RISE++: A Symbolic Environment for Scan-Based Testing

STEVE VINOSKI
Hewlett-Packard Company

Unfortunately, though scan test techniques achieve high levels of fault detection and isolation, few environments provide the facilities necessary for easily developing and debugging scan-based tests. The author describes the architecture and implementation of an object-oriented software system that makes use of distributed-computing techniques. This powerful, flexible system supports the development, application, and debugging of scan-based tests for the HP/Apollo Series 10000 RISC workstation and can be used with other scan architectures.

SCAN TEST TECHNIQUES provide hardware access and control that is superior to traditional functional test strategies. Though sequential circuits often contain nodes that are difficult or impossible to access with functional tests, the reduction of these circuits into the combinatorial circuits required by scan testing allows access to all such nodes.

Scan test techniques result in a serialized view of circuit nodes because the nodes are chained together into shift registers called scan rings. Each individual bit in a scan ring provides control and access for a particular circuit element. Regardless of its regular functionality, we can generally test a circuit with scan by shifting in a control state, applying functional clocks, and then shifting out the new state and comparing it with the expected bit values. In effect, this serialized view abstracts the functionality of the circuit for testing purposes.

Unfortunately, this same access and control afforded by scan test techniques can also be the source of testing problems. A failing scan test means that one or more scan bits contain unexpected values. Isolating the cause of the failure can be a daunting task when hundreds or thousands of bits are involved in a single test. Keeping track of the function of each scan bit and any relationships it has to other bits in the scan ring can be nearly impossible, especially when performing board-level tests involving multiple scannable chips. Scan techniques may abstract the functionality of the hardware under test, but in return they greatly increase the amount of information that must be managed during debugging.

The software driving typical test systems is usually designed to ensure rapid and accurate production testing, not to aid with debugging. Any debugging features provided by such software normally facilitate only general control of the test hardware. Thus, test program developers must rely on simulation environments and custom test harnesses to debug their software, hoping that their tools are accurate enough to minimize the effort required to get their programs running correctly on the actual test system. Unfortunately, even the best simulations cannot predict every problem that will be encountered during actual testing procedures.

By managing the observation and

control afforded by scan testing techniques, the Remote Interactive Scan Environment (RISE++) provides a powerful development and debugging facility for scan-based testing of the HP/Apollo Series 10000 workstation. RISE++ executes on a separate workstation and communicates with the system under test via a local area network. Test engineers can easily develop and debug scan tests under RISE++ because it tracks many trivial but important details for them and provides them with precise control over the test hardware. RISE++ effectively raises the level of abstraction and allows test engineers to concentrate on the problem at hand.

DN10000 scan architecture

The scan subsystem of the HP/Apollo Series 10000 workstation[1] includes the scannable hardware and the software that drives it. Users can plug various boards, including CPU boards, memory subsystems, and graphics boards, each populated with a number of scannable chips, into one of eight slots in the main system bus (called the X-Bus) of the DN10000 workstation. Each board also contains one scan and clock resource (SCR) chip. This chip allows control of up to eight separate scan path ports, with up to one internal and one boundary or external scan ring connected to each port. By convention, no more than four scannable chips reside on any scan path.

Software executing on the service processor (SP) of the machine performs all scan operations. The service processor is a Motorola 68020 used for booting and testing the system. Each SCR connects to a systemwide diagnostics bus (DBUS) that the service processor accesses via a three-register interface called the diagnostics bus interface (DBI). Through the DBI, the service processor can address any individual SCR, or it can broadcast commands to all SCR chips in the system. Figure 1 illustrates the Series 10000 scan hardware architecture.

In addition to implementing scan control functions, each SCR distributes and controls the clocks for the board it resides on. The SCR allows clocks to be started, stopped, burst, and pulsed for each scan path independently.

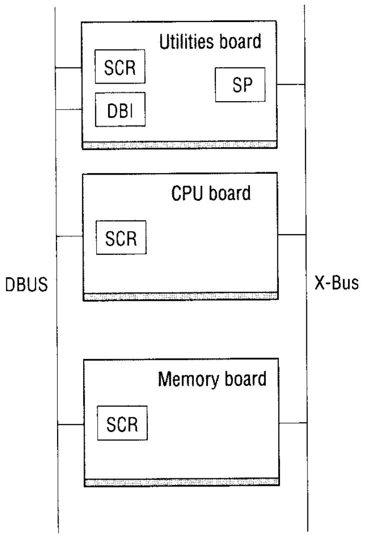


Figure 1. HP/Apollo Series 10000 scan hardware architecture.

