Advanced firmware verification using a code simulator for the IBM System z9

K. Theurich
A. Albus
F. Eickhoff
D. Immel
A. Kohler
E. Lange
J. von Buttlar

Our methods for simulating host firmware of the IBM System z9™ facilitated rapid development from first power-on of the system to achieving a platform with a functional operating system. Hundreds of code bugs were eliminated before the code was run on System z9 hardware for the first time. This paper describes the methods used in host firmware simulation for early and efficient firmware tests. The central element for firmware simulation is the Central Electronic Complex Simulator (CECSIM), which offers new facilities to manage the hardware of the simulated system. This management includes concurrent configuration changes of processors, memory, and I/O along with the ability to automatically test complex system functions. To verify correct implementation of the z/Architecture™, we introduced a new test-case framework called the Verification Interface for System Architecture, or VISA, which is used in simulations as well as on the actual system. All of these features are used separately and in combination. A comprehensive and flexible regression environment ensures periodic execution of the test scenarios, and code path coverage measurements show the degree to which the code was actually verified.

Introduction

The use of simulations for testing and debugging firmware and software has a long history. In addition to typical unit test environments, in which only certain parts of a system are considered, the need for a full system model [1–3] is becoming increasingly important given the growing complexity of systems. Such comprehensive testing is crucial for the IBM System z*, with its huge firmware stacks in various subsystems.

The IBM System z9* uses firmware to implement functions that include the execution of complex instructions in the CPUs, I/O operations performed by the system assist processors (SAPs), the management of logical partitions (LPARs), and various recovery and serviceability features. For reasons of cost and product development time, each firmware component must be verified using a simulation environment. We use CECSIM (Central Electronic Complex Simulator) [4, 5], a firmware simulation platform that is described in this paper.

CECSIM provides a set of advanced debug features and is an essential part of the firmware development process for the IBM System z. CECSIM provides a simulated system environment for firmware test and verification. Several years ago, our test effort in this environment focused on the preparation of the initial debug phase on the actual hardware. Complex functions were not within the scope of CECSIM tests. However, with the development of even more demanding and rapid product plans, simulations were beginning to be required for verification of complex functions early in the project cycle, at a time when no actual hardware exists. In addition, firmware development teams had to reduce the number of expensive engineering systems used for verification. Therefore, such teams were required to provide better-quality tools and tests to the test teams in order to accommodate the test plans and strict schedules.

These requirements reinforced the need for an environment based on CECSIM in which the majority of the firmware can be tested without requiring actual

0018-8646/07/$5.00 © 2007 IBM

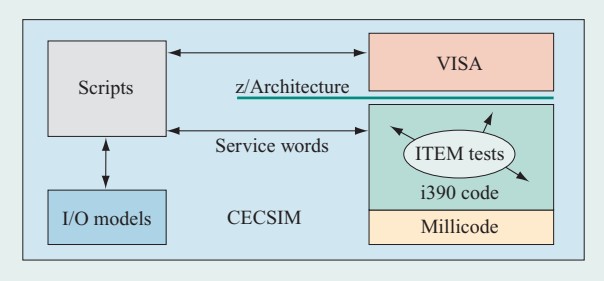


Figure 1

Overview of topics addressed in this paper. (VISA: Verification Interface for System Architecture; ITEM: i390 Test Case Execution Monitor.) The entire light blue box represents CECSIM; the green and yellow boxes represent the codes that are run within CECSIM. The bar labeled z/Architecture indicates that VISA runs at the z/Architecture level within CECSIM. The arrows extending from ITEM tests indicate calls to certain i390 functions.

hardware and in which the developer is supported by enhanced debug capabilities. An automated regression-test environment also ensures the required level of quality. Periodically our team gathered path-coverage data [6] from the regression runs; this data tells the team which part of the firmware is executed in order to indicate the quality of the test package. Our use of the terms *regression tests* and *path coverage* is clarified in the sections on regression runs and coverage near the end of this paper. In brief, regression testing of firmware permits the stability and feature interactions to be tested as the firmware base matures. Care must be taken to frequently determine whether a desired design enhancement has not in fact caused a degradation or regression in some aspect of the system. Regression testing is a lengthy process which must be conducted as firmware is updated in order to verify that no existing functionality is compromised.

A conceptual overview of various topics addressed in this paper is shown in **Figure 1**. The various phrases in Figure 1 are clarified in the following sections.

Two levels of processor firmware exist for the IBM System z. The lowest level, referred to as millicode [7], implements performance-critical functions, complex z/Architecture* (i.e., System z architecture) instructions that are not implemented in hardware, and functions for direct control of the hardware. Millicode is written in high-level assembly language and is executed on all processors.

The higher level of firmware, referred to as i390 (internal 390) [8], implements functions such as the channel-subsystem, recovery, and power-on-reset functions. The i390 is written in PL.8 [9] and C. The major portion of i390 code is executed on SAPs.

