EXPERDITE

A MODEL BASED EXPERT DIAGNOSTIC SYSTEM USING A HIGH LEVEL TEST DESCRIPTION LANGUAGE

CS501 Master's Project

College of Computer Science, Oregon State University.

Major Professor : Dr. Bill Bregar

Submitted by

Philip (Andy) Tilp

May 24, 1991

ABSTRACT

To be competitive in the development and production of complex electronic systems, a company must constantly scrutinize its manufacturing processes to find ways to reduce costs and increase productivity. One of the more costly problems is the inability to troubleshoot and repair faulty circuit boards on the spot. Rather, they must be diverted off the manufacturing line to a senior technician for diagnosis and repair. The purpose of Experdite is to provide a practical expert diagnostic system for manufacturing line personnel so they can quickly troubleshoot and repair most circuit board faults, thereby eliminating the need to send the board off line.

Experdite also addresses other inherent manufacturing line issues, including the need to have a general purpose and easily adaptable system to work with many different circuit boards. It addresses the need for an expert diagnostician before anyone has had the time to become an expert troubleshooter on the circuit board. The final issue addressed is the ability to quickly and easily modify the knowledge base to handle any inevitable hardware changes.

The component and self-test behavior for the target circuit board is abstracted and then modeled with a high level Test Description Language, or TDL. A TDL to C++ translator converts the TDL description into a knowledge base of C++ classes. Then the inference engine traces the thread of causality to a fault on a circuit board by gleaning information from the knowledge base and comparing it to the actual behavior of the circuitry.

This paper elaborates on each of these points. Also included in the paper are descriptions of the implementation and application of Experdite to a production circuit board.

1. INTRODUCTION

1.1 General background

In order for a company to thrive in the extremely competitive world of "high-tech" product development and production, a constant vigilance to reduce costs and increase productivity must be maintained. Diagnosing and repairing faulty circuit boards is one of the most costly manufacturing processes[6]. The current practice is to divert a faulty circuit board from the manufacturing line to a senior technician for diagnosis and repair. The intent of Experdite is reduce the costs of diagnosing and repairing circuit boards by providing expert troubleshooting advice so "on th e spot" repair is possible.

Beyond the primary goal of providing "on the spot" diagnostic help, Experdite addresses three problems inherent in a manufacturing process.

1. When a circuit board is first introduced into production, no one person has any significant experience trouble shooting the board. However introduction time is when an expert is most needed. This problem is referred to as the *immediate expert* problem.

2. A manufacturing line typically produces many different circuit boards. Therefore, the expert system must be usable on any circuit board.

3. Large, complex circuit boards inevitably require modifications to their circuitry after their production starts. Thus, the existing knowledge base for the circuit board must be easy to change and adapt to reflect the modifications to the circuit board.

Experdite consists of two main components, a *Test Description Language*, or *TDL* and a general purpose inference shell. The TDL is used to model the structure and behavior of the components and tests on most any type of digital circuit board. A knowledge base, specific to the modeled circuit board, is created by translating the TDL description into C++ classes.

Experdite uses a model-based inference to trace down faults on the board. This inference method compares the actual behavior of a circuit board to the modeled behavior and acts on anomalies between them.

Following is an example of the model-based methodology used by Experdite to find a fault. Figure 1 shows a system with three circuits, A, B and C. The dotted or dashed lines in the figure show the paths of tests T1, T2 and T3 through the circuitry. A test is a series of vectors passed through a sequence of circuits. The vectors should cause some predetermined behavior. A pass or fail report from a test indicates whether or not the actual behavior matches the expected behavior.

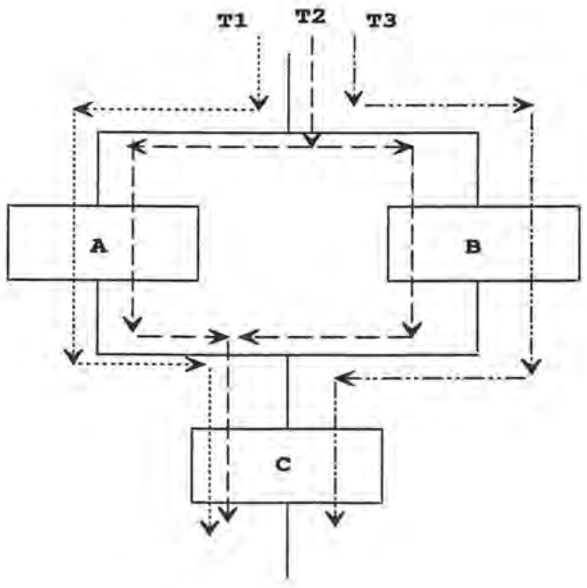

Figure 1. Three circuit example.

A test, in most "real world" applications, does not check the full functionality of every circuit it passes through. A test may range in coverage from a check of absolutely every function in the circuit to a check of next to nothing. To take the varying coverage into account, the description of every circuit/test pair includes an *extent-of-coverage* value. For this example, Table 1 lists the extent-of-coverage value for each circuit/test pair.