The scan subsystem driver software allows access to SCR operations through a hierarchical hardware access model for its clients. The model reflects the hierarchy present in the hardware design:

- a system contains scannable boards
- a board contains scan paths
- a path contains scannable chips
- a chip contains scan rings
- a ring contains scan bits

To use this model, application software specifies access to any scannable component of the hardware via scan addresses whose values reflect the place of the component in the hierarchy. The hierarchical view of scannable hardware provided by controllers like the SCR has also proven effective for board-level built-in self-test.[2] Figure 2 graphically depicts this scan hardware hierarchy.

One important aspect of the hierarchical hardware access model is that most scan-in and scan-out operations appear to occur at the chip level, regardless of the number of chips chained together on each scan path. This distinction is very important; without it, some scan vector data would be needlessly dependent upon board topology. Chip-level scan I/O permits the reuse of chip test vectors for board-level testing.

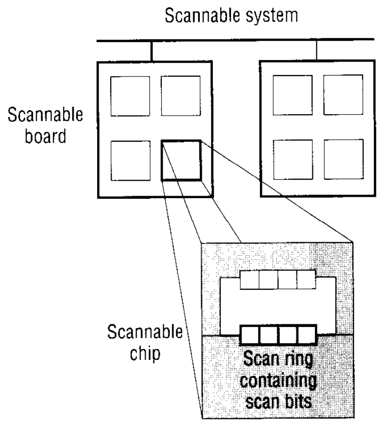


Figure 2. Scan hardware hierarchy.

All in all, the hierarchical hardware access model simplifies the amount of state information the scan subsystem driver software must internally maintain. It provides a clean, powerful abstraction of the hardware domain to applications such as RISE++.

RISE++ architecture

We can accurately describe RISE++ as a symbolic hardware debugger because it provides the ability to symbolically manipulate each component of the hierarchical hardware access model. In other words, each scannable entity in the system, including boards, chips, and scan bits, can be accessed and controlled by name. This aspect of RISE++ is analogous to the symbolic access that software debuggers provide for programming language variables and functions.

RISE++ is well integrated with the Unix environment. It follows the Unix philosophy of providing simple tools that can be combined to create other specialized,

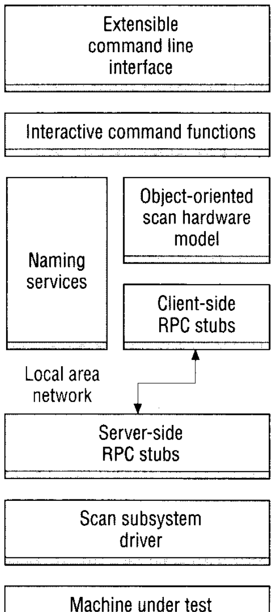


Figure 3. *Architectural layers of RISE++.*

more powerful tools as the need arises. The alternative of attempting to implement RISE++ as a fixed set of debugging tools would almost certainly have failed, since it would have been impossible to foresee all the circumstances in which RISE++ would prove useful.

Similar to most software systems, RISE++ is composed of several layers of functionality:

- extensible command line interface
- interactive command functions
- naming service for scannable entities
- object-oriented scan hardware model
- remote procedure call (RPC) software

Figure 3 graphically depicts the relationships between these layers.

Extensible command line interface. Via the command line interface, users can issue RISE++ commands and extend RISE++ to fit their particular test requirements. Ousterhout has observed that "a general-purpose programmable command language amplifies the power of a tool by allowing users to write programs in the command language in order to extend the tool's built-in facilities."[3] The Tool Command Language (Tcl) developed by Ousterhout provides the basis for the RISE++ extensible command line interface. Tcl is composed of a library of C functions that provides

- command language parsing
- a set of built-in commands
- functions enabling extension of the built-in command set

The Tcl built-in commands provide programmability for the RISE++ command line interface. Commands such as Eval, For, Foreach, and If provide programmable control constructs. Other commands such as Proc and Set allow for the creation and manipulation of command procedures and variables. Other commands enable users to manipulate mathematical expressions, lists, strings, or files. Still others provide ways for users to interact with the underlying operating system.

Tcl command arguments, variable values, and command return values are all ASCII strings. Each command interprets its arguments as it wishes. For example, the Expr command expects its arguments to be strings of numbers and mathematical operators. All string return values from command functions are automatically displayed to the user. Unix tools such as ls and grep can also be invoked from the command line, just as they are called from a Unix shell. Users do not have to reinvent the functions provided by common Unix tools.