CECSIM allows the running of i390 code and of millicode with excellent performance characteristics [4]. CECSIM also provides a simulated system environment that includes multiprocessing (MP) capability and allows for the testing of the majority of the implemented firmware functions. The applications of CECSIM range from unit tests of individual i390 functions to a complex system test environment with additional components connected to CECSIM via network connections.

CECSIM can be connected to a support element (SE), which is the system console of an IBM System z. With CECSIM, users can interact with the SE as if processor hardware were connected and available. In the standalone mode, in which no SE is connected, CECSIM scripts (small REXX programs) mimic the SE, and the user can communicate with the simulator through the use of commands, full-screen panels, and scripts. REXX (REstructured eXtended eXecutor) is an interpreted programming language.

The communication between the SE and the system is based on service words, which are structured data packets. This service word communication is also used when testing with CECSIM in standalone mode, and this paper focuses on the use of CECSIM as a standalone environment. CECSIM scripts (see the section on the REXX interface) can send service words to the central electronic complex (CEC) and retrieve the responses from the CEC. In most cases, the service words carry operator commands, such as commands for partition activation or retrieving status information. Service word handlers have been written with these capabilities in order to simulate the basic responsiveness of an SE.

CECSIM debug facilities

During firmware development, the user relies on a complex set of CECSIM debug facilities that allow developers to locate and identify bugs in the code. The facilities available in the simulation environment go far beyond what is provided in the actual machine environment, where developers are limited to a function that displays and alters memory. No SE is required for these debug facilities, because they are all available from the command line of the simulator.

Display/Alter

As just mentioned, CECSIM provides a rich set of display and alter facilities that are used to access memory locations, registers, and many kinds of special data areas that reflect the current state of the simulated processor and attached I/O hardware adapters. Data that is being displayed may be dynamically overwritten in order to correct an undesired state and to continue the debug process with the current code load. Moreover, errors may

be injected, using this feature, in order to debug recovery facilities of the firmware.

Tracing

A number of trace facilities allow the tracking of instruction streams and storage alterations. Millicode, i390 code, and software are simulated together, and traces can be established separately for each code layer. The user may request that CECSIM halt a single processor or all processors related to an event being traced, and the instructions may be stepped sequentially. For i390 code, traces may also display the name of the i390 routine and the instruction offset within that routine. This is especially useful for tracing unexpected storage alterations. When software is run in logical partitions (LPARs) [10], the same trace facilities may be applied to a specific partition.

Additional events such as interrupts or special i390-only instructions can be traced. A "traceback" is automatically generated for program interrupts in i390 code. Such a traceback lists the names of all i390 routines in the current call chain. This traceback may also be requested explicitly at any time with a user command in order to obtain a snapshot of the current activity in the simulated processor. Further trace facilities exist to monitor activity in the I/O interface and communication with the SE.

REXX interface

CECSIM allows developers to write scripts in REXX, an interpreted command language [11]. All available commands may be issued from the scripts, and an extract facility allows data to be retrieved and stored in REXX variables. Such data includes memory and register contents, addresses of i390 routines and data structures, and current configuration information that includes the number of CPUs, the memory size, and the number of I/O hubs.

Many CECSIM scripts exist to support debugging; for example, scripts are available to create a formatted display of a data structure, to load and start programs, or to verify expected results. In addition to these general scripts, many CECSIM users have written their own scripts that are adapted to their special needs.

The combination of CECSIM with REXX forms a very powerful environment that offers comprehensive capabilities. Beyond analysis, debug, and the handling of service words, additional uses of REXX scripts are described below, especially those involving the execution of firmware tests (which we frequently refer to as test cases), injection of errors, and controlling of regression tests.

Event facility

Another CECSIM debug feature is the event facility. For certain situations, such as "reset complete" or "channel-path initialized," CECSIM produces an event that can be captured by a user command or, more often, a CECSIM script. Events may apply to a particular processor or to the system as a whole.

SIMCALL facility

The SIMCALL (simulation call) facility allows the simulated code (usually i390 code or millicode) to request a special service from the simulator. It is implemented as an instruction that does not exist on the actual machine. The instruction can also be enabled for software that is run in the simulation environment.

One function of the SIMCALL instruction is associated with the event facility. A firmware-generated event may be raised through the use of a SIMCALL. As is the case for an event raised by CECSIM itself, a CECSIM script can wait for this firmware-generated event, or the event can be captured in order to trigger the execution of a CECSIM script that, for instance, verifies the implemented function or injects an error.

