AGATHA: An Integrated Expert System to Test and Diagnose Complex Personal Computer Boards

Daryl Allred, Yossi Lichtenstein, Chris Preist, Mike Bennett, and Ajay Gupta

PA-RISC is Hewlett-Packard's (HP) reduced instruction set computer (RISC) architecture that is used in its high-performance computer systems (Mahon et al. 1986). Implementations of this architecture have produced some of the most complex processor boards that HP makes (Robinson et al. 1987; Gassman et al. 1987): They can contain as many as 8 very large scale integrated (VLSI) chips—most of them custom, from central processing units to bus controllers to floating-point processors—several high-speed random-access memory arrays, one or more high-speed buses with over 100 lines, and many other components. In large part because of this complexity, the testing of PA-RISC processor boards became a bottleneck, resulting in an undesirable backlog of undiagnosed boards, growing at a rate of 10 percent each month.

Testing PA-RISC Processor Boards: The Process

Part of a typical production flow for a PA-RISC processor board is diagrammed in figure 1, emphasizing the board test. After the board is fabricated, it is run through an in-circuit, or *bed of nails test*. This process consists of individually testing the components and connections on the board and catches misloaded or missing parts and most of the open circuits and shorts between circuits.

Next, the board is tested on a PRISM tester, a Hewlett-Packard proprietary test system that integrates scan-based testing with functional testing (Schuchard and Weiss 1987; Weiss 1987). *Functional testing* involves testing the behavior of various subsystems on the board, verifying that they behave properly in a system environment. *Scan-based testing* takes advantage of HP's proprietary scan methodology, which is implemented on its VLSI chips to poke around their internal registers and test individual blocks of the chips or buses between them. All these tests can be run at various voltages and frequencies on the PRISM tester.

At any point in the production line or even in the field, a faulty board can be returned for diagnosis and repair. Diagnosis is normally done at a PRISM test station, where the technician has access to a battery of diagnostic tests that can help in localizing the problem.

The Problems with Testing

Because of the board's complexity, technicians found it difficult to diagnose failed boards. Consequently, manufacturing began to encounter several problems that became severe:

First, the PRISM test station became a bottleneck in the production process. The difficulties in diagnosing failing boards, along with long test times, began to interfere with the flow of boards on the production line.

Second, an unacceptable backlog of undiagnosed boards accumulated, growing at a rate of 10 percent each month. This backlog resulted from the time and difficulty in diagnosing certain failure modes.

Third, the thorough diagnosis of boards was a time consuming and tedious process. Thus, the technicians would take shortcuts to save time. However, these shortcuts sometimes led to an incorrect part being replaced, resulting in further repair cycles being needed.

Fourth, it was difficult to effectively train new technicians to diagnose PRISM failures. The learning curve was large. Furthermore, new manufacturing sites were being opened worldwide, exacerbating the problem.

The difficulties of diagnosing faulty processor boards in manufacturing were attributable to the following conditions:

First, because the processor board is complex, it has many subsys-

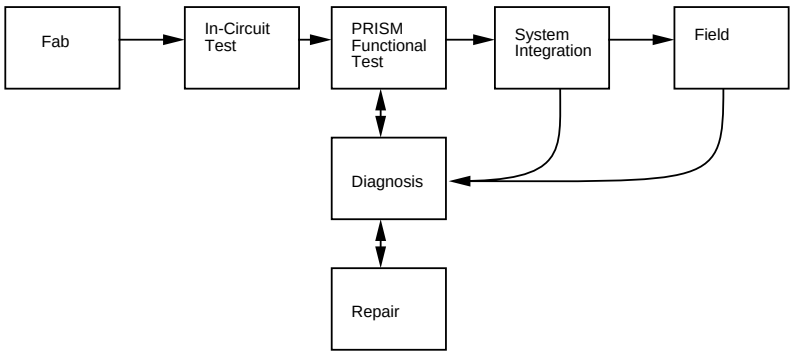

Figure 1. PA-RISC *Board-Test Process.*

tems with different functions. PRISM addresses these various subsystems with different tests, yielding a diversity of test output. A few tests try to suggest a failing component or subsystem; others just report a failure mode or code that only indirectly identifies a fault. Still others dump a lot of internal processor-state information that can only be interpreted in light of the processor architecture and the test strategy. The technician has to become familiar with the output of these diverse tests to effectively diagnose boards.

