An Expert System to Detect and
Diagnose Failures in DRAM
by
Venkatesh Satish

Submitted to the
Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
September 20, 1999
Copyright 1999 Venkatesh Satish. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

Certified by

Professor Srinivas Devadas
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
The issue of fault diagnosis in semiconductor memories becomes a greater issue each year as increases in densities result in devices that fail in new ways. This project attempted to capture the expert knowledge of test engineers whose career involves debugging memory chips that fail in nonstandard ways. A well-designed rule-base effectively captured the general strategies used by the experts. The expert system, implemented in JESS and Java, also contained knowledge about items specific to these experts’ task, such as their use of particular equipment and software, as well as physical design information about the devices they diagnose. The system successfully assisted in the diagnosis of a defective device, with experts assigning it an average score of 9.8/10 across 369 decisions. Despite the fact that most expert behavior could not be imitated or captured, the project generally showed that an expert system could provide time-savings, reliability, and training for novices.
## Table of Contents

<table>
<thead>
<tr>
<th>SECTION</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abstract</td>
<td>2</td>
</tr>
<tr>
<td>1.1</td>
<td>Introduction</td>
<td>5</td>
</tr>
<tr>
<td>1.1</td>
<td>Review of Memory Testing /Memory Fault Models</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Related Expert System Work</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Methods</td>
<td>13</td>
</tr>
<tr>
<td>2.1</td>
<td>Knowledge Domain</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Memtest Specifics</td>
<td>14</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Logic Analyzer Traces</td>
<td>18</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Layout and Design Understanding</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Knowledge Representation</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Considering Frames</td>
<td>21</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Knowledge Captured</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>Design / Implementation</td>
<td>25</td>
</tr>
<tr>
<td>2.3.1</td>
<td>System Architecture</td>
<td>26</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Physical Adjacency Understanding</td>
<td>32</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Examining Logic Analyzer Traces</td>
<td>34</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Remaining Modules</td>
<td>36</td>
</tr>
<tr>
<td>2.4</td>
<td>Testing Setup</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>Results</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>Discussion</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Appendix A – Rule Base</td>
<td>48</td>
</tr>
</tbody>
</table>
Table of Contents (Cont.)

<table>
<thead>
<tr>
<th>SECTION</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>References</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>Acknowledgments</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td><strong>TABLES</strong></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>State Descriptions and Corresponding Expert Practices (1-6)</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>State Descriptions and Corresponding Expert Practices (7-12)</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>State Descriptions and Corresponding Expert Practices (13-18)</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>State Descriptions and Corresponding Expert Practices (19-22)</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>Breakdown of Expert Decision Scores by State and Category.</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td><strong>FIGURES</strong></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A Sample Screen Display For Memtest</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>Division of Knowledge In the System</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>An Example of a JESS Rule</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Diagram of the Possible Transitions in the System’s State Representation</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Adjacency Grid with Column Row and Pin Values.</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Stepping through a Logic analyzer trace</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>A Sample Portion of the Interface</td>
<td>38</td>
</tr>
<tr>
<td>8</td>
<td>Expert System / Test Setup</td>
<td>39</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

As memory densities increase at blinding rates, they fail in manners that expose the limitations of existing memory fault models. While researchers have expended much effort in articulating new fault models and corresponding tests, the goal of this project was to approach this general problem from a different standpoint. Specifically, that approach involved writing an expert system that captured the strategies and knowledge of test engineers charged with diagnosing memory faults not detected by standard screening tests. The work involved designing, implementing, and testing a knowledge-based system that assists in the diagnosis of defective dynamic random-access memory (DRAM) modules. These test engineers regularly apply their expert domain knowledge to determine the cause of failure memory modules. They rely on heuristic techniques and clues derived from experience to quickly identify the cause of failure. The premise of this project was that these techniques could be codified into a rule-based system that would successfully assist test engineers in pinpointing failure mechanisms in DRAM modules.

The system assisted in the characterization of modules known to have failed customer tests. It is of importance to quality control groups that a reason for failure be identified in an expedient manner. One of the benefits of such a system is that it has the potential to reduce the time necessary to diagnose a module. It is also important to note that the failing modules these experts diagnose constitute a small handful of relatively complicated faults not typically discussed in testing literature. This research addresses
these obscure types of faults and makes use of relatively widespread knowledge engineering techniques.

1.1 Review of Memory Testing/ Memory Fault Models

In order to give the reader a better understanding of the limitations of existing fault models, it is useful to discuss the relevant testing literature. Those already familiar with the subject may find this review unnecessary. Researchers have constructed a wide variety of fault models, which describe the effects of a range of defects on a particular circuit’s behavior. Sometimes, fault dictionaries are extracted from the models and can be used in diagnosis. The canonical model, the single stuck-at fault (SSF), models the effect of a particular line or subcircuit in a system sticking to either 0 or 1. The SSF has some limited use for diagnosis and fault detection, and there exist a number of models that are more suited to describing faults in Random Access Memories.

Much of memory testing literature treats the development of a variety of fault models that depict typical failure mechanisms in memory. Traditionally, they rely on an abstract model of RAM that treats the memory as a black box that includes an address decoder, read/write logic, and an array of 1-bit cells. In much of the research, the emphasis is placed on the development of tests that are economic in terms of running time while providing reasonable fault detection and coverage. Memory faults are typically classified into one of the following: stuck-at faults, coupling faults, or pattern sensitive faults. Coupling faults are generally defined as those faults in which the contents of one cell directly affect the contents of another. These can be classified into two more specific categories: idempotent coupling faults, where writing a value to one cell forces the other to a specific value, and inversion coupling faults, where a write to the initial cell toggles
the contents of the malfunctioning one. This fault model may be extended to a k-coupling scenario, where the behavior of two cells is dependent on the state of the other (k-2) cells[1].

In addition to these models, there also exists the pattern sensitive fault (PSF) model. In this model, the contents of a cell, or the ability to change its contents, are influenced by the pattern of 0s and 1s in other cells. A neighborhood is defined as the group of cells that may affect the cell in question, referred to as the base cell. Faults that assume a restricted neighborhood are called neighborhood pattern sensitive faults (NPSFs). These faults are classified according to the behavior resulting from writing a certain pattern to cells in the neighborhood. An active fault refers to the situation in which the base cell changes in response to a certain neighborhood pattern. Alternatively, a passive fault indicates that the contents of the base cell cannot be changed. Finally, static faults, are those in which the contents of the base cell are forced to a particular value. Typically, the neighborhood is restricted to those cells that are physically adjacent to the base cell, and furthermore to the four cells directly adjacent to the base cell[2]. Yet another model describes memory behavior when such issues as interconnected cells, word-line shorts, and bit-line shorts occur[3].

There are a variety of tests that have been developed to catch faults described by the above models. Traditional tests include zero-one, checkerboard, and walking 1s and 0s, all of which can be conducted using standard memory test equipment. Zero-one involves writing a background of 0s and doing the same with 1s, and then verifying the array contents. Checkerboard tests refer to the checkerboard pattern of 0s and 1s written in memory, and in a walking test, a 1 is walked through a background of 0s, and contents
are verified each time the 1 is moved. These traditional tests do not even provide complete fault coverage for stuck-at faults, basic coupling faults, let alone some of the more complicated pattern-sensitive ones. There are a number of tests called “march tests,” named so because they march through all cells in the array, writing and verifying different patterns. Among these march tests are the sliding diagonal, GALPAT, butterfly, MATS (Modified Algorithmic Test Sequence), MATS+, MATS++, March B, and March C. It would be impractical to mention the specifics of each test here. In any case, it can be shown that together these tests cover all stuck-at, transition, and 2-coupling faults[4].

However, these tests cannot detect certain pattern-sensitive faults, linked faults, and multiple faults in a reasonable length of time. If one tries, for example, to exhaustively test for PSFs, the test would take order $n \times 2^n$ time to complete[5]. Linked faults are those in which one fault may affect the behavior and presence of another fault. For example, consider two idempotent coupling faults where cell i affects j and cell m affects n. In this case if j=m, then none of the tests mentioned would detect this linked, multiple fault. The entire field of multiple faults is still being developed, because in general, testing for multiple faults is complex, expensive, and time consuming. A literature survey revealed only a handful of published research on these types of faults, with most of it focusing on abstract representations and algebras to deal with multiple faults[6]. The tests that have been developed detect a very specific set of cases [7] [8]. For example, march tests have been developed to detect 5-coupling, 6-coupling, and in some cases, 9 coupling faults[9] [10]. In addition to purely deterministic tests, researchers have created probabilistic methods using Markov chains to detect more complicated faults[11].
Still, it should be emphasized again that the fault models do not articulate many problems that IBM’s memory test engineers encounter regularly. For example, IBM troubleshooters have encountered pattern sensitive faults where the base cell and coupled cell were not directly adjacent though close as far as regards physical proximity. Running a battery of standard tests would not detect this fault. In addition, IBM troubleshooters must handle faults that are sporadic or intermittent, and these classes of faults are usually not addressed in current testing research. Because test engineers use various rules of thumb and strategies to arrive at a diagnosis, an expert system seemed a reasonable project. They try to gather various clues, such as whether the failure is sensitive to temperature changes, in order to speculate on a failing mechanism and verify their suspicions. The apparent existence of such rules made it seem that knowledge engineering techniques were an appropriate means of solving the problem.

1.2 Related Expert System Work

A relatively thorough search revealed that there was no expert system designed to debug relatively exotic memory failures. However, there was much work done in the area of hardware diagnosis with expert systems. These systems were considered when deciding on this project’s knowledge representation, design, and implementation. Furthermore, this project represents yet another application of knowledge engineering practices, so it addresses the question as to whether the project’s subject matter is a suitable application for expert systems.

Generally, expert systems consist of a knowledge base, which is a representation of the domain of knowledge that applies to a given problem, and an inference engine, which applies the information to arrive at a solution. One relatively common design is to
structure the knowledge as a set of rules and then use forward chaining, or successive application of the rules to the given facts, to solve the problem. Alternatively, the inference engine could check to see which rules if applied could result in a particular solution, a process called backward chaining. Of course, there are other paradigms altogether. Frames, which organize associated concepts into a hierarchy, are often used in systems that make medical diagnoses. Nodes that represent concepts are associated with features that consist of attributes (or slots) and their values[12]. Grouping diseases with their traditional symptoms provides a convenient way of structuring such a system, and the search mechanism mimics the ability of an expert to quickly focus on the relevant information. Rules-based systems that employ forward chaining are generally used in the construction of expert systems for circuit diagnosis[13]. The standard benefits of expert systems are that they work well with heuristic knowledge. Additionally, they may be used as aids or stand-alone systems, depending on the demands of the task at hand. They provide time savings, institutional memory, and a host of other potential benefits.

