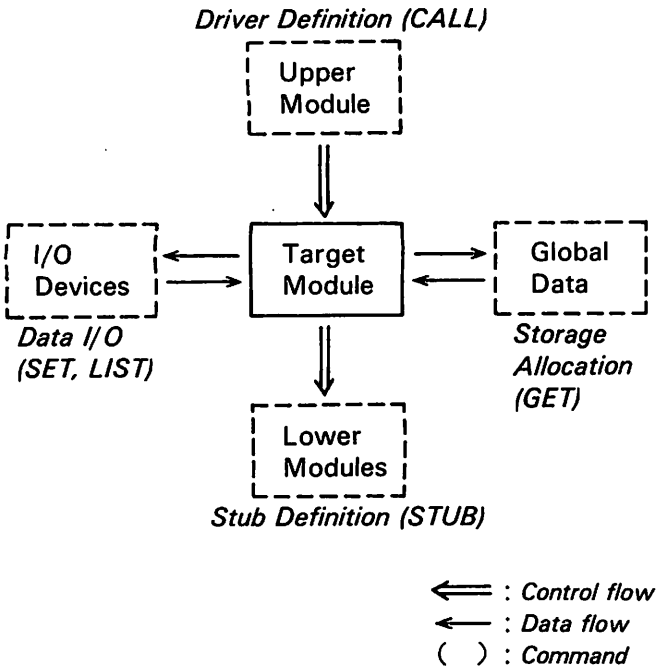

Figure 2—Environment support facilities for module testing

ber of executed branch directions is measured and the unexecuted parts are reported.

FUNCTIONS OF HITS

Test-Procedure Description Language

A test-procedure description language is designed as a command language rather than a procedural language because

1. HITS supports both testing and debugging in a uniform manner, and a command language is very suitable for interactive debugging.
2. Furthermore, a command language is easy to use even for test-procedure description and this satisfies the design policy of minimization of preparations and operations.

The structure of a test procedure is as follows:

```
PROC test-procedure name
  {commands in common use among the following
  test cases}
  CASE the first test-case name
    {commands only for the first test case}
  END
  CASE the second test-case name
    {commands only for the second test case}
  END
    .
    .
    .
END PROC
```

Commands in common use include commands such as those for binding of target modules and storage allocation of external global data. The test procedure is stored in a library and is executed by an EXEC command or is entered directly from a terminal. We would next like to look at command functions.

Simulation of Testing Environment

1. *Driver definition:* Upper module simulation is composed of value assignments to input parameters using SET commands, target-module invocation using a CALL or GO command, and result verification using IF commands. CALL may include value assignments to input parameters.
2. *Stub definition:* Lower module simulation is described in a STUB command whose subcommands may be composed of IF commands for input-parameter checks and SET commands for value assignment to output parameters.
3. *External global data:* Their storage is allocated using GET commands and may be assigned values by SET commands.
4. *Input and output:* Their instruction location is specified as a breakpoint by an AT command whose subcommands are SET commands for value assignments to input variables or LIST commands for display of output values.

Reduction of Test Procedure Description

1. *Macro-definition:* A list of commands used repeatedly is defined as a new extended command by a macro-definition facility. For example, a new command for a result check is defined as follows:

```
CLIST %CHECK
  IF &1 = &2 LIST '<O.K.>', '&1 = &2'
  IF &1 < > &2 LIST '<N.G.>', '&1 < > &2'
END CLIST
```

&n implies the *n*th parameter. Assuming that this macro is used as %CHECK (STATE, 3), if the value of the variable STATE is 3,

```
<O.K.> STATE = 3
```

is displayed. if not,

```
<N.G.> STATE < >3
```

is displayed.

2. *Simplification of object identification:* A QUALIFY command permits references to a local name without qualification that specifies the scope of the name. An EQUATE command replaces a complicated address expression with a new name.
3. *Variation of constant values:* A DATA command defines a sequence of constant values so that a test case can be executed repeatedly while varying only constant values.

4. *Communication among test procedures:* LOAD and SAVE commands permit a test procedure to use data values that are created by another test procedure.

Debugging Facilities

1. *Breakpoint:* The breakpoint is specified by an AT command which may include subcommands executed at the breakpoint. A breakpoint is expressed by the procedure names or statement numbers for the target program in S-PL/H. The specification of the procedure name causes an interruption and requests commands at the beginning of the procedure. The specification of the procedure name following END also functions at the end of the procedure. The statement numbers should be used only for interactive debugging, not for test-procedure description, so that modification of a target program does not cause modification of the test procedure. For the target program in assembly language, a breakpoint is expressed by the label names and hexadecimal offset address.
2. *Trace:* TRACE commands are used for the forward and backward control trace of branches or procedure calls, or for trace of data-value modifications. A BREAK option causes an interruption and requests commands at every trace event.
3. *Debug mode:* The BREAK option also functions at the beginning of the target-program execution if it is so specified before execution of a test procedure. Therefore, at that time, temporary commands for debugging can be entered without rewriting the test procedure in a library.
4. *Off-line output:* A large amount of trace data or dump can be output to a line printer instead of a display terminal by using an OUT option.
5. *Exception handling:* Exception handling can be described in a STUB command with an INTERRUPT option that includes an interruption condition such as an operation-code trap and an address error. References to undefined data are always detected.