Second, it is difficult for the technician to remember lots of special-case failure modes. Some of these failure modes are identified by unique patterns in the data, often across several test results. Because these special cases are less frequent than the normal failure modes, they are easily forgotten.

Third, some diagnostics produce reams of data. It is difficult and time consuming for the technician to deal with such diagnostics, and important information can be overlooked in the large volume of data.

Fourth, some test results are low level—hex numbers and bit patterns—that must manually be manipulated, indexed into tables, and cross-referenced to map to a component. This work is tedious and error prone.

Fifth, there is some uncertainty in some of the test results. For example, a test might suggest replacing a certain chip, but in fact, this chip is the actual fault only part of the time. Additionally, some faults exhibit intermittent behavior, such that a discriminating test might not always detect it.

Problem Assessment

The nature of the problem just described suggested to us that automation of the board diagnostic process by expert system technology would be of great benefit. The expertise was available but in short supply, and it was time consuming to pass the information to new individuals and sites. The diagnostic process was understandable, although laborious to carry out, because of the large amounts of data and possible failure modes that needed to be addressed. The problem was significant, with a large potential benefit to be gained from removing the bottleneck in the production process.

However, two issues seemed to go beyond the standard solutions for such problems:

First, the functional diversity of the board and, as a result, the diversity of the testing and diagnosing techniques demanded several different inference strategies and knowledge bases. It appeared to be impractical to expect a single expert system to diagnose all the different tests.

Second, the difficulties that technicians and operators suffered in interacting with the tester suggested that the expert system should replace the current front end of PRISM and become not only a diagnostic inference machine but also a high-level human interface.

The different tests were analyzed along several dimensions to start designing and implementing a diagnostic expert system. These dimensions were the following:

First was the quality of output. Some tests produce internal information that can be used in diagnosis. Others output only pass-fail data.

Second was the amount of data, from a single line to hundreds of lines of hexadecimal data.

Third was the degree of interaction necessary with the user. Some tests require no interaction, whereas others require complex manual tests to perform diagnosis.

Fourth was the sophistication and depth of the knowledge used by the expert, from simple heuristic matching to a deep understanding of causality within a subsystem.

System Design

To solve these design issues, we decided to implement a suite of mini expert systems, called *slices*. Each slice diagnoses the results of a single test. The inference process for each slice was tailored to how the test measured according to the previous four dimensions. Where large amounts of data and causal rules are available to a slice, it uses a generation-elimination process with many data abstraction predicates and no

user interaction. Other slices need to interact with the user to gather more data before recommending further tests, so they use a table lookup process.

All slices cooperate with each other and communicate through a *diagnose manager*, a further slice that is responsible for coordinating the overall diagnostic process and interfacing with the tester. The entire system, named AGATHA, is fully integrated with the PRISM tester; furthermore, with a user interface, it forms the new front end of the tester.

The Slice Architecture

The overall architecture of AGATHA (figure 2) consists of 9 different slices, with 27 associated knowledge bases and databases. The 9 slices need only 6 different inference engines between them; sharing of inference machinery is allowed between slices with similar inference strategies. The diagnose manager slice is responsible for passing control among the other slices, depending on previous test results and diagnostic hypotheses. It is also responsible for feeding results from the slices to the user and information from the user to the relevant slice.

To show the different inference methods involved in the slices, we focus on three slices: the diagnose manager, the cache bus, and the diagnostic interface port (DIP). We chose the latter two because they are different from each other in terms of the dimensions described in Problem Assessment. The cache bus slice illustrates a slice with large amounts of data but little user interaction. It analyzes the data using causal rules derived from the expert. The DIP slice, however, has sparse data and uses heuristics and close interaction with the user in performing further tests to yield the final diagnosis.

The Diagnose Manager. The diagnose manager has overall responsibility for coordinating the test runs, invoking slices to interpret them, reporting suspected faulty components, and servicing requests to repair them. It delegates these tasks to three separate submanagers: The *slice manager* invokes the proper slice for a failed test; coordinates the display of suspects; and directs other requests by the slice, such as to run a test, to the diagnose manager. The *repair manager* services requests to perform repairs, usually after the slices finish analyzing the test data. After repair, it reruns the original test to confirm the problem on the board was fixed. The *test manager* services all requests for tests to be run on the tester, providing a common interface to PRISM.