RISE++ extends the built-in command set of Tcl with commands specific to the scan-testing problem domain. These include commands to

- manipulate scan vector data
- access SCR and DBI registers
- access files containing test vectors
- interact with the scannable entity-naming service
- determine the configuration of the hardware
- address different X-Bus slots
- perform operations on scannable entities
- control functional clocks

The RISE++ command line interface provides 149 commands in all, including the built-in commands of Tcl. Extensive documentation for each RISE++ command is readily available from the command line via the Unix Man command.

Users can extend the built-in command set by writing their own command scripts, called procs. Like regular Unix commands, user-defined procs can take arguments and return values. Procs can call built-in commands or other procs as required. Once defined, a command implemented by a Proc is indistinguishable from a built-in command. Examples of procs and built-in commands appear later.

By providing a straightforward programmable interface to the scan hardware, RISE++ extends the arena of test development to hardware design engineers and manufacturing personnel who do not necessarily possess the training required to create test software. The simplicity of the RISE++ command interface allows it to serve as a common language that hardware designers, test engineers, and manufacturing personnel can all use to communicate hardware problems and share test solutions.

Naming service. The Processor Controller microcode for the IBM 3081 provided access to machine registers via scan by translating symbolic register names to the physical scan addresses of the bits comprising them.[4] The associations of the symbolic names and the physical scannable entities they represented were stored on disk in translation tables. Under RISE++, such groups of logically related scan bits are called scan buses. The scan operations that examine and change the bits making up the register are completely transparent to the user. The physical locations of the scan bits making up a scan bus do not matter; they need not be contiguous nor even reside on the same chip.

Symbolic manipulation of chains of scan bits requires

- a descriptive identifier for each of the scan bits
- knowledge of the ordering of the scan bits within the scan rings
- knowledge of the number of scan bits in each scan ring

Without this information, software like RISE++ cannot properly format scan vector data nor allow accurate access and control of individual scan bits.

A separate tool called the vector configuration file compiler (VCFC) translates a textual vector configuration file (VCF) containing names for boards, chips, scan bits, and scan buses into an associative database based on Unix ndbm files.[5] Some of the names present in a vector configuration file come directly from the hardware design netlists, while many of them are created by users for customization purposes.

Each entry in the databases contains an association of a name character string with information about the scannable entity it represents. To prevent name clashes on boards with multiple instances of the same chip, the compiler treats each chip declaration in the vector configuration file as a different naming scope. It requires all scan bit and scan bus names to be prefixed by the name of the chip containing them, as in chip_name.bit_name.

Because the compiler program is a separate compilation tool, RISE++ also allows names to be defined from its command line at runtime. Names defined via RISE++ are not added to the name database files, however, to prevent them from becoming filled with information that was only relevant for one debugging session. Using VCFC to compile a vector configuration file is the only way to create or modify scan name databases.

The use of the Unix ndbm facility for scan name databases has several advantages. The most important advantage is that scannable entity-naming information can be located rapidly; the ndbm facility normally requires only one or two disk accesses to locate an item.[5] Another advantage is that names are dynamically loaded as they are required. Without this ability, RISE++ would have to load all names when it started up, resulting in slow start-up times and unacceptable memory requirements. Since nearly 50,000 names are used to describe all the scannable entities present in a fully loaded Series 10000 system, dynamic loading allows each name to be loaded from the databases as it is referenced. This approach minimizes memory usage and spreads name loading time out over the life of the RISE++ process.

Object-oriented scan hardware model. A large portion of RISE++ was designed and implemented using object-oriented software development techniques. These techniques focus on the entities that exist in a problem domain and the operations that can be performed upon them. Thus, an object-oriented view of a system differs from an algorithmic view because the latter merely "highlights the ordering of events."[6]

The agents of the scan-testing problem domain that are modeled by the object-oriented RISE++ software include scan vectors, scan bits, scan rings, scannable chips, and scannable boards. The hierarchical nature of the hardware domain is directly reflected in the construction of the object model: Boards are composed of chips that in turn are composed of scan rings composed of scan bits. These objects are accessed via abstract address objects, conceptually similar to the methods by which they are physically accessed in the hardware domain.

```
rise> size 4
Slot 4:cpu, board type 5, revision 2
        path 0/device 0 ip, revision 4
        path 1/device 0 mmu, revision 6
        path 2/device 0 cba, revision 9
        path 3/device 0 cbd0, revision 2
        path 3/device 1 cbd1, revision 2
        path 4/device 0 fpc, revision 0
        path 5/device 0 frfl, revision 1
        path 5/device 1 frfu, revision 1
        path 6/device 0 alu, revision 0
        path 6/device 1 mul, revision 0
        path 7/device 0 amd29818,
          revision 0
```