DESIGN OF COMMAND LANGUAGE

The command syntax of HITS has the following features in comparison with conventional command languages:

1. procedural concept of block structure,
2. target language dependency,
3. abbreviation of command name.

First, it is desirable that constraints between commands be few. However, when HITS commands are used for description of a test procedure, some commands require subcommands. Therefore, the following seven block structures are introduced:

i. test-procedure block (PROC ~ END)
ii. test-case block (CASE ~ END)
iii. macro-definition block (CLIST ~ END)
iv. linkage block (LINK ~ END)
v. stub block (STUB ~ END)
vi. condition block (IF ~ END)
vii. breakpoint block (AT ~ END).

The last four blocks are used only if they have two or more subcommands. When there is only one subcommand, it is specified at their operands for simplicity. The second feature implies that a user can describe a test procedure on the target source-program level. For example, abstract operands of HITS commands, <instruction-address> and <data-address>, depend on a target language as shown in Table I. Therefore, it is easy to learn and use the command language. The third feature is provided to improve the efficiency of interactive debugging (full names of commands should be used in test procedures for readability). Our abbreviation rule is simple, that is, the latter part of a name can be truncated from an arbitrary position after the first character. If the truncated names of some commands are the same, the system decides which is which in advance, based on the frequency of use.

EXAMPLE

An example is given for explanation of a testing process using HITS. Assume that we develop a program for selecting the maximum of two values that are the minimum values of two groups of values. Two procedures, MINIMAX and MIN, are written in S-PL/H as shown in Figure 3.

A test procedure for integration testing of these procedures is shown in Figure 4, assuming that the lower procedure MAX and a caller of MINIMAX are not written yet. First, two modules, SUB1 and SUB2, including MINIMAX and MIN, respectively, are extracted from a library by an INCLUDE command that is a subcommand of a LINK command. Next, storage for the external global data, X and Y, is allocated. Then, a stub for MAX is defined and several test cases follow.

One of the test cases, C07, is composed of value assignments to global variables, X and Y, invocation for MINIMAX, and result check. The definition of %CHECK has been previously mentioned. Here, two other interesting macro-definitions can be used, namely %PRE and %POST. They are assertions for verification of the precondition and postcondition of a procedure, and are defined as follows:

```
CLIST %PRE
  AT &1 DO
    IF &2 RESUME
    LIST '&1 PRECONDITION: &2 is false.'
  END
END CLIST

CLIST %POST
  AT END &1 DO
    IF &2 RESUME
    LIST '&1 POSTCONDITION: &2 is false.'
  END
END CLIST
```

TABLE I—Command operands differing between S-PL/H and assembly language

Abstract Operand	Details for S-PL/H	Details for an Assembly Language
<instruction-address>	procedure name or statement number	label with offset
<data-address>	variable name	label with offset, indirect addressing, and register indexing

For example, these commands may be inserted before a CALL command in the test case C07 as follows:

```
%PRE (MIN, A(1) > -1)
%POST (MIN, I < 11)
```

The first command verifies that the input parameter A to procedure MIN has at least one valid value. The second verifies that the array variable A was never erroneously referred to out of range in procedure MIN.

When the test case C07 is executed, the following error message is output at %CHECK (RESULT, 2):

```
<N.G.> RESULT < >2
```

An example of interactive debugging for this error is shown in Figure 5. The test case C07 is executed with the debug mode

```
SUB1: do;
  dcl (X,Y) (10) integer external;
    .
    .
  MINIMAX: proc (var M) public;
    dcl (M,MX,MY) integer;
    call MIN(X,MX);
    call MIN(Y,MY);
    M=MAX(MX,MY);
  end MINIMAX;
end SUB1;

SUB2: do;
  MIN: proc (A,var B) public ;
    dcl A(10) integer;
    dcl (B,I) integer;
    B=A(1);
    I=2;
    do whlile A(I) >= 0 ;
      if A(I) > B then B=A(I);
      I=I+1;
    end;
  end MIN;
end SUB2;
```

Figure 3—A sample of a target program

```
PROC TP21
  LINK INCLUDE SUB1,SUB2
  GET X,Y
  STUB MAX(P,Q) DO
    SET MAX=P
    IF P < Q SET MAX=Q
  END
    .
    .
  CASE  C07
    SET X=(1,3,5,7,-1)
    SET Y=(2,4,6,-1)
    CALL MINIMAX(RESULT)
    %CHECK (RESULT,2)
  END CASE
    .
    .
END PROC
```

Figure 4—A sample of a test procedure

(BREAK option). At the beginning of MIN, the value of the input parameter A is checked. Then, MIN is executed while tracing for modifications of the output parameter B. Finally, the cause of the error in an if statement is detected, and this test procedure is terminated.