The most interesting of these submanagers is the test manager. For each test request, it must perform the following five tasks: (1) select a test point at which to run the test (the *test point* is the power supply voltage and clock frequency to be supplied to the board under test by the

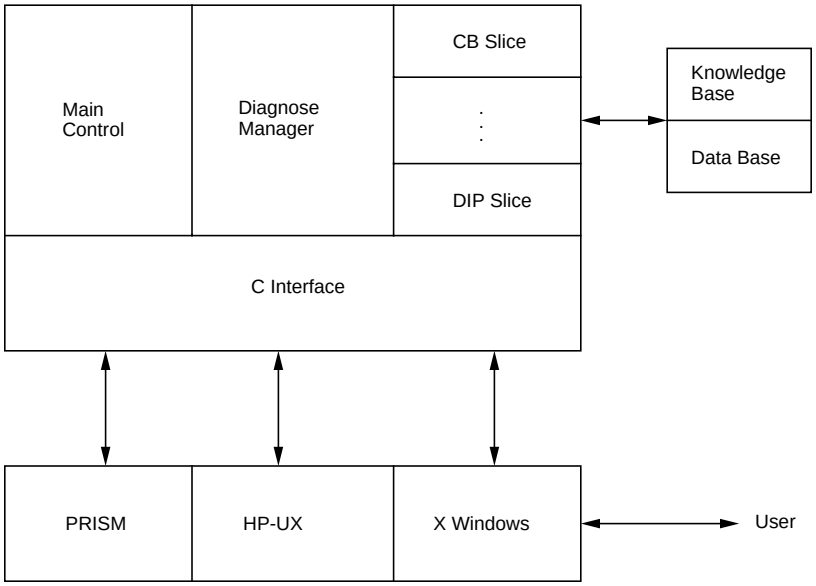

Figure 2. AGATHA *Architecture.*

PRISM tester), (2) run the test, (3) determine the failure mode of the test (for example, is this test intermittent?), (4) retry the test if necessary, and (5) recommend to the diagnose manager what to do next.

First, the test manager selects a test point at which to run the test. It can select one of a set of predefined test points or an entirely new one in an attempt to get a test to fail more reliably. The test manager then runs the test at the computed test point, parses the test output, and makes a summary decision about whether it passed or failed. No attempt is made to otherwise interpret the test results at this point.

Next, based on the pass-fail results of the test, a new failure mode is computed. The test manager supports the following failure modes:

First is *hard*: Failing tests fail at all test points with little or no variation in their output (that is, they don't differ dramatically from test point to test point).

Second is *dependent-repeatable*: Dependencies on voltage or frequency were determined (that is, it only fails at certain voltages or frequencies), and it's repeatable.

Third is *dependent-nonrepeatable*: Dependencies on voltage or frequency were determined, but it's not repeatable (it only fails sometimes at these voltages or frequencies).

Fourth is *intermittent*: It's not failing hard, but no dependency can be determined (for example, random failures).

The failure mode is a symbolic representation of uncertainty in the test results, accounting for possible intermittent test failures. Heuristic knowledge about tests, along with data about test results, is used to infer the failure mode; for example:

```
IF        this board is a field return AND
          the board test passed
THEN      the failure condition is intermittent
BECAUSE   this board has failed before (in the field)
          but won't fail now
```

With a new failure mode computed, the test manager next determines whether the test should be rerun using heuristic rules. A test might need to be run because of suspected uncertainties in the test results. An example of a heuristic follows:

```
IF    the test passed AND
      the failure mode is either dependent-nonrepeatable or intermittent
THEN  the test should be rerun up to 3 times until it fails
```

Finally, the test manager is ready to recommend to the diagnose manager what to do next. If the test failed, the associated slice is determined, and it is recommended for invocation. If the test passed, then the previous slice is reinvoked with this new information; as a result, the slice can recommend further tests. Thus, the test manager uses heuristic knowledge embedded in procedural control to run tests and manage uncertainty in their results, removing this burden from the slices.