***Figure 4.** Size command example.*

When RISE++ is invoked, it sizes the system under test and creates a hierarchy of software objects that models the scannable hardware to be tested. The example in Figure 4 shows a result of issuing the Size command for X-Bus slot 4 (the "rise>" text is the command prompt).

Once the RISE++ process constructs this internal software model, each scan-related command is ultimately performed by the software object representing the hardware entity being acted upon.

The object-oriented software used to model the hardware under test is written in the C++ programming language.[7] C++ was chosen because of its efficiency, strong type checking, and facilities for encapsulation, dynamic binding, and inheritance. The importance of these features to the software engineering aspects of RISE++ cannot be overemphasized. In fact, quite likely the development of RISE++ would have failed had it not been for the use of C++.

Remote procedure call software. While others have described remote procedure calls (RPCs),[8,9] I briefly explain their use in RISE++ to show the benefits of remote testing. Each call to a function in the driver software from the rest of RISE++ is transparently converted into a remote procedure call that is sent across a local area network to the system under test. A small remote scan server executing on the system under test processes the request, invokes the desired driver function, then packages up any return values and sends them back across the network to the caller. The RISE++ software that makes calls to the driver is completely unaware that such calls execute on another machine.

A major benefit provided by the use of remote procedure calls is that very few resources are required of the system under test. The scan remote procedure call server (ScanRS) program that services remote requests for scan operations requires only 20Kbytes of RAM space within service processor memory. Thus, if the service processor correctly executes from its memory and successfully communicates with the local area network hardware, the machine can be tested and debugged using RISE++ from a remote location.

Another advantage afforded by the use of remote procedure calls is that RISE++ executes on top of a Unix operating system on the remote client machine. Thus it can provide a much easier-to-use environment than if it had to function alone on the system under test. A stand-alone program cannot rely upon the features of an operating system to provide services such as file I/O, command line editing, and process management. These features are usually difficult to develop in a stand-alone environment, so most test facilities do not provide them. Because it executes remotely on a client workstation, RISE++ fits directly into the productive computing environment offered by Unix.

A third benefit of the employment of remote procedure call techniques comes from the remote aspect itself. Several systems can be accessed from one client workstation running RISE++. The user can easily control and observe testing of multiple machines simultaneously. This particular use of remote procedure call techniques enables research and development engineers to help with difficult testing problems in the manufacturing plant without having to leave their offices.

The use of remote procedure calls contributes very little to the time required to perform most remote scan operations. Measurements indicate that RISE++ can perform scan I/O operations at a level that is 50% of the throughput achieved by programs running directly on the service processor of the system under test. This efficiency level has proven more than adequate for interactive test debugging and development.

Test examples

The various services provided by RISE++ allow users to develop many different types of tests. Simple tests generally verify the integrity of scan hardware such as the SCR and the scan rings, while the most complex tests can isolate faults within cache RAM.

Scan ring I/O. The ability to read and write the contents of scan rings is one of the most important requirements for RISE++. Sread and Swrite commands perform these operations. The following example shows typical uses of these commands. Here, the AMD29818 device on CPU scan path 7 is being read and written:

```
rise> sread amd29818
{\
D1\
}
rise> swrite amd29818 D0
rise> sread amd29818
{\
D0\
}
```

The first Sread command displays the contents of the 8-bit AMD29818 scan path as the hexadecimal value D1. Then the Swrite command modifies the value to D0, and the second Sread command verifies that the value of the scan ring did indeed change. Values for the scan bits making up a ring are always displayed in the same order that they are scanned out of the hardware, with leading zeros always being displayed. Up to three extraneous bits may appear on the end of the displayed vector value due to the hexadecimal output format.

The curly brace and backslash characters surrounding the displayed scan ring data values allow the output of Sread to be assigned to variables and used directly as input for other commands. They also help readability by allowing long scan vectors to be split across multiple lines. Using Sread vectors as input to other commands corresponds to their usual handling, since they are typically compared against known good vectors or are slightly modified by programming and then scanned back in. The next example, in which the scan ring of one chip is written with the data read out of a different chip, shows the typical treatment of full Sread vectors:

```
rise> swrite cbd0 [sread cbd1]
```

The square brackets cause the Sread command to be performed and its return value to be substituted for it on the command line as the value argument to Swrite.