Error injection

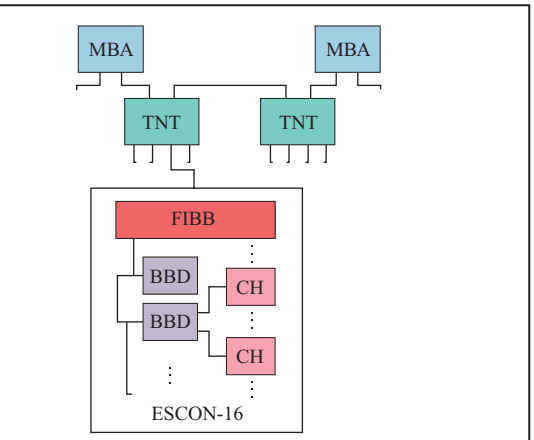
CECSIM provides various ways to inject errors, usually through CECSIM scripts. Memory may be manipulated, for example, in order to change the contents of a control block.

Other injection methods make use of interfaces to a CECSIM I/O model of a memory bus adapter (MBA), an I/O channel, or a Parallel Sysplex* channel. (A sysplex, or system complex, comprises one or more System z processors joined as a single unit. Parallel Sysplex technology permits multiple mainframes to act as one.) For example, the interface to the Parallel Sysplex allows both static and dynamic error injections. For dynamic error injection, in response to a specific I/O model event, a part of a script is executed that injects an error. Such script calls for analysis or error injection can even be issued from within a VISA test case.

The scripts can be called with one of the following as an argument:

- *init* – to establish those variables that are to be maintained across the various calls, and to instruct the I/O model to call for certain events. This may be called from a VISA test case.
- *event* – used when the I/O model calls the script in response to the expected events.
- *cleanup* – checks the variables and sets the return code as appropriate. This may be called from a VISA test case.

A wide range of options is offered, and although they have not yet been fully exploited, today hundreds of




```
...
<mba id="mba_02" addr="2">
 <tnt id="tnt_020" addr="1" alt_path="tnt_050">
  <fibb id="fibb_0230" addr="2">
   <bbd id="bbd_0230" addr="0">
    ...
   </bbd>

   <bbd id="bbd_0231" addr="1">
    <escon id="escon_104" addr="0" pchid="0×104" />
    ...
    <escon id="escon_107" addr="3" pchid="0×107" />
   </bbd>
   ...
  </fibb>
 </tnt>
</mba>
...
<mba id="mba_05" addr="5">
 <tnt id="tnt_050" addr="0" alt_path="tnt_020" />
</mba>
...
```

Figure 2

Sample I/O configuration and XML representation. A single 16-port Enterprise Systems Connection (ESCON*) card, connected via the redundant I/O interconnect (RII) of the System z9, is described by an XML file. Every major component corresponds to an XML tag with attributes that specify the link structure ("addr" and "alt_path" attributes), an ID, and additional information such as the physical channel ID (PCHID) number.

VISA test cases are available with complex error injection scenarios for Parallel Sysplex channels, running every night in regression.

Dynamic configuration changes

One of the distinguishing features of the System z platforms is the variety of concurrent hardware changes that can be made [12, 13]. Some of these changes are referred to as hot plug, hot unplug, and repair-and-verify of I/O cards and MBA fan-out cards, as well as enhanced book availability (EBA) [13], which allows a single book in a multibook server to be concurrently removed from the system. This allows service personnel to perform a repair or to physically upgrade the hardware on the book. (A processor book contains multiple processor chips, physical memory cards, and multiple I/O hub cards.) In the following, these kinds of actions are generically referred to as hot plug; more specific terms are used when necessary. All of these functionalities are simulated in CECSIM because it allows individual pre-testing of a specific concurrent change to be applied on a customer's system.

Within CECSIM, an I/O model [4] keeps track of the current hardware configuration and simulates the I/O hardware. While hardware could previously be initialized only before the initial microcode load (IML), simulating a hot-plug feature requires the modification of I/O model configurations at any time. To accomplish this, certain redesign steps were required.

The SE sends firmware configuration files to the CEC that inform the processor firmware about the installed hardware components. Previously, these files were also used to initialize the I/O model, but this proved to be particularly impractical in hot-plug scenarios, and a capability was needed for testing the resilience of the firmware in the event of mismatches between the firmware configuration files and the actual configuration. Thus, we decided to remove the firmware configuration file dependency of the I/O model and use another approach.

Several prerequisites had to be fulfilled; for example, new hardware configurations should be easy to define in a graphical or other manner. The mostly treelike topology of the hardware structure, which is determined by the nodes and multiplexers, should have a natural representation. All hardware information should be stored in a single easily maintainable file. Tools should be readily available to manipulate the format and check it for errors.

We chose to describe the configuration in XML format. Certain XML tags were assigned to all of the hardware components that are relevant for the I/O model configuration. Each component was given a unique identifier, and all of the necessary hardware and logical attributes of the components were defined as arguments for the tags. For the rare instances in which the hardware topology is not strictly treelike (for example, for Parallel Sysplex channels and for redundant I/O interconnect), special attributes were defined to describe the non-treelike connections. A sample extract of an I/O configuration and the corresponding XML source are shown in **Figure 2**. In the figure, the elements MBA (memory bus adapter), TNT (Triton-T), FIBB (fast internal bus buffer), BBD (bidirectional bus distributor), and CH (channel) are application-specific integrated circuits (ASICs) of the System z I/O subsystem. (Triton-Ts are part of a new redundant I/O interconnect feature.)