The Cache Bus Slice. The cache bus subsystem on the assembled circuit board consists of several large VLSI chips that communicate through a bus of about 150 lines. Each chip is connected to a subset of these lines and is able to transmit onto the bus by driving values onto the lines. These lines can, in turn, be read by the other chips.

Failures that can occur in this system include shorts between lines, opens on lines, and chips failing to properly transmit or receive. A fault can be intermittent; that is, sometimes a test will miss it, but at other times, it will show up. Also, multiple faults can occur. For example, several shorts can be caused by solder being splashed across the board.

The PRISM tester tests the cache bus by getting each chip to drive a series of binary values onto the bus and getting all chips to read each value back. This process is done automatically and produces a large amount of data. Discrepancies between the expected values and the observed values in these data are used by the cache bus slice to diagnose the fault.

The cache bus slice is responsible for diagnosing failures in the

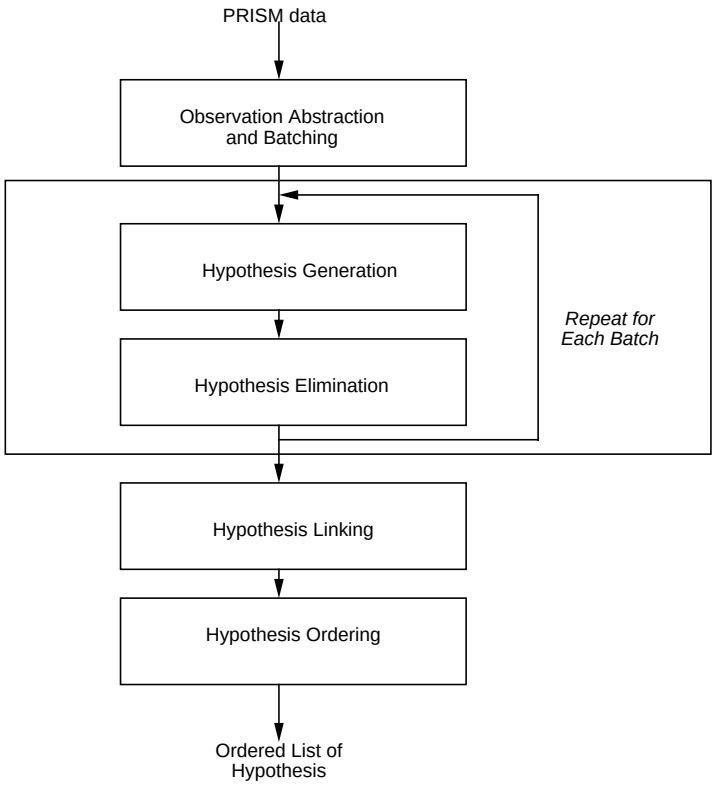

Figure 3. Cache Bus Slice Architecture.

cache bus test. The design of this slice (figure 3) allows it to handle both intermittent faults and the majority of multiple faults. Rather than dealing with the bus system as a whole, it divides it into semi-independent subsystems, namely, each line.

The data from the test is divided into *batches*, one associated with each line that is exhibiting bad behavior. Each batch contains only those data that were received incorrectly off its associated line. These data are then used to deduce the fault on this particular line. Hence, rather than assuming that the cache bus as a whole has only a single fault, the system can treat each line independently and assume that there is at most one fault for each line. The single fault assumption is replaced with the single fault to a line assumption.

A template generates a list of hypotheses for each line. Where a multiple fault on a single line is to be considered, this information is explicitly entered in the template. Elimination rules are then used to re-

move as many of these faults as possible. The knowledge in these rules is derived from the causal rules of the expert by taking its contrapositive—if a hypothesis makes a prediction, and this prediction is found to be false, the hypothesis can be eliminated. Elimination rules take the following form:

IF a '1' is observed on the line when a '0' was expected
ELIMINATE short to ground and short to another cache bus line
BECAUSE these faults can only pull a line to '0'

After elimination, each line has a small number of hypotheses associated with it. However, a line is only semi-independent. Many hypotheses, such as "short between lines" and "bad VLSI," manifest their behavior on several lines. Hence, *linking rules* are used. These rules take hypotheses associated with different lines and combine them, where appropriate, into single hypotheses that explain the bad behavior of several lines. They have the following form:

LINK: Short to cache bus on line 1
WITH: Short to cache bus on line 2
IF: Line 1 and line 2 are adjacent somewhere on the board
TO GIVE HYPOTHESIS: Short between line 1 and line 2

Finally, the remaining hypotheses are ordered, using heuristic knowledge, according to their likelihood. This list is returned to the diagnose manager for presentation to the user.