Scan buses can be accessed in a very similar manner, but the compiled vector configuration file must be loaded first via the Names command:

```
rise> names compiled_mem_names
```

Here, compiled_mem_names corresponds to the Unix pathname of the directory holding the vector configuration file name databases for the memory board. Once the names are available, a scan bus access can be done; see Figure 5.

First, the entire scan ring for the chip

```
rise> swrite mmc0 0
rise> swrite mmc0.cb_addr 55
rise> sread mmc0.cd_addr
0x55
rise> sread mmc0
{\
000000000000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000015400\
}
```

Figure 5. *Scan bus access.*

called mmc0 is set to zeros. Then, a 7-bit scan bus called mmc0.cb_addr is set to the hexadecimal value 55 (the leftmost bit of the value is not used). It then is read back to verify its new value, and the entire mmc0 scan ring is read out. As expected, the example output shows most bits as zero except for those corresponding to the mmc0.cb_addr scan bus. Unlike scan ring data, scan bus values are displayed in regular MSB-to-LSB fashion with no leading zeros. Up to three extraneous leading bits may appear in the displayed value due to the hexadecimal output format.

The scan bus example hints at the power of the symbolic access provided by RISE++. Attempting to set the value of the mmc0.cb_addr register by setting individual bits in the scan ring would be extremely error-prone and time-consuming, as would attempting to read its value by piecing together the values of the bits. With RISE++, the user can concentrate on the logical state of the hardware and let the environment handle all the necessary translations.

```
Proc dump_gprs {} {
    stop                ;# stop functional clocks
    dip_cmd 0x5b00      ;# disallow GPR writes
    scr config 0x4000   ;# enable scanning of IPU
    loop reg 0 32 {     ;# loop over 32 registers
        swrite ipu 0    ;# set IPU vector to all zeros
        # set up read address and scan it in
        swrite -s ipu.gpr_addr [hexpr $reg]
        # pulse IPU functional clock to capture GPR data
        dbi_cmd pulse
        # read out captured GPR value
        set value [sread -s ipu.gpr_val]
        print [format "IP REGISTER : %02d %08X\n" $reg $value]
    }
}
```

Figure 6. *User-defined Proc example.*

CPU register access. The example code in Figure 6 shows a user-defined Proc. This Proc can display the contents of the general-purpose registers (GPRs) of the integer processor unit (IPU) of the Series 10000 CPU-TX board.

The Stop command stops the functional clocks so that scan operations may be performed. The Dip_cmd and Scr commands configure various testability registers within the integer processor unit and SCR chips to prepare for general-purpose register access. Next, the loop construct performs 32 iterations of the sequence of actions required to read a general-purpose register. The iterations allow all 32 integer processor unit general-purpose registers to be read. In that sequence, the first Swrite sets all bits of the integer processor unit scan ring to 0.

The second Swrite sets the value of a scan bus within the integer processor unit to the address of the general-purpose register to be read, then scans the entire integer processor unit vector into the hardware. After Dbi_cmd pulses the functional clocks, Sread scans out the value of an integer processor unit scan bus containing the value of the general-purpose register. The Set command stores it into the value variable. Finally, the Print command displays the register number and its value to the user.

The example in Figure 6 shows the use of RISE++ abstractions together with direct hardware access functions. One abstraction is the use of the scan buses ipu.gpr_addr and ipu.gpr_val to symbolically access the general-purpose register addresses and values. The Dbi_cmd and Scr commands provide direct access to the DBI and SCR registers. The loop construct provides iteration, and the iteration variable reg is also used to set up the address of the general-purpose register to be read. (The Hexpr command merely returns the value of reg in hexadecimal format.)