While CECSIM is running, the I/O model can read a modified configuration and perform the necessary updates. The I/O model automatically analyzes the differences between the current configuration and the new one, and adds or removes hardware whenever necessary but leaves the remaining hardware untouched. The restrictions due to the actual hardware packaging are taken into account by controlling scripts that directly modify configurations using commands. In this manner, realistic scenarios such as plugging or unplugging I/O cards can be simulated.

In hot-plug scenarios, various firmware configuration files are exchanged between SE and i390 firmware. To simplify the implementation of hot-plug test cases, CECSIM provides commands to generate these files from the XML descriptions of the configurations before and after hot plug. Knowledge of both configurations is required because the firmware configuration files describe only configuration *changes,* while the XML file always specifies the *complete* hardware configuration [12]. CECSIM can analyze the difference and create the firmware configuration files. Service words transfer the resulting files to the CEC. Programmed or manual intervention in order to inject errors is possible.

When connected to an SE, the firmware configuration files are created on the SE as they would be for an actual machine. The SE generates an updated XML configuration and transmits it. The simulator in turn must process the file upon reception and update the I/O model configuration. Controlling scripts on the SE were implemented to initiate and specify the hot-plug actions.

Test environments

From a system point of view, three major methods exist to trigger a particular functionality of the firmware. The most obvious method uses z/Architecture instructions to test the corresponding firmware implementation, as is done with VISA or small programs. Another method exploits the capabilities of CECSIM scripts to interact with the firmware either on the service word interface or by modifying system internal states to trigger various recovery situations. These first two methods, used by themselves or in combination, allow a high degree of functional test coverage for the firmware even without the third method.

To obtain even more comprehensive test capabilities, a third method allows the dynamic loading and execution of firmware fragments. The test case is essentially a piece of firmware that calls other firmware functions to act as a unit test.

VISA—a new test-case framework

Developers often write small assembly language programs for unit tests. In many cases, these programs are not very flexible, and they must be modified each time to work with another configuration or in another environment. Also, they have no built-in capabilities such as those needed for diagnostic messages and tracing. On the other hand, very complex and sophisticated exercisers exist that can be controlled using many parameters. They are designed for automated regression runs on the real system and provide good diagnostic information for later analysis.

All of these programs run at the z/Architecture level and thus have no access to firmware-specific components in the system. Therefore, they can verify the correctness of the system behavior only at that architecture level. They cannot detect an invalid update to a firmware control block that may lead to subsequent erroneous behavior of the system. Their capability to inject errors is limited, because this would also require access to firmware-internal data areas.

VISA has been developed to close the gap between simple unit test programs and complex exercisers. The main focus of VISA is the CECSIM platform, but it is used on the actual machine as well. When running in the simulation environment, it may utilize interfaces to the simulator in order to access internal information that is not available at the architecture level. A strength of VISA arises from the fact that its test cases can be configured via parameters at runtime. Numerous macros and service routines are available to assist the developer in writing new test cases. VISA and the test cases are written in C, and the VISA environment is structured as shown in **Figure 3**.

VISA kernel

VISA is based on a small operating system kernel with a C runtime library. VISA supports multiple processors (not yet exploited in test cases) and offers, for example, the flexibility required to install private signal handlers in order to track and handle interrupt conditions.

VISA services

VISA services form the VISA application programming interface (API). They make it easy to write test cases. Standard interrupt handlers are included as well as services that compare actual data with predictions. The test-case writer may install other interrupt handlers if needed.

VISA services relieve the test-case writer from various details and at the same time allow the writer to handle details and inject errors wherever needed, even at the lowest levels. This capability was created by offering many levels of service subroutines.

A service is available that allocates storage on a required alignment boundary, clears it, and ensures that this storage is released automatically during a cleanup

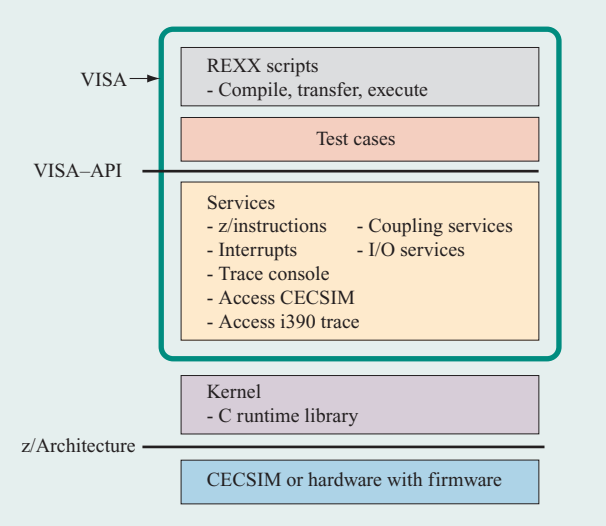


Figure 3

Structure of the VISA environment. The API is the interface between the functionalities represented by the test-case rectangle and the services rectangle. The blue box at the bottom indicates that VISA can be run on the actual system or within CECSIM.

subroutine at the end of the test case. As mentioned, CECSIM services can be invoked by using the SIMCALL instruction to inject errors or inspect firmware internal data when needed.