APPLICATION OF HITS

This system has been released to many factories and laboratories since the spring of 1982. The following advantages of HITS were confirmed.

1. *Writability:* The average number of commands in a test procedure is 4.4 ~ 5.6 per test case for module testing of a digital switching system, although the number of com-

```
{ ready }
OPTION BREAK
EXEC TP21(C07)
{ break at MINIMAX }
AT MIN LIST A
RESUME
{ display and break at MIN }
TRACE DATA(MIN#B)
AT END MIN
RESUME
{ display and break at the end of MIN }
STOP PROC
{ ready }
```

Figure 5—An example of interactive debugging

mands depends on such things as the number of input and output parameters, the number of external data, and similarity among test cases.
2. *Operability:* The target program is automatically tested by entering an EXEC command. This is because various operations required by a conventional debugger are automated or assembled into a test procedure.
3. *Reliability:* The quality of a target program becomes visible with the use of a testing-coverage facility and is improved by adding test cases for unexecuted branches. Reliability of testing is also improved because a test procedure is described on the target-program source level and clearly corresponds to a target program and its testing specifications.
4. *Productivity:* Productivity is improved by the following factors:
   i. early error detection by promotion of module testing,
   ii. high efficiency of test-data generation, execution, and result check
   iii. quick debugging.

   In our experience, when HITS was applied to only module and integration testing, testing cost was reduced by 35% in comparison with the previous testing method using the target computer. For a target program applicable to system testing, testing cost was reduced by 45%.
5. *Maintainability:* It is easy to modify and add test cases because a test procedure is separate from a target program. The test procedure is shared among module, integration, and system testing with only minor changes, and is also available in the maintenance phase of a target program.

CONCLUSIONS

A testing and debugging support system, HITS, for microcomputer 68000 software was developed for efficient and effective software validation using a large-scale computer. The main features of HITS are as follows:

1. All steps of module testing, integration testing, and system testing are supported while sharing test data and accumulating testing-coverage data.
2. Module-testing support facilities for simulation of an upper module, lower modules, external global data, and input/output devices are provided.
3. Test data, expected results, and environment simulation are assembled in a test procedure that is executed under both batch and interactive modes.
4. Both a high level language, S-PL/H, and assembly language are supported on the source-program level.

HITS has already been released to many sites and has improved software productivity and reliability.

ACKNOWLEDGMENTS

The authors wish to express their gratitude to Dr. Takeo Miura for providing the opportunity to conduct this study. They are also indebted to Tan Watanabe, who designed S-PL/H, for his invaluable technical assistance, Mitsuyuki Masui for comments on drafts of the functional specification, and Tatsuro Oishi for modification of the S-PL/H cross-compiler and the cross-assembler that pass symbolic tables to HITS.

REFERENCES

1. *Guide to Intellec Microcomputer Development Systems.* Santa Clara, Calif.: Intel Corporation, 1978.
2. Eanes, R. S., C. K. Hitcon, R. M. Thall, and J. W. Brackett. "An Environment for Producing Well-Engineered Microcomputer Software." *Proceedings of the 4th International Conference on Software Engineering,* 1979, pp. 386–398.
3. *Hitachi Microcomputer System: 68000 Super-PL/H Language Manual.* Tokyo: Hitachi Ltd., 1981.
4. Chusho, T., T. Watanabe, T. Kurosaki, and T. Yamamoto. "Design Concepts of a Microcomputer Software Testing and Debugging System." *The Fall Conference of Information Processing Society of Japan* (in Japanese), 1981, pp. 419–420.
5. Miller, E. F., and W. E. Howden. "Tutorial: Software Testing and Validation Techniques," IEEE Catalog No. EHO 138–8, 1978.
6. *Module Testing System (MTS) Fact Book.* London: Management Systems and Programming Ltd., 1972.
7. Panzl, D. J. "Automatic Software Testing Drivers." *Computer,* 11 (1978), pp. 44–50.
8. Myers, G. I. *The Art of Software Testing.* New York: Wiley-Interscience, 1979.
9. Sorkowitz, A. R. "Certification Testing: A Procedure to Improve the Quality of Software Testing." *Computer,* 12 (1979), pp. 20–24.
10. Holthouse, M. A., and M. J. Hatch. "Experience with Automated Testing Analysis." *Computer,* 12 (1979), pp. 33–36.
11. Howden, W. E. "Applicability of Software Validation Techniques to Scientific Programs." *ACM TOPLAS,* 2 (1980), pp. 307–320.
12. Miller, E. F. "Program Testing: Art Meets Theory." *Computer,* 10 (1977), pp. 42–51.