As shown in Figure 7, users may invoke the dump_gprs procedure by typing its name at the RISE++ command

```
rise> dump_gprs
IP REGISTER : 00 00000001
IP REGISTER : 01 00003F71
IP REGISTER : 02 0007EE31
IP REGISTER : 03 00007EE3
IP REGISTER : 04 00000001
IP REGISTER : 05 00800093
IP REGISTER : 06 0000000D
IP REGISTER : 07 0000001F
IP REGISTER : 08 0000475E
IP REGISTER : 09 00000000
IP REGISTER : 10 01000127
IP REGISTER : 11 0000000F
IP REGISTER : 12 0000001F
IP REGISTER : 13 00000007
IP REGISTER : 14 0000000F
IP REGISTER : 15 00000007
IP REGISTER : 16 0000007F
IP REGISTER : 17 00003F71
IP REGISTER : 18 00000000
IP REGISTER : 19 00000000
IP REGISTER : 20 7FFFFFFF
IP REGISTER : 21 00800093
IP REGISTER : 22 024D81BB
IP REGISTER : 23 0000001F
IP REGISTER : 24 0000003F
IP REGISTER : 25 0003F718
IP REGISTER : 26 00000000
IP REGISTER : 27 0000000F
IP REGISTER : 28 000FF184
IP REGISTER : 29 000FF188
IP REGISTER : 30 0000000F
IP REGISTER : 31 00000000
```

prompt. They do not have to be aware of the sequence of scan operations and clock pulses that dump_gprs performs to read the general-purpose registers.

One important aspect of the dump_gprs procedure is the use of scan techniques to access nonscannable logic. Access to nonscannable hardware is similar to traditional scan-testing techniques. Both require a setup of logic inputs, followed by an application of system clocks, followed by an examination of logic outputs. Scan-in and scan-out operations control logic input and observe logic output. Multiple clock cycles are usually needed to propagate the state of the nonscannable logic to a point where it can be observed via scan. Without the scan buses provided by RISE++, access to the scan bits that allow control and observation of nonscannable logic would be difficult, since these bits are usually spread out among several chips on a board. The scan buses allow the logic to be considered at a functional level even though it is actually being accessed via scan operations.

The brevity of the dump_gprs script is also notable. A C language program that duplicates the functionality of the script required approximately 1,000 lines of source code, yet the Tcl script is only 13 lines long. Much of the compactness of the dump_gprs script results from the use of scan buses. The values of many scan bits can be set with one RISE++ command, while each bit must be set separately in the C language program. The dump_gprs script shows both the power of expression in the Tcl language and its simplicity.

Cache RAM testing. While the dump_gprs script is relatively simple, a more complex RISE++ test procedure provides fault detection and isolation for the cache RAMs on the CPU-TX board. This facility is composed of 10 user-defined procs consisting of approximately 1,000 lines of Tcl code. We developed it over the course of several weeks during the laboratory debugging of the hardware prototype of the CPU-TX board. Development proceeded from the bottom up; that is, we wrote the simpler procs first and then refined and combined them into larger procs as the need arose.

One notable aspect of the cache RAM RISE++ procs is that they strongly resemble the code used to simulate them. Once the cache test algorithms were verified on the logic simulator, we used the RISE++ naming facility to create scan buses that provide the same logical view of the hardware as presented by the simulator. The test code was thus easily and directly translated from the logical simulator environment to RISE++.

Testability features designed into the CPU-TX hardware simplify the testing of the cache RAMs, but the testing process still requires a large number of scan bits to be controlled and observed. Without the naming services and debugging facilities of RISE++, the development of the cache tests would have required several person-months rather than several person-weeks to complete.

Large RISE++ tests. Reilly et al. hint at performance problems caused by the use of scannable entity name translation for critical functions.[4] They developed a compiler to solve their problems. It performed the name translation process and produced data for entire scan rings, allowing for the elimination of the translation overhead from the runtime of the critical functions.

For large RISE++ test procedures such as the cache RAM tests, a similar process dubbed *snapping* eliminates the thousands of name-to-address translations that may occur when using the RISE++ naming services. Once a large test Proc has been developed, debugged, and readied for production use, the process initializes input chip vectors via the RISE++ naming services and takes "snapshots" of their values. This raw scan vector data is then used in the production test procedures in lieu of calling the naming services for data translation. No special compiler is used for snapping, however; the test script itself obtains data snapshots. The snapping process results in large RISE++ test procs that execute five to 10 times faster than the original test scripts.

Other scan architectures

Note that the HP/Apollo Series 10000 scan architecture was developed before the IEEE 1149.1 boundary-scan standard[10] and differs from it in several areas. This means that RISE++ as currently implemented cannot handle devices conforming to that standard.