VISA provides tracing routines. Each trace is associated with a trace level. Whether or not a trace is actually shown is controlled via a global trace level that is specified by the user when executing a test case. A higher trace level results in the display of more data such as progress messages, control block data, or even the VISA-internal addresses of such data. The built-in services provided by VISA generate ample traces to ease the task of the test-case writer. In addition, a test case may invoke the trace function explicitly.

For z/Architecture instructions to be tested, an include file is available that declares the control blocks involved. Also available are a C function with inline assembly language code to generate the instruction, and a higher-level service that traces on the console the progress and the control block data.

The services help the test-case writers to concentrate on the subject of their tests, the firmware code. With VISA, they obtain easily maintainable test cases, output that is easy to read, systematic error messages, and efficient regression-run capability.

VISA test cases

VISA test cases are executed directly on the z/Architecture and not inside any other OS environment.

As mentioned, VISA services are offered for checking, tracing, and for accessing CECSIM, for example, in order to inject errors or analyze firmware internal data. Each step within a VISA test case can perform some or all of the following functions:

- Allocate space for control blocks and enter the data.
- Set up error injection.
- Call the subroutine to execute the instruction and indicate expected condition codes.
- Check the condition code. A service writes traces if the code is unexpected.
- Check the resulting data.

Often, a test-case writer does not want to cope with many details of the architecture but wants to use standard cases and data. This option is offered as well, and the test-case writer can vary certain particular values and create dedicated error scenarios.

VISA scripts

We previously introduced the concept of CECSIM scripts in the sections on error injection and the REXX interface. VISA scripts are a subset of CECSIM (REXX) scripts that typically control the execution of VISA test cases. In this section, we note that VISA scripts are offered to monitor and control test-case execution on CECSIM. The execution of a series of test cases may be stopped when an error occurs. In conjunction with CECSIM and machine features that permit single-stepping of instructions, comprehensive debug capabilities are available.

The scripts that execute VISA test cases use control files. One file contains a matrix of related test cases and one or more applicable types, e.g., I/O channel types. This is required for automated regression runs, but manual execution of a single test case may also make use of this file. Additionally, files are available that contain the prepared configuration-specific data to be transferred when prompted by a test case. When a new I/O configuration data set (IOCDS) for CECSIM is being prepared, effort is required to create these files so that test cases can make use of them. (IOCDS provides a software view of the I/O hardware.)

Each test case begins with an initialization macro. At execution time, the macro establishes the internal data structures, installs default signal handlers, and prompts the user to enter the configuration data needed. On the actual system, or when CECSIM is connected to an SE, this prompt is displayed on the integrated console of the SE as an operator message. On a standalone CECSIM, a script answers this prompt and responds with prepared data.

In addition to the high-level VISA script using control files, other scripts (see the section on CECSIM scripts

that follows) can call a low-level VISA script to start a test case, providing the required parameters and configuration data as an argument string. The return code of a test case is displayed by a message on the console and is also saved in storage. VISA signals the end of a test case to CECSIM via a SIMCALL instruction. This event is recognized by the VISA script, which inspects the return code in storage to decide whether the test case was successful or had failed. For regression runs, depending on the severity of the failure, the script decides whether it continues running (in the case of minor errors), performs a system reset before issuing the next test case, or even performs a re-IML of the simulated system before starting the next test case.

CECSIM scripts

In this section, we reemphasize and consolidate important information related to CECSIM scripts as they relate to VISA test cases, channel configuration, and partition management. As mentioned earlier, an example of a CECSIM test is a script that can configure or deconfigure a channel by starting a VISA test case. This can be accomplished for a single channel or a group of channels, for a single partition or for all applicable partitions. A list of channels may be given for configuration and deconfiguration. To determine all configured channels, we may first deconfigure them and then configure them again for all applicable partitions. An option is also available to deconfigure a channel, run a given VISA test case, and then configure it again.

We developed CECSIM scripts to initialize the simulated system in logical partition (LPAR) mode and to activate partitions. This functionality is usually provided by the SE. In CECSIM, standalone scripts send service words carrying the appropriate command to the CEC. The basic input parameters for the partition activation include the number of CPUs that should belong to the partition, the amount of central storage, and the amount of expanded storage.

A script can be used to send several operator commands to the CEC, and a program can be loaded and executed in a partition. Thus, the partition handling can be tested without requiring an SE. Other operator commands can configure an adjunct processor when it is necessary for testing the IBM System z cryptographic functions.