Fault management for electronic devices includes a variety of steps, as outlined in [14]. Fault detection involves the discovery of the existence of a fault in the system. Fault localization indicates constraining a fault to a sufficiently small subregion of a system. Finally, fault identification refers to naming a fault value for a specific point in a system. This would entail naming the type of fault and pinpointing that fault’s location. This paper describes a system based on “first principles,” or on an understanding of precise circuit behavior, that suggests a testing plan for a VLSI chip to a human engineer. It uses design information, as well as probe measurements, to help the user locate the fault and ultimately identify it. The system primarily relies on generic knowledge of testing and
fault sensitization to construct tests that detect and localize hypothesized faults. As will be discussed later, this project does not include reasoning on first principles, even though the option was considered in the design phase. In [15], a rule-based system that relies on specific knowledge is outlined; the authors present a reasonable way of integrating such knowledge with that gained from querying a user about information he observes. One of the goals of that project was to allow inexperienced users to troubleshoot equipment with a degree of success similar to that of experts. This motivation was incorporated into the author’s system.

Yet another useful system was described in [16]. The paper details some of the pitfalls of using a standard fault model approach to diagnosis. Specifically, it claims that the model/sensitization method is best suited to generate tests that detect faults. However, the study further asserts that the method is not suited for diagnosis. It suggests a method that notes deviations from expected behavior, generates possible reasons for failure, and then tests each of those reasons. This strategy was considered for use in this system, although ultimately most of the knowledge gathered was related to the collection of key facts related to the failure.

A further study attempted focused on an analysis of expert reasoning in diagnosis, a key part of the proposed undertaking[17]. The authors studied the manner in which experts use pattern recognition, model-based reasoning, and a few types of strategies in order to diagnose hardware at the board-level. They specifically outline a system that performs sequential diagnosis tasks, defined as those in which systems must gain additional knowledge about hardware in order to arrive at a diagnosis. That includes determining what tests need to run and what knowledge must be surmised from the
results in order to proceed. Memory troubleshooters go through a similar process, and this methodology was included in the overall decision flow of the system. Additionally, this paper made it clear that when testing hardware, experts often use inexact models, where only diagnostically useful information is part of the knowledge base. The author’s system includes a similarly inexact model of physical adjacency for the memory products it diagnosed, where the system focused on the same criteria that the experts did.

Finally, a survey of previously built expert systems revealed that an expert system to diagnose VLSI memories was created at IBM Bordeaux in 1988[18]. It considers device information, test results, history on defects and vendors, and architecture particulars in arriving at a diagnosis. Much of the focus of the project, however, was on analog parameters such as setup and hold times and more standard faults, which did not apply to task of this project. The system used product history to influence the likelihood of various diagnoses, and this was made possible by the focus on standard faults, since the large volume of modules processed allowed the authors to make statistically valid estimates of the likelihood of various defects. This part of the system was not incorporated into the system described in this paper, because of the limited number of malfunctioning modules that constitute a case history. So, for the above reasons, the closest system to a predecessor did not have the machinery to address the problem at hand. This discussion was intended to point out which features the author’s system considered from previous expert systems in the area of hardware diagnosis.
Chapter 2

Methods

2.1 Knowledge Domain

Designing the system first required a study of the task that the experts perform. So any explanation of the system's design and implementation requires a description of the knowledge used by the expert test engineers and the specifics of their methodology, tasks, and toolsets. This information was obtained through a series of interviews and observation sessions.

While the task of these IBM experts may be generally described as fault diagnosis, there are some specifics about the situation that complicates matters. The defective memory modules they must inspect are initially known to have failed on a customer's system-level test. After the customer refers a module to the department, it is put through a series of tests. The first battery of tests takes place with a module tester. An example might be the walking 1s test that was mentioned in Section 1.1. If the DIMM (Dual In-line Memory Module) does not fail any of these tests, it is then referred to the product engineers responsible for designing the memory at the chip level. They repeat a series of manufacturing tests intended to screen out defects before they ship, as well as some tests that might not be so economic in terms of test time. An example of such a test might be one of the tests to detect k-coupling faults discussed previously. These product engineers run the tests through a number of characterization tests, and only if none of these tests generate a failure with the module at hand do the testing experts commence work on the
module. This means that the module only fails on a system-level test, and it is the responsibility of the testing experts to specify exactly which part of the test resulted in a failure. This means that test engineers are forced to tailor their strategy to deal with the inherent complexity of any system-level test. They must also recommend a screening test based on the defect they diagnose, so that the manufacturing engineers may augment their test suite to improve their quality demands. The test engineers usually receive a limited number of modules that pass the two levels of screening in one year. For those modules that are successfully diagnosed, the average time to a diagnosis is between 4 to 8 weeks.

Because the modules that test engineers examine exhibit failures only under a system test, the experts are forced to handle a testing situation that is somewhat complicated. For example, the cache may interfere with effect any test has on the contents of memory. Additionally, the experts must take care to minimize the extent to which any of their software uses the failing module in the system as memory. To achieve this end, they are armed with a suite of internal IBM tests called Memtest. Written in assembly, these tests are used to detect the failures; experts use their understanding of the tests to pinpoint the relevant sequence of events that caused the failure. A discussion of the Memtest suite would help make some of these issues clearer.

2.1.1 Memtest Specifics

Memtest consists of about 15 tests; each performs a different sequence of operations on memory. Generally, each test writes a particular pattern throughout memory, and then verifies that the data it reads is valid. If the expected and actual data do not compare, a failure is recorded and the corresponding information is written to a file and to the screen of the testing system.
There are a variety of options associated with Memtest that test engineers use when performing their task. The suite can be set to loop in an entirely random mode, where any one of the tests may be selected and a 1 Megabyte segment in memory will be selected, as well. Beyond that, the specific starting address of the test may also be selected at random, and this affects the pattern that the selected test will write in memory. This is so because for some tests, the starting address is used as a random seed to determine the pattern written and because the pattern itself is often repeated throughout the memory. When the test is run in this mode, the operations and contents in memory are exercised in a variety of ways, and the general idea is that this pseudorandom activity could help reproduce the failure that the customer observed. Each iteration of a test within the loop is called a cycle, although each test may consist of multiple parts.

Once the failure has been reproduced, the engineers will try to restrict the test they run, the address space tested, and ultimately will attempt to force the starting address of the test. Knowing that a failure is observed when only one test is run repeatedly provides some information to the experts and they may use this while testing. Reducing the address range increases the number of cycles executed in a given period of time, and this may result in an increased rate of failure. Experts may deduce characteristics of the failure by attempting to correlate any factors they change to an increase or decrease in this failure rate. The reader should remember that the defective module could possess an intermittent fault, in which case this information could prove useful. An example of a defect might be that a resistive short exists between two lines in a way that each access to an adjacent row depletes some charge from the cell. It might be that only if a certain row is accessed a fixed number of times would the failure occur. Observing an increase in
failure rate may spur the experts to hypothesize on such a defect and devise a test to pursue it. Reducing the address space helps achieve this. The engineers may restrict the address range without restricting the test or forcing the starting address to a particular value.

Memtest also gives the user the option of disabling the cache. As long as the cache is in between the memory and processor, Memtest does not have full control over the contents of memory. For example, a write to an address with no substantive physical relationship to the failing address could cause the failure. It might be that Memtest wrote to an address that was close in terms of the system address space and within the block size of the cache. That could trigger the cache to write the contents of that line to memory, thereby exercising the actual failing address at that time. This example shows how the cache makes the test engineers job more difficult. Consequently, they prefer to analyze tests that fail with the cache off. The only reasons they would leave the cache on would be that the failure is only observed when the cache is on or that the testing time, which understandably is reduced when the cache is on, presents an issue for that failure. Since test engineers do process multiple modules at any one time, they must efficiently spend their time. If it is clear that they cannot obtain useful clues about a module, they will shelve it until further information is obtained from the customer. They make use of test functions that limit the running time of the tests. They may limit Memtest on the basis of actual time spent or on the basis of the number of cycles the test executes.

A sample failure log that is printed out to the screen is shown in Fig. 1. The test records the timestamp, the number of cycles completed, the cycle on which each failure occurs. In addition to the failing address, it also records the test that failed and start
address picked for the test. In this particular example, Memtest is run in random mode over the entire address range. The system reports the actual and expected data as well.

<table>
<thead>
<tr>
<th>Application Started... Current Test Started</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles Completed: 1203</td>
</tr>
<tr>
<td>Test in Progress: 002b</td>
</tr>
<tr>
<td>Errors detected: 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Last Errors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
</tr>
<tr>
<td>15:24:19</td>
</tr>
<tr>
<td>15:44:19</td>
</tr>
<tr>
<td>16:25:20</td>
</tr>
</tbody>
</table>

Figure 1. A sample screen display for Memtest

One of these tests writes 0’s to four consecutive addresses, 1’s to the next four addresses, and repeats. Just after writing the data, the system will verify whether this data is actually written there. In the next portion, the system reads the data by skipping across the test’s address range to verify the data in memory again. If at any point, the data expected differs from the data read, an entry in the log is generated similar to the one shown in Fig. 1. This example also illustrates a point made earlier regarding the effect of Memtest’s starting address affecting the pattern in memory. One should note that for this specific test, the contents of memory for the address range of the test will satisfy the equation <k> = <k+8>. In other words, one may increase (or decrease) the starting address by a multiple of 8 without affecting the contents of memory, except for those addresses that are newly excluded (or included). One might reasonably expect for the most part to see no difference in the test results of tests with starting addresses offset by a multiple of 8. A change of any other value will presumably have unpredictable results.

When the experts receive this failure information, they may analyze it in a variety of ways. One such way is to determine whether there is a relationship between the failing
addresses. They may want to know whether addresses are aligned along a particular column or row, for instance. This information is available from the module specifications, and the engineers have a tool to help them with this. The translation requires a scrambling of the bits that constitute the system address. Adjacency in system space does not necessarily correlate to logical row, column, and bank information. It should be also noted that rows and columns that are logically adjacent need not be physically adjacent. Still, noticing patterns in the reported failing addresses is useful to the test engineers.