Circuit	Test	Extent-of-coverage
A	T1 T2	Considerable Moderate
B	T2 T3	Absolute Moderate
C	T1 T2 T3	Considerable Moderate Moderate

Table 1. Example test coverage.

Assume one circuit is faulty, though Experdite initially suspects all circuits equally. Experdite must pare down the list of suspects in the least amount of time and the fewest number of test executions. The operations of Experdite for this example are described in the following paragraphs and summarized in Table 2.

The first task of Experdite is to choose the best test to start the process. (A detailed discussion of the selection process is in section 4.) Suppose test T1 is selected and it fails. This indicates a fault is somewhere in path of the test. The failure provides several pieces of information. First, since circuit B is not in the test path, it cannot possibly be the cause of the fault. Circuit B is removed from the suspect list. Second, observe that test T1 has the same extent-of-coverage value for both circuits A and C.

Thus, there is no evidence to indicate one circuit is more likely the cause of the fault than the other.

Experdite continues its search by selecting another test. Notice test T3 checks circuits B and C, but not A. Since circuit B is known to be good, if test T3 fails, then circuit C is definitely the cause of the fault. If, on the other hand, the test passes, then the evidence leans toward naming circuit A as the most likely cause. However, since test T3 only moderately evaluates circuit C, the possibility exists that the test did not detect the fault in the circuit. Therefore, circuit C cannot be eliminated as a suspect.

Suppose test T3 is selected and it passes. Circuit A is now the primary suspect, but for Experdite to reach a definitive conclusion, it needs more evidence. Suppose Test T2, the only remaining test, fails. Since test T2 checks both circuits A and C only to a moderate extent, little additional information is produced. Experdite still cannot draw a definitive conclusion. However, since all the tests have been executed, a best guess must be made based on the available evidence. The data point to circuit A as being the cause of the fault.

Target Circuit	Test Run	Test Result	Implication
C	T1	Fail	B determine good. A and C equally likely suspect.
C	T3	Pass	A most likely suspect. C shown to be partially fault free.
A	T2	Fail	Confirms fault in A or C. Leaning towards A.

Table 2. Summary of example Experdite operations.

Experdite uses a secondary reasoning process of if-then type rules, called *look-aside rules*. Look-aside rules are used to handle special cases where the model-based method may break down, as explained in the following example.

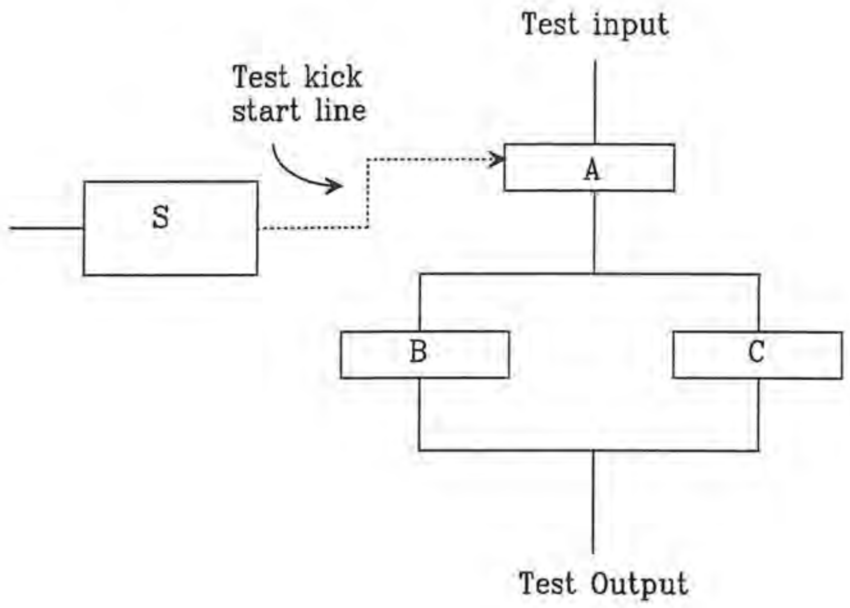

Figure 2. Example circuit for special case rules.

For example, consider the circuit shown in Figure 2 containing the four circuits, A, B, C and S. Suppose a test T1 passes its vectors through circuits A, B and C only, but requires circuit S to "kick start" the test execution. Except for the initialization, circuit S is in no way affiliated with the test. Therefore, circuit S is not included in any circuit/test relationship description. However, should circuit S fail, then the normal behavioral model would still point to circuits A, B and C as suspects and miss circuit S altogether. The model-based method breaks down and an erroneous result is reported. So, for cases such as these, a look-aside rule provides special attention to potentially faulty "invisible" circuits.

1.2 Previous work

There have been several previous examples of working, viable diagnostic expert systems, from early work on MYCIN[1] to the system currently aboard the Hubble Space Telescope[2]. Pau's survey[8] of expert diagnostic systems lists many different approaches to the inference methodology and types of knowledge bases used in various systems. His survey indicates the most common inference scheme used in diagnostic systems is the rule-based methodology. However, the model-based inference scheme is also prominent.