ITEM test cases

In order to force i390 code into specific states and to create error situations, the i390 test-case execution monitor (ITEM) is available. ITEM allows the loading and linking of new functions to an existing and running i390 code load. After linking, the new functions can be executed. Figure 1 placed ITEM tests in perspective with other topics discussed in this paper. The ITEM test cases can be run in the CECSIM standalone environment, in combination with the SE, or on the real machine. The user interface on CECSIM is a script.

The ITEM load function on CECSIM sends the ITEM test cases to the i390 ITEM loader function of the executable and linking format (ELF) loader [14]. The ITEM loader links the test cases to the resident code load. Thus, the ITEM test cases can use all procedures and services of the permanent i390 code load, such as i390 tracing, logging, remote procedure call services, and storage management.

The dynamically added functions can be called by using service words. It is also possible to change variable values, call any routine, or just change a single bit. Particular i390 code paths can be executed by performing these kinds of operations. It is also possible to pass arguments to the ITEM test case in order to control the execution. Depending on the arguments, the execution of different code paths can be triggered. The execution can take place on any processor in the system. The ITEM test-case results are sent via service words to the SE. In CECSIM standalone modes, scripts can analyze and handle these results.

Regression runs

During firmware development, it is not sufficient to test only new or changed functionality. Rather, we must verify that code changes and new functions do not damage any existing functions. A good way to detect unexpected side effects as early as possible is to run a regression test package every day. The i390 regression test does not replace other tests; rather, it is a tool for quality improvement, and it reduces the test effort on actual machines and thus reduces development costs.

By using VISA, script, or ITEM test cases, all tests are developed so that they can be executed in regression. Normally, online users start CECSIM using a full-screen panel. In addition, we provide a command-line interface that is used for automatic regression runs.

Automation is necessary to ensure that regression is performed regularly. Every night, a series of CECSIM runs is started (**Figure 4**). The results are written into files such as overview files that contain the result summaries from all CECSIM runs. Additionally, during each run, other result files are created, and these files are referred to as console files, firmware trace files, log files, or specific test-case output files.

These files are automatically checked for irregularities, such as error messages on the console or missing or unexpected system reference codes in the log file, so that many kinds of errors are found in the summary file. As specified in the control files, some of these files are always

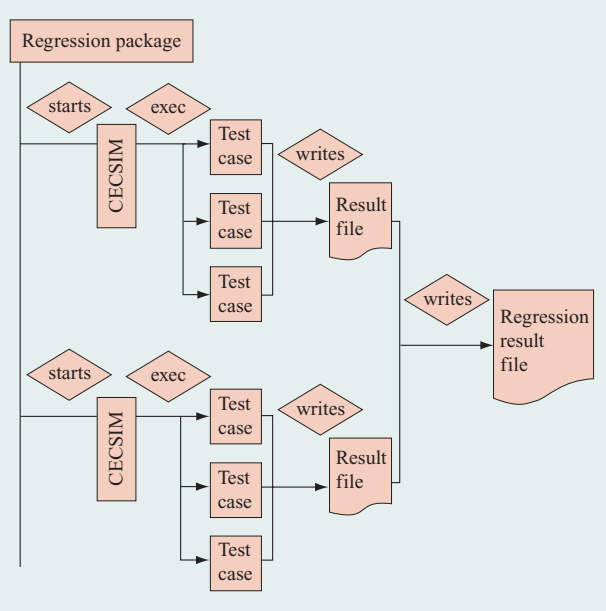


Figure 4

Regression tests. The regression package starts CECSIM several times during a complete regression run. After CECSIM has been started, the regression software executes several test cases. Each test case writes its own detailed result file, and also writes a single line, denoting success or failure, into the regression result file depicted at the right of the figure.

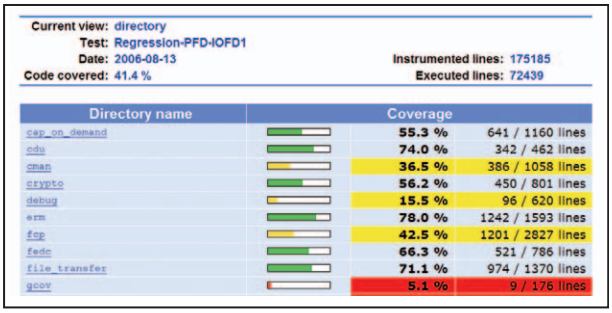


Figure 5

Coverage measurement — sample part of a directory view.

retained in a library for evaluation, while others are retained only when an error is detected.

Various precautions are taken to prevent problems that might hamper the progress of the regression runs. As previously mentioned, if regression runs are unable to proceed (i.e., they "hang"), system reset (re-IML) may take place, or a single CECSIM run can be terminated to permit regression to continue with the next CECSIM run.