The experts may decide to use the Memtest information by modifying the tests to force the value of particular cells they think are contributing to the failure. In this fashion, they have the ability to generate hypotheses and modify the software to help them test those hypotheses. They may therefore use these tests in a way that extends beyond the means one could give a traditional expert system.

2.1.2 Logic Analyzer Traces

Test engineers also make use of a logic analyzer to analyze the module’s aberrant behavior. They have equipment which allows them to compare the behavior of the defective module to that of a module which theoretically functions perfectly. By triggering on a Memtest failure, they obtain better insight into what discrepancies cause the failure. They may look for the fingerprints of a known cause of failure, and they also look for events that might be suspicious. For example, certain consecutive commands executed in sequence, such as a refresh to a row just before the failing read, may trigger the test engineer’s suspicions. If they obtain multiple logic analyzer traces they may be able to notice a pattern across the traces. A particular write to a row may always occur
just before a failure, regardless of the address that fails. This sort of correlation would help experts pinpoint the failing mechanism. More examples of issues they examine are timing issues, multiple accesses to the same address, whether the memory controller is bursting.

Although tests from the Memtest suite are fully specified, system involvement may interfere with the exact commands given to memory. Using logic analyzer information, experts may observe exactly what commands are issued and when they occur. This provides a level of information beyond just the Memtest logs. If the experts are able to observe the command that wrote the data to the failing address as well as the failing read that resulted in a log entry, they might conclude that the trigger events or cells are identified in the portion of the trace in between. Given the limited depth of the logic analyzer trace, this makes reducing the address range of the test the best method to try to observe the entire sequence that resulted in the failure. Furthermore, they may alter Memtest to insert a read between those two commands so that they may determine which command on the trace actually triggered the failure.

There is a considerable amount of convention involved simply in reading the traces for particular items. Experts must know how the data is organized in the trace, where to look for values that correspond to system address, and about various timing specifications such as when values are valid for a certain signal. They pay attention to the commands, which include reads, writes, and precharges; addresses, including both rows and columns; and relevant data, on both the defective and theoretically functional DIMMs, to do their jobs. Lastly, they often examine the trace to see if there are accesses to cells adjacent to the failing one.
2.1.3 Layout and Design Understanding

The testing engineers also are able to draw upon their understanding of the chip’s design in order to debug defective modules. The need for the engineers to understand physical adjacency, as it relates to the memory’s cell array, is clear given that most failures somehow involve the concept of physical proximity. Engineers therefore try to determine the contents of cells surrounding the failing one, and often look beyond the four directly adjacent ones. In some cases, a write or read from a cell that was 2 or more cells away caused the failure. Beyond this use of physical understanding, the experts will also use specific knowledge gained by consulting with the physical designers. The designers may confirm hypotheses and help brainstorm with the testing engineers once the engineers have identified the sequence of events that led to the failure. This sort of interactive activity is relatively difficult to capture in a standard knowledge base.

2.2 Knowledge Representation

As mentioned previously, an expert system contains both a knowledge base and an inference engine. After various attempts to characterize the knowledge, a rule-based system seemed the most appropriate choice. Knowledge was extracted through an observation of the experts, interviews on their methodology and reasoning, and generation of a history of recent cases that provided further examples of what they deemed important. Subsequently, those elements of the knowledge domain that could be readily encoded were included in the rule-base, and those aspects that were beyond the reach were left to the user. The experts had a relatively significant role in choosing which parts of the knowledge were included, especially because an expert’s ability to articulate his knowledge links with how well a system could perform the task.
2.2.1. Considering Frames

Initially, the frames paradigm was considered for this purpose. This knowledge representation is typically used in medical diagnosis, where a variety of symptoms are correlated with the appropriate diagnoses. On first glance, it appeared that this might prove useful for IBM’s fault diagnosis methodology. However, there were a few reasons that this was not an ideal match for the task. The first is that such systems often depend on statistically relevant relationships between symptoms and diagnoses. Given the limited number of modules, experts could not reasonably compute such correlations and would be speculating at best. Another critical detractor is that once a particular module is diagnosed, IBM’s manufacturing engineers will place a test that screens out future failures of that variety. Consequently, such modules will never be shipped to the customer and will never find their way back to these test engineers. Experts therefore only tend to see new and different cases. So, the simple matching of previous case characteristics to diagnoses was not a valid strategy.

Even though case generation is optimally suited to creating frames, it is also a useful and accepted way for knowledge engineers to construct rules. By asking the engineers to recall aspects of the testing, the author was able to identify common patterns in the approach to the debugging. So this did prove useful in generating the rule-base. However, the case generation method did not provide an exhaustive listing of all the knowledge the engineers used. As the author quickly found out, the engineers made use of a wide range of knowledge about electrical engineering and device physics in order to ultimately make certain hypotheses. This method of generating and testing hypotheses based on broad knowledge of memory devices could not be captured in the knowledge base. Moreover,
through no fault of the engineers, the thinking behind some of the decisions they made in the past could not be recalled. This underscores a very relevant issue in terms of knowledge extraction for expert systems; even if experts keep strong documentation about their course of action, it may not be enough to reconstruct their thought process as well.

Additionally, when creating the knowledge base, there were issues that exposed the assumptions that test engineers are free to make. If some of those assumptions are disproven by the testing results, they are free to stop pursuing that route and rethink the approach. An expert system does not have that benefit, so relatively thorough thinking was required to ensure that the knowledge base included rules that would arrive at a correct conclusion. If, for example, they suspect that a specific pattern is contributing to the failure and proceed along a course of testing, they can at any point notice that something else might be the case. It was important to delineate the universe of actions that the system could take and those that were best left to humans. Also, in some cases, there were specifics associated with the task that forced more rigidity on the system than the human. For example, engineers, when guessing that a particular test should cause a failure, may run Memtest until they observe a failure and then quit. This interactive feature may not be duplicated by the system considering the way the tests are run (it would have to monitor the screen on the testing system, which has a very minimal setup). So, the engineers had to consider some of those limitations when specifying the system's behavior. In this case, the engineers came up with various metrics to use as limits on the test time in both the number of cycles run and amount of elapsed time.
2.2.2 Knowledge Captured

After the knowledge extraction was completed, decisions were made as to what could be included in the system and how it would be represented. It was clear that experts use various rules of thumb in order to proceed to diagnose a module. For instance, much of their approach rests on the assumption that only one defect mechanism occurs on the chip, even if multiple failing addresses are reported by the Memtest suite. The approximation is that the odds of two unrelated physical defects occurring simultaneously may be safely ignored. This clearly is not a metaphysical certainty, but rather the sort of heuristic knowledge that will arrive at a quicker solution most of the time. Additionally, the experts did focus on certain items very quickly, and it was clear that this intelligent fact-gathering could be incorporated into the system. So, a rule-base format was chosen as part of the system’s implementation.

The specifics of the expert’s testing methodology, as well as their understanding of Memtest, are within the scope of modern-day expert systems. Additionally, the experts’ focus on reading the logic analyzer, specifically because they quickly scanned for concrete patterns, was a close fit for the capabilities of a rule-based system. While the experts themselves employed the ability to draw upon full-fledged model-based reasoning, the sheer amount of information from which they could draw prevented inclusion in the knowledge base. Even though individual facts were gleaned during the observation phase, that knowledge could only be helpful in the diagnosis of those specific defects; once more, the very nature of the job prevents such defects from arriving at the test laboratory ever again. Despite this limitation, it was clear that some understanding of the physical layout would be useful in attempting to simulate the expert’s behavior.
Ultimately, it became clear how the system could include understanding of physical proximity as it relates to the cell array of the memory device. It would have to include cells that were not just directly adjacent, but ones that might be further away. Test engineers do not systematically spill the contents of cells that neighbor the failing one, but in retrospect, the engineers thought their efforts were significantly expedited in a number of cases by doing so.

Another desirable trait in the expert system was module-specific knowledge that experts often had to seek elsewhere. For example, in order to determine whether particular failing addresses were aligned on a row or column, they used software to translate system addresses into logical rows and columns, and in order to read the logic analyzer traces for data written to a particular address, they used similar information. Some of these functions were relatively straightforward, and the only knowledge-level issue they posed involved knowing when to make use of the module specifications. Still, because the activity is error prone, the experts decided that it belonged in the knowledge base. Much of the knowledge required algorithmic processing, which made encoding them in rules somewhat tedious. Performing the scrambling of a bit vector, for instance, is a function that would be tedious to write entirely with the machinery of most rule-base toolkits. So,
as Figure 2 shows, some of the information was kept separate from the primary rule-base even though they represent important knowledge on which the experts rely.

2.3 Design/Implementation

A relatively straightforward inference engine was used as part of the expert system. The Java Expert System Shell (JESS) is a freely available tool for knowledge engineers written in Sun Microsystem's Java language. JESS was designed to interface well with Java, and the latter was chosen as the implementation language.

The system is composed of the knowledge base, which contains 84 rules, and a variety of Java software modules totaling approximately 3,000 lines of code. An example of a rule written according to JESS syntax appears in Figure 3. Each rule contains a precondition and an assertion. The preconditions in this system are sometimes used in a forward chaining manner to control decision flow. Specifically, if the system is to decide whether a fail is reproduced, it checks to see if tests conducted till that point have generated a failing address that corresponds to the logical row or the column supplied by the customer. If this rule fires, it enters a different state, called test-will-proceed, in which the system continues with additional testing. The assertions may also be used to execute function outside the knowledge base. The message command displays an informational message so that the user has a qualitative description of what is occurring, as opposed to only presenting him a series of tests to perform.

```
(if (and (decide-if-reproduced)
  (or (not (no-row-matched?))
    (not (no-column-matched?)))
  (assert
    (message "The fail has been reproduced
      since the failing signature
      (row or column) has been
      matched")
    (setstate test-will-proceed)))
```

Figure 3. An example of a JESS rule
2.3.1 System Structure

After interviewing the experts, the author identified a series of stages in their thought process and methodology. Each stage was intended to reflect an expert practice, and the system as a whole was intended to direct a user through the steps an expert would take to diagnose a particular module. While the system does not necessarily arrive at a final diagnosis, it notes items of importance in the way that an expert would. Figure 4 depicts a full listing of all the states and a diagram of the possible state transitions. The names of the states are relatively self-explanatory and touch on some of the knowledge described previously. The system makes use of all the Memtest options, and it contains specific knowledge about when the experts use the various options to accomplish their tasks. The system makes decisions that may draw on the results of any tests run to that point, and it collects the information from the appropriate Memtest logs. Conclusions that the system may draw are intertwined with its state. Generally speaking, the system pays attention to patterns the experts normally deem important. The failure rate, in terms of the number of failures per test cycle, is closely monitored to determine what factors play a role in causing the issue. Factors include temperature, whether the cache is on, which tests fail most frequently, and the test’s address range. Tables 1 – 4 present a detailed description
of the system’s behavior. They specify each possible state transition and they summarize the actions that the system takes at each step. The reasoning behind the state and the corresponding concern for the testing engineers appears in the third column of each of these tables. The reader may obtain a very comprehensive understanding of the system by examining these figures. The decision flow itself is not altogether complicated, and the document provides a verbal description of the system’s behavior. As one might observe in the rule base (printed in full in Appendix A), much of the detail has been pushed into
State Name | Description / Actions Taken | Corresponding Expert Practice
---|---|---
Initialize-setup | The system gathers important information about the defective module as specified in a document that accompanies a defective module. This includes module type, valid minimum and maximum addresses for any tests, the row and column address of the failure, and failing temperature. The system then enters the state first-test-run. | When experts first receive a module, they note certain failure information and perform some checks to ensure that the testing equipment is configured appropriately.