Thus, the cache bus slice combines causal-based rules with heuristic knowledge. The causal rules are used to deduce which hypotheses are possible and which are impossible. The heuristic knowledge is then used to determine the relative likelihood of the hypotheses that remain. Full details of the cache bus slice are available in Preist, Allred, and Gupta (1991).

The Diagnostic Interface Port Slice. Before scan-based tests can be performed on a VLSI chip, the DIP on the chip, which is the serial scan port, must be tested to verify that the tester is able to properly communicate with the chip and that other electronic subsystems are working reasonably well (that is, the power, clocking, and reset subsystems). The DIP test tests the DIP port on all VLSI chips, and the DIP slice diagnoses any failures.

Unlike the cache bus slice, the DIP slice works with simple pass-fail and test-point information from the DIP test and interacts with the user to give the final diagnosis. Because of the sparsity of data, it is unable, alone, to deal with intermittent and multiple faults, relying instead on the user to explore these possibilities.

The DIP slice performs its diagnosis in two stages: First, it proposes which subsystems it considers are the main and secondary suspects. Second, it aids the user in performing further manual tests on a particular

subsystem to determine if it is indeed faulty and, if so, exactly where.

The first stage, generating suspects, uses a heuristic mapping from the symptoms to the possible causes. The symptoms are the pass-fail data of the DIP test. The possible causes, divided into main and secondary suspects, are the candidate subsystems and connections, at least one (possibly more) of which is faulty. There are 20 such mapping rules; the following is an example:

IF all chips failed the DIP-test
AND the System-Test passed
THEN suspect the MDA as a main suspect
AND the Reset, Clock, Power, and System-bus-connector as secondary suspects.

The main-secondary distinction is a simple form of probability handling. More complex schemes were rejected because the expert couldn't substantiate a finer separation of fault likelihood.

The second stage of reasoning—guiding the user in manual tests—is an iterative process. The user chooses to concentrate on a particular subsystem and tries to find out which of its components (if any) are faulty. The decision of which subsystem to focus on is left to the user.

Each subsystem is composed of a set of components and a list of tests to test them, both automatic and manual. A table (figure 4) then represents the knowledge that tests would give about the components, as follows: If a test fails, at least one of the components with an F entry in the table will be faulty; if a test passes, all the components with a P entry in the table will be functional (not faulty).

With this knowledge, together with an approximate cost of performing each test, the DIP slice presents the user with an ordered list of tests that are worth carrying out. The user then chooses which test to perform, receives instructions on how to do it, performs the test, and enters the result (pass or fail) into the system. This process continues until the DIP slice is able to diagnose a component as faulty, or the user chooses to explore another subsystem.

Hence, the user, working with AGATHA, is able to explore the different candidates and diagnose exactly which of them is indeed failing. The user is always in control but can rely on an ordered set of tests, arranged to isolate the fault as fast as possible.

Tester Integration

As previously indicated, to solve crucial testing issues, Agatha would not only act as an automated diagnostic system but would also provide a user interface and become the new front end to PRISM.

The challenge here was to integrate AGATHA into an old PRISM system

Components:

1	Tester
2	System-bus (signals PON/NPFW)
3	Reset-buffer
4	SIU
5	Cache-Bus (signals NRS0 and NRS1)
6	VLSI chips (other than SIU)

Tests:

1	Probe NRS0/1 on Cache Bus
2	Scope NRS0/1 on Cache Bus
3	Ohm out NRS0/1 on Cache Bus
4	Probe PON/NPFW on System Bus
5	Scope PON/NPFW on System Bus
6	Ohm out PON/NPFW on System Bus
7	Probe and scope PON/NPFW on both sides of buffer
8	Inspect System Bus connector
9	Test rest path through tester subsystem