Coverage

Path coverage measurement determined which i390 code paths were tested by the regression package. By using the measurement results, components that were not covered were found, and new test cases were written for these and integrated into the regression package. Existing test cases were enhanced to cover more code. In some cases, the results even indicated the presence of unused code, which could be removed.

To be able to measure code path coverage, the i390 code load must be instrumented by the compiler. Global counters are inserted at all branch points. The i390 loader ensures that these counters are allocated for each processor unit (PU) in the system. Thus, it is even possible to determine which PU executed a particular piece of code. Normally, an instrumented executable, such as a Linux** executable, writes the counters into a file at the end of execution. Because i390 never really ends its execution, the counters must be extracted from the running system. To achieve this, i390 storage can be accessed directly through CECSIM facilities, or service words can be used to send the counters. The first method is relatively fast, while the latter is slower but can be also be used on the actual machine.

After a regression run, the raw coverage data (i.e., the counter values) are processed by the LCOV software tool. LCOV is based on the GNU coverage tool GCOV [6], which provides information about which parts of a program are actually executed (i.e., "covered") while running a particular test case. LCOV is able to generate HTML pages that show the results of the measurements in a directory view, a file view, and a source-code view. [The term GNU (GNU's Not UNIX**) is a recursive acronym that refers to a UNIX-like development effort of the Free Software Foundation.]

The generated HTML pages are published on an Intranet web server so that each i390 developer can examine the coverage of his components. **Figure 5** shows a sample extract from a coverage measurement. Each line represents a source-code directory and its path coverage data. The developer can click on the directory name to obtain the data for each individual source file within the directory. A click on the file name opens the source code itself with the counters beside each line. A configurable color code is available that indicates the degree of path coverage. For example, 0–15% may be represented as red, 15–50% as yellow, and 50–100% as green.

Because coverage measurement is currently still semiautomatic, we are working to fully automate the process and run it weekly. However, each developer can also measure coverage as desired by using only a specific set of test cases to verify a code change. Our ultimate goal is to provide a history of coverage data to ensure that the

amount of code covered by the regression package is continuously increasing.

Outlook

The recent enhancements for the CECSIM platform discussed in this paper clearly indicate a trend toward increased capabilities for firmware test that include better unit test capabilities, higher code quality through regression tests, and additional means for complex function testing. The next stage of CECSIM development will include the ability to attach VHDL (hardware description language) simulators running certain I/O hardware parts in order to allow early and more precise firmware testing with realistic hardware behavior. This will not reduce the need for simple and fast I/O models such as the ones used today.

Further enhancements will integrate other firmware components such as the I/O channel code, and our use of a remote GNU debugger for i390 will enhance our general debugging ability. Although many capabilities are currently established, many improvements of CECSIM may be developed in order to create an even more comprehensive system integration platform.

References

1. P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer* **35**, No. 2, 50–58 (2002).
2. A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood, "Simulating a $2M Commercial Server on a $2K PC," *IEEE Computer* **36**, No. 2, 50–57 (2003).
3. J. Engblom, B. Werner, and G. Girard, "Testing Embedded Software Using Simulated Hardware," *Proceedings of the Conference on Embedded Real-Time Software (ERTS 2006)*, Toulouse, France, January 2006; see *http://www.engbloms.se/jakob_publications.html*.
4. J. von Buttlar, H. Böhm, R. Ernst, A. Horsch, A. Kohler, H. Schein, M. Stetter, and K. Theurich, "z/CECSIM: An Efficient and Comprehensive Microcode Simulator for the IBM eServer* z900," *IBM J. Res. & Dev.* **46**, No. 4/5, 607–615 (2002).
5. M. Stetter, J. von Buttlar, P. T. Chan, D. Decker, H. Elfering, P. M. Gioquindo, T. Hess, et al., "IBM eServer z990 Improvements in Firmware Simulation," *IBM J. Res. & Dev.* **48**, No. 3/4, 583–594 (2004).
6. P. Larson, N. Hinds, R. Ravindran, and H. Franke, "Improving the Linux Test Project with Kernel Code Coverage Analysis," *Proceedings of the 2003 Ottawa Linux Symposium*, Ottawa, Canada, July 2003, pp. 260–275.
7. L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries* Processor," *IBM J. Res. & Dev.* **48**, No. 3/4, 425–434 (2004).
8. J. Maergner and H. R. Schwermer, "High Level Microprogramming in I370," *The Design of a Microprocessor*, W. G. Spruth et al., Editors, Springer, New York, 1989, pp. 303–316.
9. W. Gellerich, T. Hendel, R. Land, H. Lehmann, M. Mueller, P. H. Oden, and H. Penner, "The GNU 64-Bit PL8 Compiler: Toward an Open Standard Environment for Firmware Development," *IBM J. Res. & Dev.* **48**, No. 3/4, 543–556 (2004).
10. I. G. Siegel, B. A. Glendening, and J. P. Kubala, "Logical Partition Mode Physical Resource Management on the IBM eServer z990," *IBM J. Res. & Dev.* **48**, No. 3/4, 535–541 (2004).
11. "Programming Language—REXX, ANSI Standard X3.274-1996 (printed copies can be ordered from the American National Standards Institute store); see *http://webstore.ansi.org/ansidocstore/find.asp?*.
12. G. Mayer, G. Doettling, R. F. Rizzolo, C. J. Berry, S. M. Carey, C. M. Carney, J. Keinert, et al., "Design Methods for Attaining IBM System z9 Processor Cycle-Time Goals," *IBM J. Res. & Dev.* **51**, No. 1/2, 19–35 (2007, this issue).
13. IBM Corporation, "IBM System z9 Enterprise Class Technical Guide"; see *http://www.redbooks.ibm.com/redbooks/pdfs/sg247124.pdf*.
14. C. Axnix, T. Hendel, M. Mueller, A. Nuñez Mencias, H. Penner, and S. Usenbinz, "Open-Standard Development Environment for IBM System z9 Host Firmware," *IBM J. Res. & Dev.* **51**, No. 1/2, 195–205 (2007, this issue).