First-test-run | The Memtest suite is repeatedly run over the entire address range in random mode. In each iteration, a randomly selected test operates over a 1 Megabyte portion of the entire valid address range. The test runs with the cache on for 5 hours, although the user may decide to interrupt if he encounters a substantial failure count. It moves on to decide-if-reproduced. | Before experts may try to obtain information about a failure, they must first be able to confirm that referred module actually fails. The random nature of this test expedites detection of the failure.

Decide-if-reproduced | The system compares the random test results with those observed in the customer test. If the failing characteristics, such as failing row and column, of both match, the system enters test-will-proceed. Otherwise, it enters fail-not-reproduced. | Test engineers must determine whether they can replicate the conditions that allowed the customer to observe a failure. Only after the failure has been reproduced may experts proceed with any diagnosis attempts.

Fail-not-reproduced | The system gives the user the option of exploring other tests even if the specific fail has not been specifically reproduced. If other fails were generated, it may enter test-will-proceed, or else it will exit. | If a module cannot fail according to the customer specifications within a specified time (5 hours of random tests), engineers will work with the customer to obtain more information.

Test-will-proceed | The system now checks to see if there is a failing address below 1 Megabyte. If so, the system advises the user to take steps to avoid potential testing issues and then commences another round of random testing with Memtest. The 5-hour test runs with the cache enabled, and the address range is confined to the address range observed in first-test-run. The system then proceeds to determine-temperature-sensitivity. | Engineers must be careful if failures are generated in the first Megabyte of address space. Since the test software runs in this region, memory failures here could result in unpredictable test behavior. Test engineers also try to increase the number of failures they observe, and they restrict the address range for another random run to do so.

Determine-temperature-sensitivity | The system directs the user to sweep the temperature of the module under test from 10° to 80° C in increments of 10. Memtest is run in random mode for 1 hour at each setting. If the failure across these temperatures varies significantly, the fail is deemed sensitive. The system proceeds to cache-test. | Knowing whether a failure is temperature sensitive helps test engineers hypothesize various failing mechanisms, and they accomplish this by assessing whether the failure rate changes when the temperature of the failing module is changed.

| Table 1. State Descriptions and Corresponding Expert Practices (1-6) |
|---|---|---|

the function calls. Less complexity makes the knowledge base more manageable in that it will be easier to maintain and expand in the future. The real-world success of expert systems hinges on factors such as these. A typical rule of thumb is to keep a rule base
Table 2. State Descriptions and Corresponding Expert Practices (7–12)

<table>
<thead>
<tr>
<th>State Name</th>
<th>Description / Actions Taken</th>
<th>Corresponding Expert Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache-test</td>
<td>The system runs Memtest in random mode with the cache off, and it is restricted to each failing test observed to this point in the Memtest logs. The results of each test (whether it generated failures with the cache off) are recorded. The system proceeds subsequently to analyze-cache-test.</td>
<td>Although having the cache on during testing increases the speed at which tests are run, the cache adds a level of indirection. Since troubleshooting is easier if the memory is directly tested, they prefer to test with the cache off. They consequently pursue first those Memtest tests that generate failures with the cache off.</td>
</tr>
<tr>
<td>Analyze-cache-test</td>
<td>The system prioritizes the list of tests that have failed in the order an expert would investigate them, and continues to select-test.</td>
<td>When multiple tests generate the customer’s failure, experts prefer to analyze first the ones that perform relatively simpler operations on the module. The system follows this priority scheme after the cache prioritization is done.</td>
</tr>
<tr>
<td>Select-test</td>
<td>The next test is selected from the prioritized list of failed tests. If there are no more tests to explore, the system exits. Otherwise, it proceeds to restrict-test.</td>
<td>The experts would prefer to generate any observed failures by running a single test, for ease of analysis.</td>
</tr>
<tr>
<td>Restrict-test</td>
<td>Memtest is next run restricted to the selected test, and the system moves to analyze-restricted-test.</td>
<td>Experts restrict Memtest to individual tests with the hope they may observe failures.</td>
</tr>
<tr>
<td>Analyze-restricted-test</td>
<td>If the restricted test from restrict-test generates failures, then the system will limit all future tests to a set number of cycles (40 * the mean failure rate). It then proceeds to pick-fail. If Memtest does not generate fails when restricted to that test, the system recommends that another test be pursued, returning to select-test.</td>
<td>Even though a certain test may indicate failures when Memtest is run in random mode, it does not mean the test will generate failures by itself. (Consider the case where a prior – and different – test wrote the pattern that exhibits the failure) Test engineers must verify that Memtest fails in restricted test mode before proceeding.</td>
</tr>
<tr>
<td>Pick-fail</td>
<td>If multiple failing addresses have been generated in the course of all the tests to this point, the system picks one to pursue. It performs a histogram on the basis of bank, column, and row, in order to pursue the most frequently failing address. Previously investigated failing addresses are not considered, and if there are no further failures, it proceeds back to select-test. Otherwise, it enters investigate-fail.</td>
<td>When multiple failing addresses are generated, experts have to choose which ones to pursue. The general idea is that the more frequent the appearance of an address in the failure logs, the easier it will be to troubleshoot should the engineer pursue that particular failure.</td>
</tr>
</tbody>
</table>

around 100-200 in size, particularly for a medium-sized application. This was another motivation for the design strategy. The Java modules that implement the function calls vary in their importance to this discussion, although some of them are an essential part of
<table>
<thead>
<tr>
<th>State Name</th>
<th>Description / Actions Taken</th>
<th>Corresponding Expert Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Investigate-fail</td>
<td>After picking a failing address, the system forces a Memtest run to start at an address that generated the entry in the log. If forcing the starting address successfully duplicates the desired failure, the system moves to squeeze-back-end. If not, the system enters suppose-background-contribution.</td>
<td>The general idea behind forcing the starting address in Memtest is that the starting address typically determines the pattern repeating throughout memory. If this test generates failures, experts know that the failure can be reliably recreated, and may proceed with analysis.</td>
</tr>
<tr>
<td>Suppose-background-contribution</td>
<td>This state checks to see if a simple background pattern can affect the failure, since we know that forcing on a previously observed starting address does not recreate the failure. The system first tries to recreate the desired failure using the test parameters that initially generated the failure. If that test does not fail, the system lets the user choose between pick-fail and exiting. If the test does fail, the system writes a simple background pattern to see if that pattern makes the failure disappear. It then enters analyze-background-test.</td>
<td>If forcing the starting address cannot reliably recreate the failure, experts will hypothesize that something trickier is happening. For example, the fault could have a complicated pattern-sensitivity. They then try to verify their suspicion by writing simple background patterns in memory to determine whether that may make the failure disappear. If those attempts are not successful, they will pursue a diagnosis using knowledge not captured in the expert system. (Such as changing the test software to test model-based theories)</td>
</tr>
<tr>
<td>Analyze-background-test</td>
<td>The system checks if the patterns had any effect on the failure, and which ones. It then reports its findings on certain pattern sensitivities. Then it allows the user to either proceed to select-test or exit to pursue a strategy beyond the system’s means.</td>
<td>It is useful for engineers to know whether a particular pattern, such as zeroing a range of addresses, modulates a failure. This can be used as a starting point for determining whether certain cells are coupled to the failing one and what data those cells must contain.</td>
</tr>
<tr>
<td>Squeeze-back-end</td>
<td>The system tries to reduce the range of addresses over which the chosen failing test runs. It does so by maximally reducing the distance between the ending address and the failing address while still getting the test to fail. This search is performed in a binary fashion, and the system reports the cutoff address. Then, it proceeds to squeeze-front-end.</td>
<td>Reducing the address range serves two purposes. First, it eliminates from suspicion numerous cells to which the failure could be sensitive. (If the failure occurs even when certain cells are not exercised, it could free them from suspicion). This practice potentially allows the entire test sequence to be captured on a logic analyzer trace and studied carefully.</td>
</tr>
<tr>
<td>Squeeze-front-end</td>
<td>After the ending address has been pushed as far as possible, an analogous process begins for the start address. In this mode, the system makes sure not to reduce the address range of the test below 512K if the test fails only with the cache on. The system then enters determine-forward-increment.</td>
<td>Reasons for moving the start of the test address range parallel those of moving in the ending address. Engineers must be careful not to reduce the address range below the size of the cache if it is enabled, since test results would be unreliable (engineers would not know if results reflected the contents of memory or those of cache)</td>
</tr>
</tbody>
</table>