F/P Table:

1 Tester	2 Sysbus	3 Buffer	4 SIU	5 Cache Bus	6 VLSI	Tests to perform
F	F	F	F	F		1
			F/P	F/P		2
				F/P		3
F	F	F				4
F/P	F/P	F/P				5
	F/P					6
		F/P				7
	F/P					8
F/P						9

Figure 4. Diagnostic Interface Port Slice Knowledge for the Reset Subsystem.

whose code had not been touched for a long time. We opted to layer AGATHA on top of PRISM, as diagramed in figure 5. Communication is through the HP-UX interprocess communication facilities. With this layering, the PRISM code remains virtually untouched except for minor modifications of the stream files or scripts.

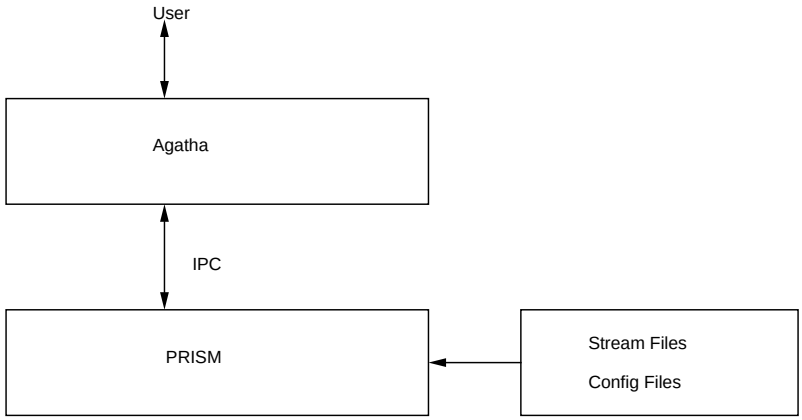

Figure 5. AGATHA-PRISM Interface.

The layering of AGATHA on top of PRISM had several advantages:

First, the PRISM code remained unmodified, minimizing its maintenance requirements.

Second, a new, friendlier front end was now provided for PRISM.

Third, AGATHA now detects some failures that PRISM could not detect before. For example, some messages are printed by system tests that PRISM never recognized, possibly passing faulty boards. AGATHA now detects these failures and reports them accordingly, improving the reliability of the testing process.

Fourth, some tests that required a long, tedious sequence of commands to execute are now automated by AGATHA. This process saves time and is more thorough because the technician would otherwise shy away from running the test to completion or even running it at all.

How AGATHA Is Used

In manufacturing, operators test the boards on the PRISM testers, separately binning the good and bad boards. Later, technicians put the failed boards back on the tester and diagnose them. This procedure was modeled in AGATHA, supporting three levels of users. These users are, in order of increasing capability, (1) the operator (the user can only test boards and bin them), (2) the technician (the user can also diagnose boards, including running manual tests, as well as access other utilities, such as review or print results), (3) the maintainer (the user can also edit the knowledge bases and databases and drop down into Prolog).

This approach made a simpler interface for the operator to learn, greatly reducing the learning curve. At the same time, additional flexi-

bility was available for technicians, making their task simpler. This flexibility proved to be a significant contribution of AGATHA—a major benefit to our users.

With this model, AGATHA could run several additional diagnostic tests while in operator mode, which would save time for the technician who would otherwise have to run them later. Thus, a mode is provided, called *automatic test*, where AGATHA automatically runs all tests it believes necessary, provided no intervention is required from the user. This approach provides a significant time savings to the technician and effectively reduces the skill level required for this task. This feature can be turned off during times of heavy workload, where higher throughput is needed from the operators.

One major decision was whether to provide a diagnostic system or a diagnostic adviser. Specifically, would it dictate to the user what to repair or only advise? During knowledge acquisition, we found that which component to repair would vary depending on certain circumstances, including variations in the production process that could cause one failure mode to start appearing more frequently. Hence, AGATHA only advises the user on repairs, presenting an ordered list of candidates to the technician, who chooses the most suitable. The technician would usually pick the first item from the list, unless s/he was aware of extenuating circumstances that might suggest another.