Novak[12] contends that a back-chaining rule-based inference technique is sufficient to troubleshoot electrical components in an AC/DC motor control circuit. He does, however, use a model-based approach to develop a knowledge base. This means conclusions for rules are developed based on circuit behavior. Novak opted for the model-based type knowledge because "it allows for a systematic and comprehensive method of defining the conclusions".

Rule-based systems, including Novak's, are very domain specific, allowing for little carryover into other domains[1]. A set of rules is written exclusively for a specific system and are usually not transferrable to any other system. Thus, using a rule-based inference methodology exclusively in Experdite would not fulfill its requirement of being general purpose.

Fink[7] used both rule-based and model-based inferences in her diagnostic system. She asserts that the shallow knowledge of a rule based system is sufficient in most common cases. But for problems not classified by a rule, the deeper knowledge of the system behavior is needed. Fink uses two separate knowledge bases, one for rules and one for modeled behavior. The inference engine uses both knowledge bases in conjunction, deciding on case by case basis which type of knowledge and inference method to use.

Fink's idea of using both model-based and rule-based inference methodologies is used in Experdite. Fink, however, relies more heavily on experiential, or rule-based, knowledge. Experdite is required to be working before any experiential knowledge exists. Therefore, Experdite relies primarily on a model-based system, using rules only as backup for uncommon cases.

Davis[3] and DeKleer[4], in contrast to the other systems, both built their systems entirely around a model-based knowledge base and diagnostic inference methodology. Both systems use a similar technique that Davis calls *constraint suspension* to track down faults. Constraint suspension looks for the constraint, or circuit output, which if retracted, leaves the system in a state that is consistent with the model. This technique is applied when the actual behavior of a system is inconsistent from the modeled expected behavior in the knowledge base.

Rather than direct the search to a consistent state, as done by Davis[3] and DeKleer[4], Experdite follows Marques'[9] lead in focusing the search always on the most likely candidate. This allows the constrained circuits to be quickly eliminated from further consideration.

Davis[3] and DeKleer[4] require that a component model description in the knowledge base specify the actual behavior of the component. Describing functionality of a a circuit to this level permits diagnosis to a very fine grain. But creating the descriptions is an extremely complex and time consuming task. Experdite simplifies the modeling process by abstracting the model to a higher level and by providing a standard, easy to use description language.

Both Yen[5], and Fink[7] use frame objects to represent knowledge. As Yen states, the advantage of frames is that they can hold mutually exclusive information for the entity they are describing. Experdite extends the frame concept by making a frame a C++ object. This allows each circuit and test to store information about itself, and also provides an exclusive way for each object to update its own information. This relieves the inference engine of the burden of knowing how to manipulate data. Rather, the inference engine needs to know only how to tell the object to handle its own data.

The problem of drawing a conclusion from partial data that is accumulated over time to has been the subject of much study. Yen[5] handles this problem with an acquired belief measure derived as an extension to the Dempster-Shafer theory. Experdite modifies Yen's approach by simplifying the algorithm that determines the belief factor. Since each component is represented by its own object, each circuit maintains its own belief measure. This allows each object to handle any special considerations for itself, something not possible in a central belief calculation.

Horvitz[10] asserts that when reasoning with partial information, the decision of when to refine the search strategy and when to assert a conclusion should be made as the incomplete information is gathered. Experdite uses this idea by switching its search mode based on the previously collected partial data. Also, Experdite determines when a continued search is futile and makes a best guess based on the information gathered to that point.

1.3 Paper organization.

Section 2 presents a detailed overview of problems this project is tackling. Included in the section is background on how a manufacturing process works and why a faulty circuit board in the process is so costly. The section also covers the current tools and methodology used to troubleshoot faulty circuit boards. The final topic covered in the section is the solution this project proposes.

Section 3 describes the abstraction and modeling of the hardware circuitry. Included in the section is an overview of the high level description language and the methodology of how it is used. Rule generation is also covered in this section.

In Section 4, the inference engine is discussed. The section starts with an overview, followed by details on the so-called belief factors, the modes of the inference engine operation and look-aside rules. The criteria of when to declare a solution and a discussion of how the "what is faulty" decision finishes the section.

Section 5 gives the results of experiments when Experdite was applied to a real, production circuit board. An overview of the hardware circuitry and an explanation of experiments are given. The section finishes with a listing of the actual experimental results.

Section 6 contains suggestions for future improvements to the system. Conclusions and analysis of the system are presented in section 7.

Appendix A discusses the inference engine program design and implementation considerations. Appendix B covers the implementation of the high level test description language. Appendix C contain the Test Description Language model of the hardware used in the experiments.

2. THE APPLICATION DOMAIN

2.1 The high cost of faults

The first real application for this project is a high performance graphics workstation manufacturing line. The manufacturing process, shown in Figure 3, starts with the assembly of raw material into several different kinds of circuit boards. After assembly, a junior technician verifies the functionality of each board with a suite of tests. Fully functional boards continue down the line to final assembly, where they are combined with other circuit boards to complete a product. If, however, a fault is detected on the board, then it is diverted from the main stream to the repair station. At this point a senior technician diagnoses and fixes the faulty circuit board. The repaired board then reenters the normal manufacturing stream.