Table 3. State Descriptions and Corresponding Expert Practices (13–18)
<table>
<thead>
<tr>
<th>State Name</th>
<th>Description / Actions Taken</th>
<th>Corresponding Expert Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determine-forward-increment</td>
<td>The system looks up the appropriate increment that corresponds to the Memtest test that is currently being explored. This ensures that the starting address is moved in only by a multiple of the increment, so that the pattern in memory will remain the same in the new test. The system moves to <strong>perform-forward-scrunch</strong>.</td>
<td>The starting address often determines the pattern written in memory by any test from the Memtest suite. Each test writes a repeating pattern, so there is an offset from the initial starting address for which the general pattern remains the same. The engineers must remember to always shift the address by a multiple of this value.</td>
</tr>
<tr>
<td>Perform-forward-scrunch</td>
<td>The system performs a binary search similar to the one in <strong>squeeze-back-end</strong>. However, when dividing the address range, it takes into account the offset issue with the starting address and adjusts the starting address to number offset from the original starting address by a multiple of the increment value. The system reports the cutoff address and then goes to <strong>read-logic-analyzer</strong>.</td>
<td>The engineers perform the forward scrunch to see if they may possibly narrow the address range of the test to as little as a few hundred addresses. With an analyzer trace recording this information, they could then deduce why the failing module is malfunctioning, or even modify the Memtest software to verify a hypothesis.</td>
</tr>
<tr>
<td>Read-logic-analyzer</td>
<td>The system requests a trace of the Memtest test in question. It then reads the trace, looking for a variety of issues, and reports on whether it found any of those items in the trace. Such items include a TRP = 2, looking for read/writes to cells physically adjacent to the failing one, excessive bursting, or suspiciously placed refreshes. The system then enters <strong>finish-analyzer-trace</strong>.</td>
<td>The engineers screen look through the trace to see if there are any clues present. If, for example, they can see a write and a read to the failing address on the same trace, they will attempt to isolate that sequence of commands to arrive at a diagnosis.</td>
</tr>
<tr>
<td>Finish-analyzer-trace</td>
<td>The system asks whether the user would like to investigate another fail by proceeding to <strong>pick-fail</strong>, choosing another test in <strong>select-test</strong>, or concluding.</td>
<td>Experts will decide if they have enough information or whether they need to perform additional investigation. This system leaves this choice to a human.</td>
</tr>
<tr>
<td>Exit</td>
<td>The system concludes and prints out a log of any messages it generates, instructing the user to take the logs to an expert for final analysis. As such, nothing happens in this state, although knowing when to transition to it is critical.</td>
<td>Knowing when to finish an investigation is important, either because a satisfactory amount of information has been obtained, or because it is wise for the expert to give up and spend his time more usefully.</td>
</tr>
</tbody>
</table>

Table 4. State Descriptions and Corresponding Expert Practices (19–22)

the application. These modules are described in the following sections.
2.3.2 Physical Adjacency Understanding

The experts will sometimes attempt to reconstruct the contents of cells that are physically adjacent to the failing one, and they are usually interested in the values of these cells at the time the failure was reported. Since Memtest does not report these, the engineers must do so by obtaining a logic analyzer trace of the test by triggering on a Memtest failure and examining it carefully. They must use specific knowledge of the chip’s layout, the module’s specifications, and their understanding of how to look for those things in the logic analyzer trace to perform this task. The test engineers stated during the design phase that it would be useful to know the contents of a 7x7 cell grid that is centered on the failing cell. It should be noted that the system’s method of considering adjacency only functions for IBM’s 64 Megabit Synchronous DRAM product.

From the Memtest log file of a failure and a trace of the test, the system generates the signature for each of the cells in the grid so that it may find the contents of that cell in the trace. The next subsection will detail how the system actually reads the trace; assume for the purposes of this discussion that single cell may be located in the trace with values for its bank, logical row, logical column, and the appropriate bit on the analyzer’s pods. From the Memtest log file, the system maps the system address to the failing cell’s bank, logical row, and logical column. The bank of all the adjacent cells is the same as that of the failing one. The logical row and column may subsequently be translated into physical row and column space by an appropriate mapping. The point of doing so is that in physical row and column space, consecutive rows and columns are physically adjacent. That is to say that physical column 8 is adjacent to physical column 9.
Once this information is generated for the failing cell, the system will then determine the relevant information for all other cells in the grid. Figure 5 illustrates this process.

One crucial point in understanding the diagram is that for the 64 Mb DRAM product, pins on the chip package are associated with internal adjacency in groups of 4. Specifically, those pins always contain adjacent data and the exact pattern of pins may be determined from the cell’s general location in the chip. For this area in the chip, the pins 2, 6, 3, and 7 are adjacent to each other. Generating the appropriate values for one of the failing cell’s neighbors will illustrate how this information is used. The cell signature for the failing cell’s neighbor to its north would be pretty straightforward. Its physical column would be the same as the base cell, and its physical row would be the base cell’s physical row minus one. Its memory chip pin would be 3, which the system can map to the appropriate module pin and analyzer pod bit. The physical row and columns may then be mapped back into logical row and column space. The system may then look for any writes to that cell in the analyzer trace; it repeats this process for each cell in the grid. The translation from logical column to physical column is simply a matter of scrambling the column bits correctly. In this fashion, the system is able to make use of information about physical adjacency issues.
2.3.3 Examining Logic Analyzer Traces

The system is able to read a logic analyzer trace and search for some of the things that would interest a test engineer. A brief discussion of the items that appear in the trace might prove useful. Engineers monitor the commands issued to the memory, whether it is a write, read, bank activate, or a precharge command. A data bit mask signal informs the viewer about which bits on the chip are being read or written, and depending on its length indicates whether the memory controller is fetching data from the module in bursts. Engineers may determine what is written by a particular command by looking at the address bus given the appropriate timing specification. A clock signal plays a role for obvious reasons, as does the data bus.

Stepping through one example of reading the trace will make matters more concrete. Figure 6 shows a sample trace and how one would proceed to look through the trace for a read from the address on bank 0, column 0x74, and row 0x75. A bank activate begins any chip cycle, and after this, read and write commands to that bank may occur. The row is presented on the address bus at the time of a bank activate, and column information is presented to the memory at the time of a read or write. Checking the data bit mask signal DQM one cycle after the read command informs the viewer that all bytes are being read for this command, and its length indicates the length of the data burst. Lastly, one needs to look at the appropriate byte (normally eight bytes are recorded at a time by the module) for the data bit in which the viewer is interested after the read latency, which is 3 clock cycles. This is one example of something for which the system can search.
Figure 6. Stepping through a logic analyzer trace, looking for a particular command

The system looks for writes and reads to the cells adjacent to the failing one and generates a list of all such accesses and marks them with a timestamp. It also prints out the contents of the 7x7 grid detailed in the previous section, leaving blank those cells to which it did not observe an access. In addition, the system examines a timing issue that can aggravate a signal margin in a Synchronous DRAM. Another issue relates to a refresh termed CAS-before-RAS which refreshes an address that is specified by the chip internally. Experts are sometimes suspicious of these commands, especially if they appear near a critical read or write to the failing cell, since the address affected is not explicitly mentioned. The system also checks to see if the failing read and the write to the failing cell may be observed on the same trace. If this is the case, then engineers may surmise with relatively strong confidence that one of the commands between the two contributed, even if only partially, to the failure. The system leaves it to the user or
experts to modify the Memtest software to isolate and verify exactly which command causes the failure. Engineers also monitor whether there is excessive bursting present in a trace, since there are some conditions for which bursting would prompt a failure. In order to deploy the system, a specific logic analyzer configuration was used for the sake of convention. Ordinarily, different experts may configure their equipment differently according to their preferences and which information they feel is most relevant. All told, the functionality associated with reading the logic analyzer constituted about one-third of the total project in terms of code size.

2.3.4 Remaining Modules

Of the remaining modules, only the address translation and log reading portions contained information that could be thought of as expert knowledge. Reading the Memtest log was, for the most part, a matter of conventions. However, there are a few issues related to Memtest that were noteworthy. For example, the Memtest software reports an error in a particular address by including the base address, the expected data, and the actual data. The expert computes the offset from the base address in order to determine which bit is failing. It is possible that the base address could differ in another entry yet still represent the same failure, presuming the data is offset by the appropriate amount. So, when the system has to pick a unique failing address as part of the pick-fail state, for instance, it must take this discrepancy into account. The system’s implementation involves a plethora of such details, many of which are impractical to list in a document of this nature. The address translation software simply maps system addresses and offsets to row, bank, column, and module data bit information.
A small but relevant part of the design is its interactive message window. Each state has a useful message associated with it; the intent is to help guide a novice engineer through the process that an expert would take. The user is required to acknowledge these messages, some of which are important for the user to really understand what the system is doing and why it is pursuing that route. In some cases, the user (with the help of an expert) may decide what route to take. An expert may sometimes feel that enough data has been collected, and the system presents the user with an option to quit if she feels this is the case.

An example of this message window is shown in Figure 7. The system first recommends a particular test to run. The user is then asked to copy the results of the corresponding log file into the appropriate directory, and then is asked to choose it using a file dialog. The system determines that the failure has been reproduced and gives the user a reason why it is pursuing further tests; in the figure, it is frozen as it waits for an acknowledgment. Incidentally, this example involves a transition from decide-if-
reproduced to test-will-proceed. All messages printed to this window are also stored to a log file, so that the user has a comprehensive record of the system's decisions.

The system supports four module types, and the key issue there is the address translation functionality. The IBM designations are UAEQ, UAJW, GECP, and GECN. It is also restricted to IBM's 64 Mb generation of memory. This is because the physical adjacency portion uses a scheme which is only correct for that generation. The general methodology encapsulated by the state diagram, however, would be applicable to other generations of IBM memory products. Additionally, the system supports all versions of the 64 Mb generation, including the 4-pin, 8-pin, and 16-pin varieties.

2.4 Testing Setup

In order to determine whether the system performed successfully, equipment was arranged so that it could order tests on a defective module and be supplied the actual results of those tests. The goal in terms of success was to measure how much the system agreed with the approach taken by the experts. Once a sizable and representative sample of decisions had been achieved, the plan was to ask the experts to score each decision on a scale from 0 to 10, with 10 representing full agreement.

The testing equipment included a fully functional machine dedicated to running the expert system. Additionally, the department procured a machine to test the defective module that was similar to the one on which the customer observed a failure. The expert system could not be run on the testing machine itself for a variety of issues, one of which is that the system uses the defective module to store program data and instructions. In addition, a module of the UAEQ type was randomly selected from the assortment shipped to IBM. The logic analyzer configured was a Hewlett Packard 16500B model. It should
be noted that any tests requested by the system required the operator to type the test command into the testing machine and manually transfer any files to the expert system machine. Figure 8 displays the setup quite accurately.

![Figure 8. Expert System/ Test Setup](image-url)
Chapter 3

Results

The system generated over 350 decisions in its investigation of the test memory module. For the most part, the module failed virtually every test that the system requested, and so the system logged thousands of failures. This exercised the various components to the extent that it verified both the high-level functionality and the implementation details of the system. Finally, every state from the histogram generation to the logic-analyzer functions was exercised at least five times. The tests were run over the course of three working days, with the total test time, including manual log transfer and the capture of 5 analyzer traces, lasting approximately 20 hours.