Received April 13, 2006; accepted for publication April 26, 2006; Internet publication February 6, 2007

Klaus Theurich *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (theurich@de.ibm.com).* Mr. Theurich studied electrical engineering at the Fachhochschule Esslingen, graduating in 1987. He joined the IBM development laboratories in Boeblingen that same year to work on S/370 system testing and development for the Parallel Processing Compute Server and on hardware development for the first intersystem channel. In 1994, he joined the S/390 I/O microcode development effort. During an international assignment (1998–2000) in Poughkeepsie, New York, he worked on a new concept for simulation of coupling firmware. After time spent on various System z firmware responsibilities, Mr. Theurich is currently the team leader for CECSIM; he has the global responsibility for the System z firmware simulation environment.

Alexander Albus *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (aalbus@de.ibm.com).* Dr. Albus received an M.S. degree in physics from Oregon State University and a Ph.D. degree in physics from the University of Potsdam, Germany. His research interests in physics have included theoretical solid-state physics and the quantum theory of Bose–Einstein condensates. After a year in the automotive engineering industry, in 2004 Dr. Albus joined the IBM development laboratories in Boeblingen, Germany. He is currently developing the I/O device driver part of i390 firmware for the System z and parts of the CECSIM I/O model, with a focus on MBA models and the dynamic configuration change component.

Felix Eickhoff *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (eickhof@de.ibm.com).* In 2002, Mr. Eickhoff received an M.S. degree in computer science from the University of Connecticut and an M.S. degree in software engineering from the University of Stuttgart, Germany, in 2004. That same year he joined the IBM development laboratories in Boeblingen, Germany. He is currently working on System z processor firmware development for the I/O-hot-plug feature.

Daniela Immel *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (immel1@de.ibm.com).* Mrs. Immel is currently a developer of System z processor firmware. In 2004, she received her B.S. degree in business information systems from the University of Applied Sciences Karlsruhe. She joined IBM in 2004, working in System z processor firmware development.

Andreas Kohler *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (akohler@de.ibm.com).* Dr. Kohler received Dipl.-Phys. and Ph.D. degrees in physics from the University of Stuttgart, Germany, in 1993 and 1999. He joined the IBM development laboratories in Boeblingen, Germany, in 1999. His current responsibilities in System z I/O firmware development include test tools, simulation, and error-recovery code.

Eberhard Lange *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (elange@de.ibm.com).* In 1981, Mr. Lange received an M.S. degree in electronics (Dipl.-Ing.) from the University of Stuttgart. He started as an electronics engineer in the IBM Boeblingen development laboratories. For the first four years, he developed S/370* processor microcode and software for floating-point calculations, and he spent another four years working on microcode for S/370 communication adapters. For the next seven years, he was a marketing expert for the S/370 operating system VSE/ESA*. In 1996, he joined the International Technical Support Organization, with worldwide responsibility for the support of the relational database DB2* on VSE and VM. In 1998 Mr. Lange returned to S/390* processor development, responsible for microcode communications between processors within a Parallel Sysplex. He is currently responsible for simulation to ensure firmware quality.

Joachim von Buttlar *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (joachim_von_buttlar@de.ibm.com).* Mr. von Buttlar is an Advisory Engineer in the System z9 Firmware Development group. He received an M.S. degree in computer science (Dipl.-Inform.) from the Technical University of Berlin in 1983. In 1984, he joined the IBM development laboratories in Boeblingen, Germany, to work on microcode development for the IBM 3092, 9221, and 9672 systems. From 1990 to 1991, he worked as liaison engineer on international assignment in Endicott, New York. In 1997 he initiated the CECSIM project, developed its concepts, and implemented the simulator kernel. Mr. von Buttlar received an IBM Corporate Award in 2002. His current responsibilities include the development of CECSIM for future System z platforms as well as I/O subsystem design.