However, most if not all of the concepts and techniques used to design and implement RISE++ are applicable to 1149.1 and other scan architectures. For example, remote testing has been achieved for 1149.1 architectures via specialized hardware attached to a host system that supports the remote display capabilities of the X Window System.[11]

Test systems that run on workstations or personal computers rather than expensive dedicated test machines have been described elsewhere.[12,13] Most Unix systems provide simple databases like those used for symbol storage and retrieval in RISE++, and other more sophisticated database systems are commercially available. The Tcl software used to implement the RISE++ extensible command line interface is freely available via the anonymous file transfer protocol (FTP) over Internet from sprite.berkeley.edu. Discussions concerning it can be found on the Usenet comp.lang.tcl newsgroup. Finally, compilers for the C++ programming language are readily available for most platforms. All in all, the success of RISE++ comes from the application of proven technologies to the scan-testing problem domain.

Comparison to other systems

Three other remote scan debugging systems for the HP/Apollo Series 10000 workstation preceded RISE++. These programs proved that remote scan testing was possible, and their development provided many insights into how a tool like RISE++ should ideally be designed and implemented. The Advanced Technology Logic Analyzer System (ATLAS)[1,14] was the first of these systems, followed by the Extensible Logic Analyzer (ELA),[14] and the first-generation Remote Interactive Scan Environment (RISE). All three tools aid in the debugging and testing of the prototype hardware, but only RISE was intended for use in a manufacturing environment.

The implementation of ATLAS was based on prototype versions of much of the software used within RISE++. Careful analysis showed that substantial efficiency gains could be made by redesigning the ATLAS scan subsystem driver to execute faster and take up less RAM space. The scan subsystem driver software used by RISE++ was the result of this analysis and redesign. Also, improvements could be made in the remote procedure call server software; this resulted in the custom remote procedure call code used in RISE++.

For scannable entity naming services, ATLAS relied upon compiled Pascal records to represent scan vectors, and the fields of the records could be examined and modified via a software debugger. Developers quickly recognized that considerable flexibility could be gained by moving to an interactive naming service.

The designs of ELA and RISE solved some of the ATLAS problems. For example, the scannable entity-naming schemes of both ELA and RISE relied upon ASCII text files. Loading these files either when the program starts up or at the request of the user allows users to easily customize the names of the scannable elements.

However, an effective naming mechanism for RISE++ was needed to improve the speed and efficiency of the name-loading mechanisms. Measurements for ELA indicated that the naming files for each board in the system took approximately five minutes to load. For a system containing seven or eight boards, the ELA and RISE naming mechanisms were not efficient. The dynamic name-loading mechanism of RISE++ overcame these problems.

The command lines of ATLAS and ELA provided limited extensibility for users. RISE provided a graphical user interface rather than a command line, making predetermined operations simple to perform. However, users required

All in all, the success of RISE++ comes from the application of proven technologies to the scan-testing problem domain.

programmability at the command line level to create custom test scripts that solve special unforeseen test problems. The programmable RISE++ command line interface grew out of the recognition of this need.

The Advanced Support System for Emulation and Test (ASSET) from Texas Instruments is arguably one of the most advanced scan software systems commercially available today.[12] With ASSET, users can develop and apply tests to scannable circuits that conform to the IEEE 1149.1 boundary-scan standard. It shares many of the same features of RISE++, such as the naming of scannable entities and commands for abstract scan operations.

ASSET executes on a personal computer, which is attached to the system under test via a direct hardware connection. ASSET provides no programmable command language, but it does allow users to use C++ modules to customize its graphical user interface for their specific test applications. The fact that the Series 10000 scan architecture preceded the 1149.1 boundary-scan standard meant that ASSET was not a viable environment for Series 10000 test development.

The various abstractions provided by RISE++ provide a productive environment for the development and application of scan-based tests. The scan subsystem driver software allows RISE++

to provide low-level access to the scan hardware from its command line. Users can accurately control the testability features of the Series 10000 workstation from a remote system. At the same time, the RISE++ naming services provide a high-level logical view of the scannable hardware under test by allowing arbitrary named scan bits to be grouped into logical scan buses. The use of the programmable Tcl language enables users to build upon and greatly extend RISE++ functionality to suit their needs. Even tests for embedded nonscannable logic such as cache RAMs can be easily developed using scan test techniques due to the powerful combination of the RISE++ features described here.

Even though RISE++ is a complex software system, the use of object-oriented design and implementation techniques made it relatively easy to develop. RISE++ is composed of approximately 40,000 lines of C++ and C, and it required one person-year to design and implement.

Note that the design goals and architecture of RISE++ are suitable for most other scan architectures, including the 1149.1 boundary-scan standard, even though the current implementation is specific to the Series 10000 scan architecture.