After the system was tested on a module and the functionality of the results verified, the experts reviewed the message log printed by the system in order to assign a score to each decision. The experts then rated each decision and critiqued the system’s performance on the basis of whether the correct decision was made and how well a novice user could understand the thought process behind the system. Both experts rated the majority of decisions a 10, as one may observe from the summary of results in Table 5. The average score over all the decisions was a 9.8. The table displays the mean score for each overall category, and in some cases, subscores were tabulated. The system investigated two failures through Memtest 1 and one failure through each of Memtest 2 – 4. The results reflect decisions made across those tests since they were grouped by state, not on the basis of the iteration. All of the categories should be familiar to the reader from
Tables 1 through 4 except for the decision type called Test Anomaly. This particular flag was raised during the cache-test phase for Memtests 5 and 4. In some cases, these tests generated anomalous output that the system detected correctly.

<table>
<thead>
<tr>
<th>State</th>
<th>Decision Number</th>
<th>Score</th>
<th>State</th>
<th>Decision Number</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize-setup</td>
<td>1</td>
<td>8</td>
<td>Pick-fail</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bank histogram</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Column Histogram</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Row histogram</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>First-test-run</td>
<td>1</td>
<td>8</td>
<td>Investigate-fail</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Test parameters</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fail recreated ?</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Decide-if-reproduced</td>
<td>2</td>
<td>10</td>
<td>Squeeze-back-end</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Test 1 – 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test-will-proceed</td>
<td>2</td>
<td>8</td>
<td>Failure rate decreased?</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Code Interaction test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Determine-temp-sensitivity</td>
<td>9</td>
<td>10</td>
<td>Cutoff Address</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Test 1 – 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache-test</td>
<td>10</td>
<td>10</td>
<td>Squeeze-front-end</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Cache test parameters</td>
<td>32</td>
<td>10</td>
<td>Test 1 – 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test anomaly</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyze-cache-test</td>
<td>1</td>
<td>10</td>
<td>Determine-forward-increment</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Cache prioritization</td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simplicity ordering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select-test</td>
<td>4</td>
<td>10</td>
<td>Perform-forward-scrunch</td>
<td>56</td>
<td>9</td>
</tr>
<tr>
<td>Test 1 – 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restrict-test</td>
<td>4</td>
<td>10</td>
<td>Read-analyzer-trace</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Test 1 – 4</td>
<td></td>
<td></td>
<td>Read failing cell</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Write failing cell</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CBR Status</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 accesses in a Row</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Trp of 2</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bursting percentage</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Grid contents/Final</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Values</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Marker</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Analyze-restricted-test</td>
<td>4</td>
<td>10</td>
<td>Finish-analyzer-trace</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Test 1 – 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Breakdown of expert decision scores by state and category.

The experts expressed little disagreement on the paths chosen by the system, but felt that in some instances the system should have explained its choices to the user in a clearer and more prominent manner. One example of this was the binary searches that reduce the address range from the starting and ending sides. The system printed out a
variety of tests to perform the said searches, but ended the search without explaining to the user the reason for doing so. For the most part, though, the system received high agreement ratings.

Another way to assess the system’s performance is to consider the end result of the system’s testing. The system successfully narrowed the address range of four of the Memtests that failed to within 100 addresses while still observing the failure, and the user chose to halt before investigating any other tests. This success in address range reduction and corresponding increase in failures per cycle ultimately resulted in the logic analyzer capturing a write to the failing cell, as well as the failing read. This means that users of this tool would achieve a result that is often the goal of the test engineers. Though any further analysis through code modification is beyond the reach of the expert system, getting that far alone is quite useful. Additionally, since the system also noticed that it could get the module to fail with the cache disabled, it would eliminate one layer of complexity for a test engineer that continued where it left off. The system further noted when reconstructing the cell contents that the vast majority of the cells in a 7x7 neighborhood of the fail did not appear in any of the five traces. All of these factors bolster the case for an expert system significantly reducing the time to diagnosis. The system could, at the least, know enough to give a human engineer a head start in debugging the module, even if it could not perform a diagnosis single-handedly. A final point is that the system arrived at a significant milestone in the engineers’ diagnostic process in 20 hours of testing time, which is considerably less than it usually takes. However, this module did fail most tests at a higher number per cycle, which may mean the shorter time could be result of the module’s particular characteristics. In any case, it
would be valid to argue that the system could provide time-savings on the whole by freeing experts to spend their time on more challenging cases as opposed to gather information the system could do as well.
Chapter 4

Discussion

One of the purposes of this thesis was to explore whether expert systems could help diagnose a class of memory faults not addressed by standard fault model and testing literature. To that end, it appears that given the appropriate expert background, it is quite feasible to encode a significant body of knowledge that testing experts use to perform fault diagnosis on exceptional cases. More specifically, the project attempted to determine what knowledge representations might be best suited to model the behavior or thinking of these troubleshooters; from the results, it appears that a rule-based system may certainly succeed in capturing some of the heuristic strategies and assumptions they make. Moreover, an expert system may certainly serve as an accurate repository for expert strategies and other task specific knowledge, such as how to employ various tests and make simple deductions from them.

It should be noted that there are a variety of reasons that made this subject a less than ideal one for the purposes of designing an expert system. One difficulty was that the experts had difficulty at times – and understandably so – in articulating various rules. When this is the case, observation and generating case histories are the only resort and certainly are not as efficient a means of knowledge extraction as having an expert dictate them. A level of maturity or stability in a field makes it a more suitable application for expert systems; one often hears of standard applications, such as systems that successfully solve intermediate chemistry problems or provide standard auto repair. Most
such fields have the property of stability. Unfortunately, the memory manufacturing and testing business, as with the rest of the semiconductor industry, creates a new generation of products and associated problems about every 18 months. This may make maintenance of a full-blown memory fault diagnosis system an almost annual and ceaseless re-design effort. Lastly, a system that attempts a full diagnosis on its own may not be possible since the experts may have great difficulty diagnosing a module as much as 50% of the time.

Given the constraint of having to diagnose this failure with a computer system (since that is the only way it fails), this can be quite a remarkable record. Still, expert systems are best equipped to handle routine decisions where the success rate is much higher. Despite issues that detract from the feasibility of an expert system in this field, it is encouraging to note that a significant amount was achieved anyway.

Attempting to duplicate the reasoning the experts use was unmanageable given current knowledge paradigms and toolsets. After characterizing the knowledge domain extensively, it became clear that experts are able to recognize entirely new patterns and devise new hypothesis to explain things they observe. This behavior sometimes leads them in the correct direction, and often times it does not. Even if generating arbitrary hypotheses were possible, it is impossible to capture the broad base of knowledge that allows them to confirm or disprove their hypotheses. The latter conclusion is certainly not much of a revelation, but any knowledge engineers wishing to tackle this subject must understand the degree to which troubleshooters of any kind rely on such practices. Because there is no strong paradigm to address this part of fault diagnosis, knowledge engineers in this area should focus on encoding any general strategies that experts in the field use. In this project, a rule-based system served well to achieve this task.
Many of the benefits associated with expert systems can be gained in this subject area despite the inability to capture the human parts of the job. Numerous tasks require specific knowledge about devices and circuits that experts may not be able to recall quickly and accurately. For instance, an expert might not easily generate the contents of cells that neighbor the failing one, since the individual might forget how adjacency is defined in that region of the device. Automating these types of tasks may eliminate potential errors that could slow down diagnosis.

Another benefit that expert systems often provide is the ability to share expert knowledge with novices. One of the purposes of this project was to create a system that could lead an inexperienced user through the general strategy the experts use. This could be a useful way to train incoming individuals. This example shows that one could approach this problem just from the point of view that this is an interesting expert system application and determining what features of these system may be exploited best when dealing with fault diagnosis.

The system’s relative success with information gathering and preliminary decision-making could be used to reduce the time to a diagnosis and allow more efficient use of engineer’s problem-solving ability. Although the user must manually run Memtest tests and transfer test logs to the system, if this were automated, the system could be employed effectively as a screening tool that automatically points users, even expert users, in the right direction. Additionally, the system could easily be extended to support all testing systems and module types to make it more useful as a screening mechanism. The methodology captured in the rule-base could be applied to other generations of IBM memory products, but physical adjacency knowledge for new generations would require
writing that part of the system from scratch. All in all, this project could serve as the foundation for more work in this area, and a relatively powerful system might be developed if more effort were expended on those parts of the knowledge domain that were most easily captured and bore most fruit in this version.
Appendix A

Rule Base in JESS

//In initialize-setup, the system gathers information about the defective module that is usually contained in a referral toetag. After gathering that information, it proceeds to a first test

(if (initialize-setup)
  (assert
    (message "Please complete the specified module information and click OK") get-toetag-info)
// First, get initial data (including temp, row/col, test that // failed, also get address range information from user for first // test (Beg. Addr, Ending Addr.) get module type ("UAJW, GECP, // GECN") From that infer x8 or x4 - UAJW-x8// GECP, GECN - x4, // slot of failing module (0-3)

//Note: Failing temperature is given as low, ambient, or high, temp is // string
/toetag.temp, num is the number 10, 25, or 80
  (message "Please set the temperature of the device to that specified by the toetag," toetag.temp ". You must actively heat or cool the device to room temperature if the failing temperature is ambient.")
  (set-temperature) // tempset = toetag.temp

//the following is a reminder message to disable ECC
  (message "A common mistake made by engineers is to leave ECC enabled. Please disable it now because that will hide failures")
  (setstate first-test-run))
// leave this mode now that the initialization is complete.

// This mode runs Memtest in random mode across the entire address range // for five hours to see if it can generate failures. The MAX-time is the // amount test engineers specified as the minimum necessary to continue // investigating that module

(if (first-test-run)
  (assert
    (message "The first test run will randomly pick from all the tests and operate over the entire valid address range")
    (run-random-test) // run Memtest with current-test = all, // CacheOn? = true, Timeout? = true, Time = 05:00:00
    (setstate decide-if-reproduced)))

// Generally, we are now comparing test results to toetag info
we now check to see if the test specified by the toetag was
reproduced
from interview notes, reproducing test isn't necessary but helpful
so it gets a reminder

(if (and (decide-if-reproduced)
  (test-different?)) // the test specified in toetag.test did not
  // fail
  (assert
    (message "The test specified in the toetag, " toetag.test "
did not generate a fail, although this is not
crucial to reproducing the fail")))

(if (and (decide-if-reproduced)
  (not (test-different?)))
  (assert
    (message "the test specified in the toetag generated a
fail")))

(if (and (decide-if-reproduced)
  (no-row-matched?) // all failing Rows are different from
  // toetag.row, Columns different
  (no-column-matched?)) // from toetag.column
  (assert
    (message "Because none of the rows or columns of any fails
matched those given in the referral toetag, the
fail has not been reproduced.")
    (setstate fail-not-reproduced))
  //note on details: if list of fails is nil, then each boolean test
returns false

(if (and (decide-if-reproduced)
  (or (not (no-row-matched?))// any row is same as toetag row or
  (not (no-column-matched?))) // any column is same as toetag
  // column
    (assert
      (message "The fail has been reproduced since the failing
signature (row or column) has been matched")
      (setstate test-will-proceed)))

(if (and (fail-not-reproduced)
  (other-fails-generated?))
  (assert
    (message "While the fail has not been exactly reproduced,
other fails were generated Do you want to pursue
them?")