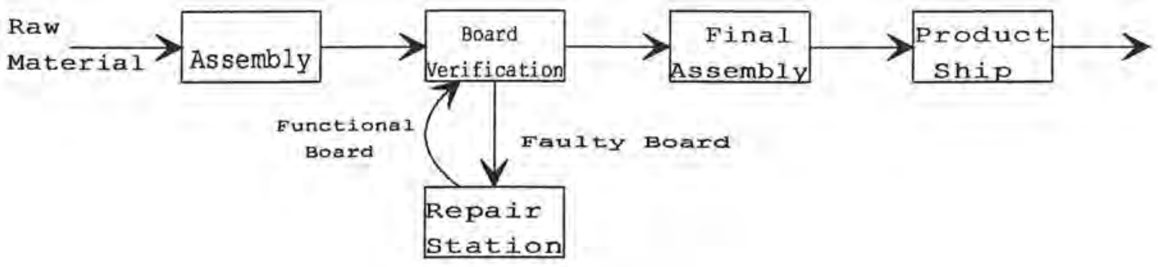

Figure 3. Manufacturing line sequence of operations.

According to statistics gathered from the manufacturing line, workmanship or process errors, such as solder bridges or incorrect/misplaced parts, account for 95% of all the faults on the circuit boards[6]. These types of faults are typically visible and obvious once spotted. However, the circuit boards commonly measure 14X16 inches and contain several hundred integrated circuits. This translates to several thousand solder points. Obviously, given the magnitude of the visual task, trying to spot a workmanship or process error simply by looking at every IC and every solder point on the board is wasteful and beyond human capability.

Junior technicians do have the ability to fix workmanship and process errors, but do not have the time nor the expertise to find the fault. Thus the board is always diverted to a senior technician. Experdite provides the junior technician with the expertise needed so s/he can find and quickly correct faults on the spot. The result being no disruption to the manufacturing line or costly diversion. Also, the senior technician is freed to diagnosis the more difficult problems, thereby increasing his/her productivity as well.

2.2 Built-in self-test and diagnostics

The tests used by the junior technician to check a board are so called *built-in self-tests*. The term *built-in* means the self-test firmware is embedded into the hardware. A *self-test* is firmware program that injects a set of known vectors into a section of the circuitry. The test vectors pass through several components from the injection point to the exit point. The behavior of each component alters the vectors in some known manner. When the vectors reach their exit point, they are compared against a set of known good vectors. If all the actual results match the expected results, then the tested circuits are deemed healthy, with the caveat that the test may not check all the hardware 100%. If the vectors do not match, then some circuit in the test path is faulty.

The self-test suite for a circuit board is typically developed in conjunction with the circuitry design. Presumably, the test designer has in-depth knowledge of how the circuitry should behave. Using this knowledge, each self-test is developed by first determining what needs to be checked in a circuit and what vectors are required to do the circuit operations. Then, the path through any other circuitry that the test vectors must take to get to the target component is determined. The effects of the other circuits on the test vectors must be factored into the test design. Finally, the software is written to achieve such an operation.

Since a self-test merely discriminates between the expected behavior and actual behavior of the components in its path, the functionality of the tested circuits can be quickly ascertained. However, this also a weak point in self-tests. A test result gives only an indication whether or not a fault occurred along the test path. The key item here is that a self-test is an excellent <u>verification</u> tool, <u>not</u> a satisfactory <u>diagnostic</u> tool[3],[4]. Experdite, however, uses the knowledge of the interactions between the self-tests and the components to make inferences regarding the cause of the fault. This notion is discussed in detail in section 4.

2.3 The production line's need for an immediate expert.

When a circuit board is first introduced into production, a twofold problem can occur. First, a higher than normal percentage of faulty circuit boards are typically seen at the circuit board verification station[6]. Second, the senior technician's experience level diagnosing and repairing the new board is minimal at best. Combined, these two problems have the potential of choking the production line.

By creating an Experdite system for a circuit board during its design phase, an expert diagnostician can be available when the board enters production. Thus, possible choking of the production line is prevented and productivity of both the junior and senior technician is increased.

3. ABSTRACTING AND MODELING A SYSTEM.

3.1 Overview

Abstracting the interaction between a test and a circuit can be thought of as modeling the deep knowledge of the domain with functional primitives[7]. Deep knowledge characterizes its domain in terms of functional information. It is used to model how the system works. This is contrary to so-called shallow knowledge, or "rules of thumb", which models how experts reason about the domain. Represented within the deep knowledge primitives are the structural and behavioral knowledge of the domain, where the expected behavior of each circuit is based on the stimulation provided by the test.

Experdite uses a high level description language, called *Test Description Language*, or *TDL*, to describe the high level abstraction of a hardware system. Primitives representing the hardware system's components, tests and their interactions are combined to generate a description language[8]. The language provides a simple mechanism for the self-test designer to model the behavior of the system and create a knowledge base specific to the circuit board.