The object-oriented design and implementation of RISE++ has resulted in a system that mirrors the construction and operations of the hardware domain. The abstractions provided by RISE++ fit naturally into the problem domain of hardware designers, test engineers, and manufacturing technicians, thus enhancing their productivity.

For several reasons, including an internal company focus on Hewlett-Packard's PA-RISC architecture and a shift away from the HP/Apollo Series 10000 architecture, work on RISE++ ended in early 1991. I hope the ideas described here, which made RISE++ a productive and successful test tool, will help others who face similar test development needs.

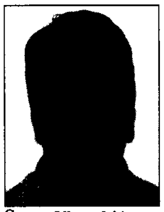
Acknowledgments

I thank Craig Bardenheuer, John Hoglund, and Mike Ricchetti for their contributions to the design of the RISE++ system. Mark Kline and Paul Mageau helped define the requirements for the extensible command line interface. Dan Stirk and John Wood wrote most of the RISE++ Tcl procs that exist today, and in particular Dan was responsible for the development of the cache RAM test code and the snapping process.

Thanks also go to Charlie Matthews, who helped tremendously with the technical literature searches required for this article, and Tom Lynch, who helped obtain the example RISE++ output. Finally, I thank Craig Bardenheuer, John Hoglund, Larry Kaylor, Tom Lyons, and especially Teri Witham for their help in reviewing and improving the manuscript.

References

1. B.I. Dervisoglu, "Scan Path Architecture for Pseudorandom Testing," *IEEE Design & Test*, Vol. 6, No 4, Aug. 1989, pp. 32-48.
2. O.F. Haberl and T. Kropf, "A Chip Solution to Hierarchical and Boundary-Scan Compatible Board Level BIST," *Proc. Second Great Lakes Symp. VLSI*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 16-21.
3. J.K. Ousterhout, "Tcl: An Embeddable Command Language," *Proc. 1990 Winter USENIX Conf.*, Usenix Association, Berkeley, Calif., 1990, pp. 133-146.
4. J. Reilly et al., "Processor Controller for the IBM 3081," *IBM J. Research and Development*, Vol. 26, 1982, pp. 22-29.
5. *4.3BSD Unix Programmer's Manual Reference Guide*, University of California, Berkeley, 1986, ndbm(3).
6. G. Booch, *Object Oriented Design with Applications*, Benjamin/Cummings, Inc., Redwood City, Calif., 1991.
7. B. Stroustrup, *The C++ Programming Language, Second Edition,* Addison-Wesley, Reading, Mass., 1992.
8. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 39-59.
9. L. Zahn et al., *Network Computing Architecture*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
10. IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary-Scan Architecture*, Institute of Electrical and Electronics Engineers, Piscataway, N. J., Feb. 15, 1990.
11. K.T. Kornegay and R.W. Brodersen, "A Test Controller Board for TSS," *Proc. First Great Lakes Symp. VLSI*, IEEE CS Press, 1991, pp. 38-42.
12. *ASSET Scan-Based Diagnostics User Guide*, Texas Instruments, Inc., Dallas, Tex., 1990.
13. A.J. van de Goor and J.A.M. van Tetering, "A Low-Cost Tester for Boundary Scan," *Microprocessors and Microsystems*, Vol. 15, No. 2, Mar. 1991, pp. 82-89.
14. B.I. Dervisoglu and M. Keil, "ATLAS/ELA: Scan-Based Software Tools for Reducing System Debug Time in a State-of-the-Art Workstation," *Proc. 26th ACM/IEEE Design Automation Conf.*, Association for Computing Machinery, New York, 1989, pp. 718-721.



Steve Vinoski is a software design engineer with the Hewlett-Packard Distributed Object Computing Program. His work on RISE++ took place while he was a member of the Testability and Diagnostics Group. He has also worked as a test engineer in the Advanced Development Group at Texas Instruments, Houston, Texas. His technical interests include the C++ programming language, object-oriented programming, distributed computing, and compilers. Vinoski holds a BSEE degree from Christian Brothers University, Memphis, Tennessee. He is a member of the IEEE Computer Society, the Association for Computing Machinery, and the Tau Beta Pi National Engineering Honor Society.

Address questions and comments about this article to the author at MS CHR-03-DW, Hewlett-Packard Company, 300 Apollo Drive, Chelmsford, MA 01824; vinoski@apollo.hp.com.