  // Now, prompt the user to ask if we should proceed anyway

(if (and (fail-not-reproduced)
  (other-fails-generated?)
  (clicks-yes?)) // user clicks "yes" button then testing will
  // continue
  (assert
    (message "You have chosen to investigate other fails that
failures that

(setstate test-will-proceed))

(if (and (fail-not-reproduced)
  (other-fails-generated?)
  (not (clicks-yes?))
  (assert
    (setstate exit)))
  // user clicks "no" button then we exit

(if (and (fail-not-reproduced)
  (not (other-fails-generated?)))
  (assert
    (message "The fail has not been reproduced, and since no other failures were generated in the maximum allotted time, this module should be shelved and revisited later")
    (setstate exit)))

(if (and (test-will-proceed)
  (fails-below-meg?) // searches through any logs not marked // ignore-for-buddy and returns true if // there are fails < 1Meg
  (assert
    (message "Please insert a buddy DIMM. Errors were detected in the first 1 Meg of address space, and this may cause interactions between test and code. Once you have done so, please enter the new slot # of the failing DIMM and click")
    (request-new-slot) // textfield & button, overwrites history, toetag // failing slot, ret integer
    (mark-logs-ignore-buddy) // marks logs till this point as ignore-for-buddy test so
    (setstate first-test-run))) // we don't count fails before the // switch next time through

(if (test-will-proceed)
  (assert
    (message "Limiting the address range some generally tends to increase the fail rate, which in turn, helps provide more data in a reasonable amount of time.")
    (run-tight-random-test)// run Memtest for 5 hours over range // earliest SA to last fail rounded // up to nearest meg
    (setstate determine-temperature-sensitivity)))

(if (determine temperature-sensitivity) // start at zero, increase the // temperature 10 degrees at a // time from 10 to 80
  (assert
    (message "One point of interest to test engineers is determining if this failing module is sensitive to temperature changes. This information may prove useful in helping them brainstorm on a failing
mechanism.

(perform-temp-test)  // and see if failrate varies
  // by more than a factor of five,
  // it is sensitive

(setstate temp-test-done))

(if (temp-test-done)
  (assert
   (message "The next step is to assess which tests may be made
to generate failures with the cache off. In
general, investigating fails with the cache off is
a better situation than investigating fails with
the cache on. This reflects the fact that taking
the cache out of the picture indicates that the
tests are directly running and memory, and
experts can use this to reliably to recreate the
sequence of commands that led to a fail.")

(determine-cache-sensitivity)  // for each test using bounds
  // of tight-random-test, mark
  // fail, no fail in a table
  // table indexed by test #

(setstate cache-test-done)))

(if (and (cache-test-done)
  (one-column-failed?))  // sets single-column to value if true
  (assert
   (message "All the failures to this point are on the same
column," single-column-fail ".")

(if (and (cache-test-done)
  (one-row-failed?))  // sets single-row to correct value if true
  (assert
   (message "All the failures to this point are on the same
row," single-row-fail ". This is not a standard
occurrence and it would be difficult to pinpoint a
mechanism in this case.")

(if (and (cache-test-done)
  (all-cache-table-entries-same?))  // there are only cache on tests
  // or only cache off tests
  (assert
   (message "Since there isn't a mix of tests that fail with
   cache off and those that fail with cache on, the
tests will be investigated in the order of
   simplicity.")

   (prioritize-list)  // create test-list
   (setstate select-test)))

(if (and (cache-test-done)
  (not (all-cache-table-entries-same?)))  // there's a mix of cache-
  // on tests cache-off tests
  (assert
   (message "In order to prioritize a list of tests that fail
   with cache on and cache off, the cache-off tests go
first, and within the groups, one applies the standard prioritization scheme.
(prioritize-mixed-list) //apply prioritization to cache list
(setstate select-test)))

(if (and (select-test)
 (not (empty-test-list?)))
 (assert
 (modify-test-list) // pops test list, sets current test
 (set-current-test)
 (modify-tried-test-list) // add the current test
 (setstate restrict-test)))

(if (and (select-test)
 (empty-test-list?)) // check if vector is null
 (assert
 (assert
 (message ("There are no tests remaining on test-list that can be explored")))
 (setstate exit)))))

(if (restrict-test)
 (assert
 (message "Restricting test to " current-test ".")
 (run-restricted-test) // Memtest, restricttest = true, test= // current- test,address
 (setstate analyze-restricted-test))) //range same as tight-
 // random-test, timeout = 1 hour Analyze restricted // test starts the process of compiling results
 // from restricted tests

(if (and (analyze-restricted-test)
 (failures-generated?))
 (assert
 (set-cycle-rate) // calculate Mean Time to Failure in last
 // test and make 40* that the cycle rate
 // from now on
 (setstate pick-fail)))

(if (and (analyze-restricted-test)
 (not (failures-generated?)))
 (assert
 (message "When the restricted test was run, it did not generate failures, so another test must be selected."
 (setstate select-test])))

(if (pick-fail)
 (assert
 (message "A failure must now be selected for investigation.")
 (create-total-failure-list)// make vector of all failure
 // objects so far, initialize a // list of failures we've
 // already checked
 // (tried-fails = nil)
 (make-histogram-list)))

// create HistogramList of failures by copying all failures
// in test's history except for those which match signature
// of failures in tried-fails for an address to match, the
// banks, rows, columns, adj delta, and 8-aligned address
// must be equal

(if (and (pick-fail)
        (empty-histogram-list?))
  (assert
   (message "All failures in this test have been investigated.")
   (setstate select-test)))

(if (and (pick-fail)
         (not (empty-histogram-list?)))
  (assert
   (message "A fail will be picked by creating a histogram on
   the bank, column, and row dimensions.")
   (make-bank-histogram) // select the bank which has most
   // fails
   (make-column-histogram)// select the column which has the
   // most fails
   (make-row-histogram) // select the row which has the most
   // fails, returns first failure
   // object in vector to selected-fail
   (add-fail-to-tried-fails) // copy the column, row, and
   // adjusted address
   (setstate investigate-fail))) // adjusted delta to tried-
// fails

(if (investigate-fail)
  (assert
   (get-test-info) // gets test options associated with
   // selected-fail, such as starting address,
   // ending address cacheon = opposite of
   // lookup in cache-table, cycles = same as
   // global cycles, limit = true timeout=false
   (message "An attempt will be made now to reproduce the fail
   by forcing the starting address")
   (forced-test))) // forced starting address, (restricted test
   // and cycles, still)

(if (and (investigate-fail)
        (failures-generated?))
  (assert
   (set-cycle-rate) // calculate mean-failure-rate
   (setstate squeeze-back-end)) // scrunch the address range
   // of failing test

(if (and (investigate-fail)
         (not (failures-generated?)))
  (assert
   (message "Because test conditions were duplicated and the

53
fail could not be reliably reproduced, it is possible that background data is contributing to the fail"
(setstate suppose-background-contribution)))

(if (suppose-background-contribution)
  (assert
    (get-all-test-info) // lookup the test info associated with
    // this fail restricted address range, restricted test, cacheon, hour
    (message "In order to test whether background data contributes to the failure, the procedure is to repeat test conditions known to have caused failure and see whether the fail can be made to disappear by altering the background data")
    (run-old-test))) // run old-test to see if we can get the fail again

(if (and (suppose-background-contribution)
  (not (failures-generated?)))
  (assert
    (message "The restricted test does not fail repeatably, so I suggest giving up on this fail. Do you want to continue with another fail?")
    // put up yes/no box now

(if (and (suppose-background-contribution)
  (not (failures-generated?))
  (clicks-yes?))
  (assert
    (setstate pick-fail)))

(if (and (suppose-background-contribution)
  (not (failures-generated?))
  (not (clicks-yes?)))
  (assert
    (message "You have chosen to exit")
    (setstate exit)))

(if (and (suppose-background-contribution)
  (failures-generated?))
  (assert
    (message "A simple background pattern will be run to see if it affects the failure.")
    (write-zeros) // by running Memtest #24 over
    // toetag.staddress to toetag.endaddress
    (get-all-test-info) // this gets that same test info again
    (run-old-test)
    (write-ones) // by running Memtest #25 over toetag.
    // .staddress to toetag.endaddress
    (get-all-test-info)
    (run-old-test)
    (setstate analyze-background-test)))

(if (and (analyze-background-test)
  (zeros-test-failed?)

54
(ones-test-failed?))
(assert
  (message "There is no effect for writing ones and zeros in
  other portions of the address space")))

(if (and (analyze-background-test)
  (zeros-test-failed?)
  (not (ones-test-failed?)))
(assert
  (message "There is a bit in memory, that when set to
  one causes the failures to disappear")))

(if (and (analyze-background-test)
  (not (zeros-test-failed?))
  (ones-test-failed?))
(assert
  (message "There is a bit in memory, that when set to zero
  causes the failures to disappear")))

(if (and (analyze-background-test)
  (not (zeros-test-failed?))
  (not (ones-test-failed?)))
(assert
  (message "The results of the background-test are confusing,
  because the fail disappears in both case. Perhaps
  this particular test is unrepeatable.")))

(if (analyze-background-test)
  (assert
    (message "Do you want to continue with another test or
    quit?")))