Development

AGATHA was a joint development effort between the Knowledge-Based Programming Department (KBPD) of HP Labs, Bristol, England, and the Integrated Circuit Business Division (ICBD) in Fort Collins, Colorado. The development process was broken down into the following phases:

The alpha phase produced a prototype version that was deployed first in ICBD. It consisted of only three slices. The goal was to gain experience with it and get user feedback.

The beta phase reviewed this feedback, producing major refinements to the slices and a new diagnose manager. More slices were added. This phase produced the first production version of AGATHA, which was installed at a user's site.

The refinement phase continued to add slices and make refinements to the knowledge. More users were added, leading to the manufacturing release of AGATHA.

The maintenance phase followed, where minor enhancements and refinements are ongoing.

Implementation Language

Prolog was chosen as the principal implementation language for AGATHA. (In the figure 2 diagram, everything above the C interface box was written in Prolog, including the knowledge bases and databases.) The main reason for choosing a language, rather than an AI shell or tool kit, was the need to be able to code different inference strategies for the different slices. These strategies do not always fit into the classical forward-backward chaining regimes provided by shells and, thus, would have been awkward and inelegant to code in this way. The disadvantage with this decision is that we lose the support that the shell provides, namely, a good user interface and ready-written inference strategies. These elements had to be coded and maintained, imposing some additional burden on the project, yet could completely be tailored to the task, not restricted by what a shell or tool kit provides.

Verification

Prior to AGATHA, all tests run on PRISM were directly sent to the printer. We had lots of printouts to use in the design of AGATHA but no machine-readable tests to verify the implementation. Therefore, scaffolding was built to capture the test results on disk to be used for verification. With this scaffolding, a large suite of verification cases was gathered from the following sources: (1) poisoned boards (faults were caused on an otherwise good board, including shorts, opens, and missing parts), (2) bad VLSI (bad VLSI—acquired from field returns, and so on—were inserted into a socketed board), and (3) actual cases (later on, the verification cases were augmented with actual cases that AGATHA encountered while in use on the production line).

These verification cases helped to verify the knowledge and functions of AGATHA and refine it while in use. They helped assure the delivery of a reliable, confident system to our users.

Maintainability

Maintainability was a foremost consideration throughout the design and implementation of AGATHA for two reasons: First, it was decided from the outset that the original designers of significant parts of the system, HP Labs, would not be responsible for their maintenance. Instead, it would be the work of ICBD. Second, the system had to be able to deal with new board types, which were structurally different from the original board but were tested using the same tester.

This approach led to the clear separation in each slice of the knowledge specific to a certain board type and the knowledge specific to the

tester. Hence, the tester rules had to deal with an abstract board and call the structural knowledge base to gather information specific to a certain board type.

Where the knowledge consisted of simple relations (such as which faults resulted in the failure of which further tests), it was directly represented as relational tables. Rules then access these tables as necessary. This approach reduces the number of rules needed and allows easy and rapid maintenance.

This policy has paid off. The system is now entirely maintained by ICBD. It has successfully updated the system to support three different board families, with a fourth nearly completed. (Some families have multiple board types distinguished by varying cache random-access memory sizes, and so on, all of which AGATHA has to know about.) This update process was partly automated. C routines access design data files that are used by the PRISM tester and extract structural information that is relevant to AGATHA, constructing files of Prolog clauses. The update process takes only a short amount of time to deal with a new board family.

Other than supporting new tests and board families, most of the maintenance since the production release of AGATHA has been in adding new features and enhancements rather than refining the slices or their knowledge. In addition, most of these enhancements were outside the slice architecture proper.

Deployment

AGATHA has been in routine use since January 1990. It was successfully deployed at 3 sites within HP—2 are manufacturing facilities; the third is a field repair center. One manufacturing site, for example, uses it 24 hours a day: Operators test boards on the production line, often letting AGATHA diagnose them to the extent that manual testing isn't required. Technicians then examine failure reports and let AGATHA work on boards that might need further diagnosis.

The field repair center receives failed boards from the field and diagnoses them on AGATHA. Although the center doesn't see the volume of boards that the production sites do, AGATHA is perhaps even more critical to its operation for this reason: It preserves diagnosis and repair knowledge that it might otherwise lose with low volumes. Furthermore, it has a great need for some of the features of AGATHA that support uncertainty. Because a board returned from the field is suspected to be faulty, one can't simply return it if the board test passes when run just once; it must be thoroughly exercised and checked out. AGATHA addresses such intermittences by running a test many times to try to get it to fail.