Previous description languages[3],[8] required detailed modeling of internal workings for each component. The TDL abstracts a system description to use only the existence of, and the relationship between, the components and the tests. Therefore, it is possible to model essentially any system that verifies its components by a series of tests, where the tests report whether or not the expected behavior occurred.

3.2 How to abstract and model a system

The creator of a TDL model for a circuit board should be an expert on how the board's components and tests interact. Most likely, the TDL author and the self-test designer are the same. To write the TDL description of a circuit board, the writer only do the following two steps. (The syntax for the TDL is described in Appendix A.)

1. List all the components and tests.

2. For each component, describe the relationship it has with each test that passes vectors through it. The description must include the following information:
   a. the extent of test coverage
   b. the cost/benefit ratio of the test.

3.2.1 Extent of test coverage

How thoroughly and completely a test checks a component is quantified in the *extent-of-coverage* value. Extent-of-coverage takes four values: *Absolute, Considerable, Moderate,* and *Minimal.*

The *Absolute* category indicates the test checks 100% of the circuit. If an *Absolute* rated test passes, then the associated component is fully functional. Conversely, if the component is faulty in any way, this test will catch the incorrect behavior.

The *Considerable* category indicates the test covers 99% to 66% of the circuit's functionality, the *Moderate* category indicates a 65% to 33% check and *Minimal* means 32% to 1% of the circuit is tested. In each of these cases, there is a chance the test did not stimulate the faulty part of the component, even if the test passes. Thus, there is insufficient evidence to declare the component fully functional. Each category does, however, provide some degree of assurance that the component is functional. This partial assurance is used by the inference engine in the determination of the most likely cause of the fault. The inference engine is explained further in section 4.

For each test/component relationship, the TDL writer must determine which extent-of-coverage category to apply. The writer must make the judgement based on his/her knowledge of how the circuit operates and how definitively the test checks the circuit.

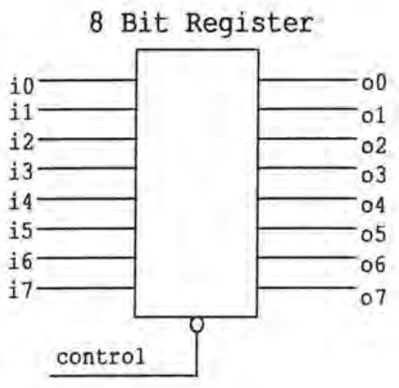

Figure 4. Simple 8 bit latch.

For example, consider the simple 8-bit latch shown in Figure 4. A latch is a device that captures the data on the input lines when the control line is asserted and holds the data on the output lines. To classify a test on the latch as *absolute,* it must check all input, output and control lines in all possible combinations. A *considerable* extent-of-coverage type test may test all combinations of only six of the data lines. Checking three data lines puts a test in the *moderate* category. Finally, a test that merely flips the state on one line, one time is a *minimal* type test.

For another example, consider a counter. A counter latches in a starting value, increments the value with every clock input and places the value on the output lines. An *absolute* test would require starting the input with every possible value *n*, then increment the value its full range (*n, n+1.... n-1*), while checking the output value after each increment. A *considerable* check might increment the counter's full range, starting with only a single value, and checking only the final value. A *moderate* check may count a value far enough to cause the state to change on half of the output lines. A *minimal* check may increment an incoming value by one, causing only a single output bit to change.

3.2.2 The cost/benefit ratio.

The other parameter in the component-test relationship description is the *cost/benefit ratio*. This ratio indicates the cost versus the benefit of executing a test on the specific component. The inference engine uses this ratio when determining the next test to execute. For example, given two tests which are equal except in their cost/benefit ratio, the test with lowest ratio is selected.

The *cost* of a test can be defined in terms of four quantities: *time, money, invasiveness* and *destructiveness.*

The *time* value is directly related to the amount of time required to setup and/or execute a test. The more time a test requires, relative to other tests, the higher the cost.

The *money* value relates the monetary expense of executing a test or of any test fixtures or equipment. For instance, if a test requires a very expensive lab setup, then it has a high cost value.

Invasiveness is defined as the amount of dismantling or probing within a system that is required by a test. Any such action requires extra time and expertise, thus a higher cost.

Destructiveness means the amount of irreparable damage a test may cause. The more destruction, the higher the cost.

The benefit portion of the cost/benefit ration is quantified in terms of *definitiveness, isolation* and *thoroughness.*

The *definitiveness* factor relates the likelihood a test will discover a fault in a circuit, and not be bothered by the other circuits in the test path. That is, if a test has "absolute" coverage of a circuit, and "minimal" coverage for all other circuits in its path, then the benefit of the test for this circuit is very high.

The *isolation* factor is inversely proportional to the number of circuits in a test path. If a test fails, then all circuits in its path are suspect and all the suspended circuits, or those not in the test path, are eliminated as suspects. The smaller the number of circuits, the lower the suspect count and the easier to pin down the fault. Thus the benefit increases as the path count decreases.

Thoroughness is defined as how rigorously a test checks a component. The more exhaustive the test, the higher the value. This is related to the extent-of-coverage value defined above.