(if (and (analyze-background-test)
  (clicks-yes?))
  (assert
    (setstate select-test)))

(if (and (analyze-background-test)
  (not (clicks-yes?)))
  (assert
    (setstate exit)))

// Binary search over the space of the test
// first squeeze in the back end binarily
// note that these are known RestrictTest? TestValue, limitcycles?,
// cycles
// Cacheon? RestrictAddress? = no
// ForceAddress = yes StartAddress is same, EndAddress is same

(if (squeeze-back-end) // runs tests, returns squeezed-end-address,
  (assert
    (message "The goal is to reduce the address range and hence
    narrow the space of events and addresses that

55
could be affecting the failure. First, we binarily move in the back end and see if a sensitive cell becomes apparent. Moving in the back end does not affect the pattern written to memory."

(move-back-end) // performs the min-max algorithm and binary search, returning a cell value, calculates MTTF and compares it to cycles 40, if at any point it is less than a factor of five, it prints a message out to the user

(setstate squeeze-front-end)))

(if (and (squeeze-front-end)
  (cacheon?))
  (assert
    (message "Because the cache is on, we cannot move both sides to within 512K, since that is the cache size and only the cache would be exercised by the tests."
    EndAddress = original-end-address
    (setstate determine-forward-increment)))

(if (and (squeeze-front-end)
  (not (cacheon?)))
  (assert
    (message "Because the cache is on, we can move the forward end in while the end address is scrunched also."
    EndAddress = squeezed-end-address
    (setstate determine-forward-increment)))

(if (decided-forward-increment)
  (perform-forward-scrunch)) // goes through forward scrunching algorithm, returns value in squeezed-forward-address

(if (and (decided-forward-increment)
  (cacheon?))
  (assert
    (run-wide-test) // run test between squeezed-forward-address and original-end-address
    (setstate read-logic-analyzer))

(if (and (decided-forward-increment)
  (not cacheon?))
  (assert
    (run-narrow-test) // run test between squeezed-forward-address and squeezed-end-address
    (setstate read-logic-analyzer)))

// Each of the forward scrunch rules
(if (determine-forward-increment)
  (assert
    (message "It is critical that one remembers that changing the starting address by the wrong offset is going to change the pattern being written to memory and could have unpredictable consequences on the module.")))
(if (and (determine-forward-increment)
  (= current-test 1))
  (assert
   (message "Test 1 is invariant to pattern shifts of 256 addresses")
   (set_forw_incr 256)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
  (= current-test 2)
  (= toetag.bitwidth 16))
  (assert
   (message "Test 2 is invariant to pattern shifts of 65536 addresses for x16 modules")
   (set_forw_incr 65536)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
  (= current-test 2)
  (= toetag.bitwidth 8))
  (assert
   (message "Test 2 is invariant to pattern shifts of 65536 addresses for x8 modules")
   (set_forw_incr 256)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
  (= current-test 2)
  (= toetag.bitwidth 4))
  (assert
   (message "Test 2 is invariant to pattern shifts of 16 addresses for x4 modules")
   (set_forw_incr 16)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
  (= current-test 3)
  (= toetag.bitwidth 16))
  (assert
   (message "Test 3 is invariant to pattern shifts of 65536 addresses for x16 modules")
   (set_forw_incr 65536)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
  (= current-test 3)
  (= toetag.bitwidth 8))
  (assert
   (message "Test 3 is invariant to pattern shifts of 256 addresses for x8 modules")
   (set_forw_incr 256)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
  (= current-test 3)
  (= toetag.bitwidth 4))

57
(assert
  (message "Test 3 is invariant to pattern shifts of 16 addresses for x4 modules")
  (set-forw-incr 16)
  (set-state decided-forward-increment))

(if (and (determine-forward-increment)
  (= current-test 4)
  (= toetag.bitwidth 16))
  (assert
    (message "Test 4 is invariant to pattern shifts of 65536 addresses for x16 modules")
    (set-forw-incr 65536)
    (set-state decided-forward-increment))

(if (and (determine-forward-increment)
  (= current-test 4)
  (= toetag.bitwidth 8))
  (assert
    (message "Test 4 is invariant to pattern shifts of 256 addresses for x8 modules")
    (set-forw-incr 256)
    (set-state decided-forward-increment))

(if (and (determine-forward-increment)
  (= current-test 4)
  (= toetag.bitwidth 4))
  (assert
    (message "Test 4 is invariant to pattern shifts of 16 addresses for x4 modules")
    (set-forw-incr 16)
    (set-state decided-forward-increment))

(if (and (determine-forward-increment)
  (= current-test 5))
  (assert
    (message "Test 5 must be handled by performing a binary search, it is a copy-type test")
    (set-forw-incr 1)
    (set-state decided-forward-increment))

(if (and (determine-forward-increment)
  (= current-test 6))
  (assert
    (message "Test 6 must be handled by performing a binary search, it is a copy-type test")
    (set-forw-incr 1)
    (set-state decided-forward-increment))

(if (and (determine-forward-increment)
  (= current-test 7))
  (assert
    (message "Test 7 is invariant to pattern shifts in multiples of 8 addresses")
    (set-forw-incr 8)

58
(setstate decided-forward-increment)))

(if (and (determine-forward-increment)
         (= current-test 8))
  (assert
   (message "Test 8 must be handled by performing a binary search, it is a copy-type test")
   (set_forw_incr 1)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
         (= current-test 9))
  (assert
   (message "Test 9 is invariant to pattern shifts in multiples of 16 addresses")
   (set_forw_incr 16)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
         (= current-test 10))
  (assert
   (message "Test 10 is invariant to pattern shifts in multiples of 4 addresses")
   (set_forw_incr 4)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
         (= current-test 11))
  (assert
   (message "Test 11 is invariant to pattern shifts in multiples of 16 addresses")
   (set_forw_incr 16)
   (setstate decided-forward-increment)))

(if (and (determine-forward-increment)
         (= current-test 20))
  (assert
   (message "Test 20, the Memtest test that imitates the behavior of MEMTEST, is invariant to pattern shifts in multiples of 56 addresses")
   (set_forw_incr 56)
   (setstate decided-forward-increment)))

// Each of the methods that search the logic analyzer trace

(if (read-analyzer-trace)
  (assert
   (message "If you would like for the system to analyze the logic analyzer trace of a failure please capture the appropriate information in a text format then, copy the test log file and ASCII file into \expert\analyzer. Then you must type in the names of the files in the specified fields and click the "Yes" button. If you are either unable to capture a trace or do not wish to do so, click "No" and the window will disappear")))

// ask user for a logic analyzer trace (window has two textfields, OK
// button, and No button)

(if (and (read-analyzer-trace) (clicks-yes?))
  (assert
   (parse-logic-analyzer))) // this reads the file in to
   // appropriate signals
   // and columns on which the
   // methods/algorithms operate

(if (and (read-analyzer-trace) (not (clicks-yes?)))
  (assert
   (setstate finish-analyzer-trace)))

// verify that the fail seen in the trace matches the one in the logic-
// analyzer these are items when looking at a single failure's trace
// two cycles between a pall and a bank activate, or a TRP = 2 error

(if (and (read-analyzer-trace) (trp-of-2?))
  (assert
   (message "A TRP=2 has been detected, which can
   aggravate signal margin. You should
   investigate")))

(if (and (read-analyzer-trace) (not (trp-of-2?)))
  (assert
   (message "TRP=2 was not observed in the logic analyzer
   trace.")))

(if (and (read-analyzer-trace) (bursting?))
  (assert
   (message "Bursting cycles have been noted")
   (calculate-bursting-percentage))) // reports value in
   // message box a cycle (from BA to BA) is
   // bursting if at any time DQM = 0 twice in a
   // row

(if (read-analyzer-trace)
  (assert
   (message "An attempt to recreate the adjacent cell contents
   has resulted in the creation of the file cells.txt
   in \expert\. Also, the file marker.txt,contains
   the time marker where any such cells were observed
   in the trace")
   (print-cell-contents))) // this creates an array of adjacent cells to
   // the failing one looking for physical adjacency
   // issues, create an 7x7 grid with(adjacent-
   // marker)Record position of adjacent cells using
   // row/col/bank/DQ and time marker

(if (and (read-analyzer-trace)
(five-in-a-row?) // note any time a particular address was accessed 5 times in a row
(assert (message "At one point, the address " access_addr " was accessed at least five times in a row. This might be an indication of something suspicious.")

(if (and (read-analyzer-trace) (write-to-failing-cell?))
  (assert (message "A write to the failing-cell has been observed at following marker " write-marker)
  (report-cbr-status))) // if cbrs are within 3 chip cycles of write-marker, (message "suspicious cbr's observed near failing write")

(if (and (read-analyzer-trace) (read-from-failing-cell?)) // returns read-marker
  (assert (message "A read from the failing-cell has been observed at following marker " read-marker ").
  (report-cbr-status))) // if cbrs are within 3 chip cycles of read-marker, (message "Suspicious cbr's observed near read from failing cell")

(if (and (read-analyzer-trace) (not (clicks-yes?)))
  (assert (setstate finish-analyzer-trace)))

//finish read-analyzer-trace by asking user to look things over and see if we want to investigate another fail, give window, yes or no box
(if (finish-analyzer-trace)
  (assert (message "Do you want to investigate another failing address?")))

(if (and (finish-analyzer-trace) (clicks-yes?))
  (assert (message "You have chosen to investigate another failing address")
  (setstate pick-fail)))

(if (and (finish-analyzer-trace) (not (clicks-yes?)))
  (assert (setstate exit)))
(if (exit)
  (assert
   (message "The system will now quit.")
   (System.exit(0)))
)
References


5. van de Goor, Testing, Chapter 5-6.


References (Cont.)


Acknowledgments

First and foremost, I would like to thank Phil Stevens and Bob Kruggel, the experts who provided the motivation for this concept. They both devoted a sizable portion of their daily schedules to help the project come to fruition. They played major roles in the system’s design, implementation, and verification. They assisted not only with the intellectual aspect of the task but with various logistics, as well. Most importantly, they displayed outstanding patience when quizzed repeatedly and extensively about the details of their responsibilities.

Prof. Srinivas Devadas provided valuable guidance in ensuring the project’s standing as a thesis. This included advice on a wide range of matters, including the initial project concept, its scope, the system’s design, and testing plan. I also appreciate his input and help with this document’s preparation.

Gary Grise, as the manager who supervised this project, provided critical logistical support to this project. I would like to thank him for freeing both the human and traditional resources needed to complete the thesis.

Bob Fiorenza, the IBM 6A coordinator, facilitated the project at the outset and was a key source of support throughout its course.

I would also like to thank Archit Shah for his advice on implementation issues as they related to Java.

I should note that Ernest Friedman-Hill is the creator of the freely available tool JESS, which was used during the course of this thesis.