Results

AGATHA has been favorably received by its users and has proved to have many benefits for them. It has addressed many of the production problems that were being experienced prior to AGATHA:

First, the PRISM test station is much less a bottleneck in production. Although the long, raw test times (which AGATHA has no control over) are at times cause for congestion in production, the savings in diagnostic time have greatly alleviated this problem.

Second, AGATHA, in combination with other efforts on the production line, has helped to virtually eliminate the backlog of undiagnosed boards.

Third, AGATHA saves time by automatically running tests, especially when run by an operator. AGATHA also automates some tests that used to be painstaking, manual tests, saving additional time. Time savings is one of the principal benefits hailed by all AGATHA users.

Fourth, costs are saved when AGATHA automatically runs tests. By effectively lowering the skill level required, the test runs can be done by operators rather than technicians.

Fifth, AGATHA makes a more thorough diagnosis, eliminating many common human errors. It also improves test reliability by detecting failures not formerly caught by PRISM.

Sixth, AGATHA has provided an easier-to use-interface to the PRISM tester. In a recent survey, users gave a top score to AGATHA's friendly user interface. Coupled with automation, this interface has significantly reduced technician and operator training time and greatly improved user satisfaction.

It's difficult to quantify the full impact of AGATHA within HP, but we list some benefits here:

First, during AGATHA's development, test time was reduced by 80 percent on one board, yielding tremendous cost savings. Although there were several factors at work here, AGATHA was a principal contributor in this effort.

Second, the field repair center indicates AGATHA reduced scrap rate, average repair time, training costs, and material costs. These savings add up over the life of a product and could especially be valuable toward the end of its life when expert knowledge on an aging board could otherwise be scarce.

Third, one production site related a one-fourth reduction in technician training time, with the ramp-up time for a new technician dramatically improved. It also reported a 40-percent reduction in diagnostic time and a significant increase in user satisfaction with the friendlier interface.

In addition to the gratifying manufacturing results, the joint development effort between HP Labs and ICBD proved to be of mutual benefit: ICBD gained expertise in the design, development, and deployment of expert system technology. HP Labs gained knowledge of realistic problems in electronic circuit diagnosis. This knowledge has been used to drive a longer-term research program in model-based diagnosis.

Acknowledgments

We want to recognize the valuable contribution of many others: Rick Butler was the domain expert who consulted on the project. Caroline Knight of HP Labs invested a lot of time in training ICBD in knowledge-acquisition techniques. Jason Brown was a summer student who helped code part of AGATHA. In addition, we want to express our gratitude to the many operators and technicians who gave invaluable time and assistance on, and feedback to, the AGATHA project.

References

Gassman, G. R.; Schrempp, M. W.; Goundan, A.; Chin, R.; Odinea, R. D.; and Jones, M. 1987. VLSI-Based High-Performance HP Precision Architecture Computers. *HP Journal* 38(9): 38–48.

Mahon, M. J.; Lee, R. B.; Miller, T. C.; Huck, J. C.; and Bryg, W. R. 1986. Hewlett-Packard Precision Architecture: The Processor. *HP Journal* 37(8): 4–21.

Preist, C.; Allred, D.; and Gupta, A. 1991. An Expert System to Perform Functional Diagnosis of a Bus Subsystem, Technical Paper HPL-91-16, HP Labs, Bristol, England.

Robinson, C. S.; Johnson, L.; Horning, R. J.; Mason, R. W.; Ludwig, M. A.; Felsenthal, H. R.; Meyer, T. O.; and Spencer, T. V. 1987. A Midrange VLSI Hewlett-Packard Precision Architecture Computer. *HP Journal* 38(9): 26–34.

Schuchard, R. A., and Weiss, D. 1987. Scan Path Testing of a Multichip Computer. In *1987 IEEE International Solid State Circuits Conference Digest*, 230–231. Gables, Fla.: Lewis Winner.

Weiss, D. 1987. VLSI Test Methodology. *HP Journal* 38(9): 24–25