Ratios are formed by applying points to each cost and benefit category, as applicable to the test and the circuit. The actual point value assigned is currently left to the discretion of the TDL writer. The only restriction is the value must be in range from 1 to 10. For a cost category, the lowest cost value is 1, while the highest is 10. For the benefit category, the least benefit value is 1 and the most benefit value is 10.

Following is an example of how the values of the cost/benefit ratio is determined for two tests that check an EEROM (Electrical Erasable Read Only Memory).[1] The first test, T1, reads the every data location in the EEROM, performs a

[1] An EEROM is an integrated circuit capable of acting as a normal read/write memory. However, an EEROM is unique because it is able to retain data without power applied. The caveat is that the EEROM is limited to a finite number of write operations to any one cell. If the limit is exceeded, then the cell decays and rapidly fails.

check-sum on the values and compares the calculated value to the check-sum value stored in the EEROM. If values match, the EEROM data is assumed to be correct. T1 does no write operations on the EEROM. The second test, T2, writes, then reads several patterns to every EEROM memory location. Thus, T1 checks the integrity of the data and checks nothing of the write operations. T2 does a full read and write check of the part, yet with the potential of destroying the part. The determination of the cost/benefit ratio for each test follows.

Test	Ratio Type	Category	Comment	Value
T1	Cost	Time	Negligible, < 1 millisecond	1
		Money	None	1
		Invasiveness	None	1
		Destructiveness	None	1
T1	Benefit	Definitiveness	A faulty address or data bus can interfere	3
		Isolation	Only the processor, bus and EEROM	4
		Thoroughness	Moderate, only test data retention	3
T2	Cost	Time	Negligible, < 1 second	1
		Money	Potentially high, if destroy EEROM	9
		Invasiveness	None	1
		Destructiveness	Potentially great, may destroy part	9
T2	Benefit	Definitiveness	A faulty address or data bus can interfere	3
		Isolation	Only the processor, bus and EEROM	4
		Thoroughness	Fully test all functionality of EEROM	9

Thus, the cost and benefit ratio for test T1 is 4/10, or 0.4, and for test T2 is 20/16, or 1.25. Clearly, the benefit of Test T2 is much greater than test T1, but so is the cost of running the test. Therefore, test T1 would be selected over T2 because it provides a reasonable check the EEROM. T2 would only be selected in special cases when greater testing is required to help isolate a fault.

Refer to Section 6 for a further discussion on the need to provide an exact method of assigning cost and benefit values.

3.3 Look-Aside Rule definition and generation.

As mentioned in the introduction, a look-aside rule is used in the special cases when the model based system is unable to characterize unusual circuitry. A rule's conditions and conclusions are acquired from senior technicians who have significant diagnostic experience on the circuit board. A look-aside rule is always attached to a test and is activated only when the test fails.

The rule format is

"if *(premise)*$^+$, then circuit *n* is faulty".

A single rule can have multiple *premises*. Two types of premises are currently supported, *test results* and *measurements*.

A *test result* premise is derived after a diagnostician repeatedly and consistently observes that a certain test result always indicates a specific fault. For example, suppose a test is capable of reporting back "bits in error". If a fault always causes a unique bit pattern to be reported, then a premise can be derived from the information. The format for a result-based premise is

"if test *T* reports a result of *X*, then....".

For example, suppose in test T1 that an output bit of a circuit A is used exclusively to trigger another chain of events. If the bit does not toggle, the event chain fails to start and the test times-out. In this condition, the self-test control software detects the time-out and reports an "event failed to start" message. Also assume no other condition can cause such an report. A *test result* premise based rule for such a condition may be :

```
"If T1 reports event failed to start, then circuit
A is faulty."
```

Refer to Appendix B for the correct rule syntax.

A *measurement* premise stems from measurements taken by a diagnostician while troubleshooting faulty systems. If a distinguishing, measurable characteristic consistently identifies a fault, then this knowledge can be used to make a premise. The format for a measurement based premise is

```
"if measurement M1 on circuit A is X, then ....".
```

For example, suppose Circuit A is a clock signal output device. A 16 MHz signal is expected from its pin 10. A rule with a measurement premise may be :

```
"if 16 MHz not at pin 10 of Circuit A, then
Circuit A is at fault."
```

Look-aside rules can be included during the generation of a circuit board's original TDL description, or they can be added later as they are discovered. Refer to Appendix B for a detailed description of the rule syntax and the translation process of TDL rule format to C++ classes.

3.4 TDL translation and the inference engine interface.

The first step in building an Experdite system specific to a circuit board is to generate the TDL description of the circuit board. The description is then translated to C++ objects that make up the knowledge base for the inference engine. Each circuit, test, circuit/test relationship and rule is represented by its own C++ object. (The translation process is discussed in detail in Appendix B.) The translated knowledge base is compiled with the inference engine to produce an expert system unique to the circuit board.

4. INFERENCE ENGINE

4.1 Overview

The purpose of the inference engine is to guide the search for the faulty circuit. It must narrow the search space as quickly as possible[7]. The inference technique Experdite uses is similar to Davis' *constraint suspension* technique[3]. A *constraint* represents the expected relation between the nodes it connects[11]. For example, the constraint of adder B in Figure 5 is the output point z, which is the sum of the inputs at points x and y.

In Experdite, *constraint suspension* means divide and conquer. That is a constraint (the output of a component) is deliberately inhibited (suspended). Thus, if a test fails, then any suspended circuits are automatically known not to be the cause of the fault. Therefore, assuming a single fault, the suspended circuits are pruned from the suspect list.

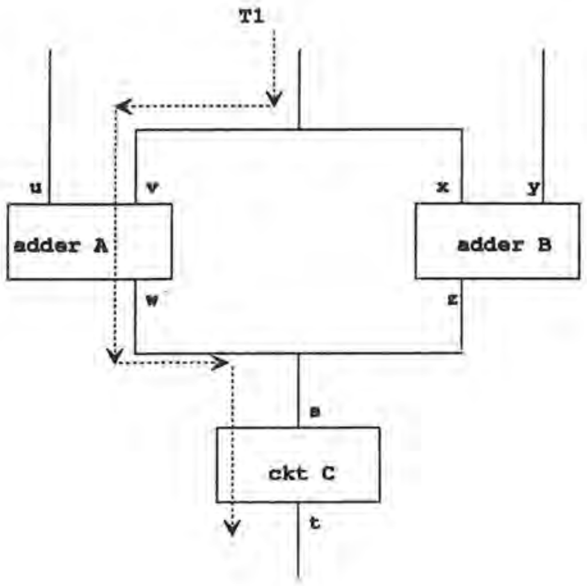

Figure 5. Example circuit to show constraint suspension.

For example, test T1, shown by the dotted line in Figure 5, checks only adder A and circuit C, but not adder B. Therefore, the constraint of adder B is suspended. If test T1 fails, then adder B is known to be good.

On the contrary, a passing test does not automatically indicate the fault is outside the test path, unless, of course the test covered all circuits in its path absolutely. Further investigation is warranted to determine if the fault cause is outside of the test path, or if the test simply did not stimulate the right condition to trigger the fault. Referring, again, to the above example, if T1 did not toggle all the output lines of adder A in all combinations, a stuck bit may have been missed.

When the inference engine selects a test to run, its decision is influenced by two primary goals. First, always keep the focus of the search on the fault. That is, choosing a test that is expected to fail keeps the fault within a known boundary, namely a circuit in the test path. Second, maximize the benefits of constraint suspension. That is, choose a test so if it fails, then the maximum pruning of the suspect list can occur.

The factors used by the inference engine during the test selection process are the cost/benefit ratio and the extent-of-coverage values. Also, depending on the inference engine mode of operation, the most likely suspect and the number of circuits in a test path are also used to varying degrees in the selection. The decision process for each mode of the inference engine is explained in Section 4.3.

After every test execution, the suspect list is updated and examined to determine if sufficient evidence exists to establish the cause of the fault. The criteria for this determination are explained in Section 4.5.

4.2 Belief in a circuit's functional status

Each circuit carries with it a notion called a *belief-factor* that is the belief in its own functional status. This *belief-factor* ranges in value from -10 to 10, where -10 indicates the circuit is definitely faulty and 10 means the circuit is fully functional. The gray area between the maximum values allows a circuit to maintain an inconclusive opinion as to its own status, ranging from *probably good* to *may be good* to *may be bad* to *probably bad.* The inference engine uses the inconclusive belief when ordering the circuits in the suspect list. The term *most-likely-suspect* is the circuit with the belief-factor closest to -10.

The belief-factor is a cumulative value. It is calculated on a per test execution basis, using the following two factors.

1. The value indicating the *extent-of-coverage* provided by the test.
2. The value indicating whether the test passed or failed.

The defined values for *extent-of-coverage* are as follows.

```
Absolute = 10,
Considerable = 7.5
Moderate = 5.0,
Minimal = 2.5
```

The defined values for pass or fail, *(P,F)*, are 1 and -1, respectively.

A value, *coverage-value*, is the combination of the extent-of-coverage and pass/fail values, where

coverage value = (extent-of-coverage * *(P,F)*).

For each circuit in the path of an executed test, the *belief factor* is calculated as the mean and standard deviation of the *coverage-value*. The mean and standard deviation equations are as follows.

$$\text{mean} = \left(\sum \text{coverage values} \right) / \text{number of tests}$$

$$\text{std dev} = \sqrt{\left(\sum (\text{coverage value} - \text{mean})\right)^2 / \text{number of tests}}$$

Suspended circuits are treated as a special case in the *belief-factor* algorithm. If a test fails, then all circuits not in the test path have their *belief-factor* set to 10, or positively good, and are eliminated as suspects. If the test passes, then the *belief-factor* of the suspended circuits are not updated, since no information can be gleaned from the tests execution.

Thus, the *belief factor* is a running history of the how robustly each circuit has been tested and the pass or fail outcome of each test. The mean value indicates how close the circuit believes it is to a definitive resolution. The standard deviation indicates how likely the resolution is correct. A mean value close to 10 indicates a high level of confidence that the circuit is functional. As the mean value approaches 0, confidence goes down, though the circuit is believed to be functional. As the value passes 0 and goes negative toward -10, the confidence that the circuit is faulty grows.

To clarify this notion, consider the following example.

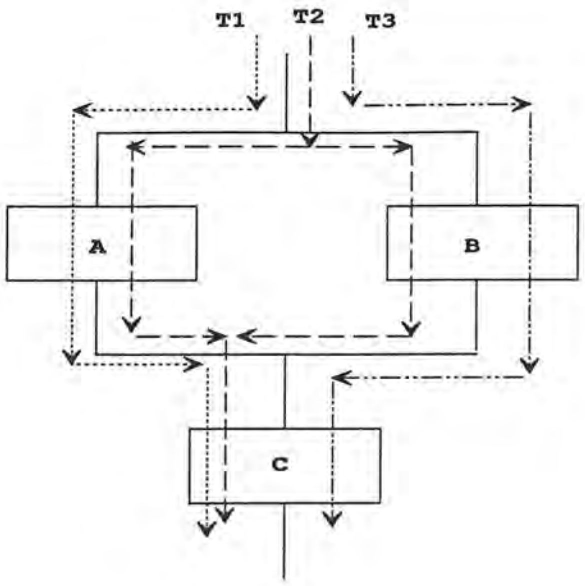

Figure 6. Three circuit example.

This example has three circuits, A, B and C and three tests, T1, T2 and T3, as seen in Figure 6. The circuit/test relationships are as follows:

Circuit	Test	Extent-of-coverage
A	T1 T2	Considerable Moderate
B	T2 T3	Moderate Considerable
C	T1 T2 T3	Moderate Considerable Minimal

Suppose test T1 is executed and it fails. Then, the resulting *belief factor* for each circuit is shown in the following table.

Circuit	Coverage Value	Belief-Factor mean	std dev
A	-7.5	-7.5	0
B	(suspended circuit)	10 (special case)	
C	-5.0	-5.0	0

Now suppose test T3 is executed and it passes. The resulting *belief factor* for each circuit is shown in the following table.

Circuit	Coverage Value	Belief-Factor mean	std dev
A	(suspended circuit)	-7.50	0
B	N/A (not suspect)	10	
C	2.5	-1.25	2.65

Lastly, suppose test T2 is executed and it fails. Then, the resulting *belief-factor* for each circuit is shown in the following table.

Circuit	Coverage Value	Belief-Factor mean	std dev
A	-5.0	-6.25	0.88
B	N/A (not suspect)	10	
C	-7.5	-3.33	0.24

In the final results, circuit A has the most negative *belief-factor*, so it is the most likely the fault. Circuit C is the next most likely since it has the next most negative *belief-factor*, and circuit B is eliminated as a suspect.

4.3 Inference engine modes of operation

The inference engine operates in four different modes depending on the current state of the search for the fault. These modes are :

1. Find the initial fault,
2. Confirm the suspected fault,
3. Eliminate as many suspects as possible, and
4. Eliminate the next most-likely-suspects.

Each of these modes is explained below.

4.3.1 Find Initial Fault Mode.

Experdite starts in the *find initial fault* mode. The purpose of this mode is to detect a fault in the fewest tests possible and to reduce the suspect list to a plausible subset[9]. Initially, all components on the circuit board are suspected equally of being at fault. The problem, then, is to establish a starting point.

A starting point is determined by emulating the methodology of a senior technician. When initially trying to find a fault, s/he will typically execute a test that closely examines the more universal circuits, while also covering as much breadth of the circuitry as possible [7],[13]. The most "universal circuit: means the circuit with the most connections to other circuits and has the most test vectors pass through it. Testing the system this way allows early detection of faulty universal circuits, while also testing as much of the system as possible.

Test selection in this mode is based on two factors. The selected test has

1. the highest *extent-of-coverage* for the most universal circuit and
2. the greatest *breadth-of-coverage.*

The *breadth-of-coverage* is based on the number of circuits in a test path. It indicates how encompassing the test is over the entire circuit board. Each test knows its own breadth-of-coverage value.

The inference engine continues in this mode, selecting tests to be run until either a fault is discovered, or the circuit board is deemed healthy. For each selection, a less universal circuit than the previous is chosen as the target of testing. This means Experdite is able to declare, with reasonable assurance, that the circuit board is fault free, without having to exhaust all tests.

If a fault is detected, then the inference engine switches to the second mode, *confirm the suspected fault.* If the board is deemed fault free, the user is so informed and the program exits.

4.3.2 Confirm the suspected fault mode

The *confirm the suspected fault* mode determines the most likely cause of the fault and then chooses the best test to confirm this suspicion. The idea is to select test with the highest *extent-of-coverage* value for the suspect and the most narrow scope. The emphasis is placed on selecting the test with the highest *extent-of-coverage* value. Scope refers to the number of circuits in the test path and a narrow scope means the fewest number of circuits in the test path. The high extent-of-coverage value provides reasonable confidence that the suspect will be thoroughly checked. The narrow scope ensures minimal interference from other components and the potential for maximum suspect list pruning.

If the test fails, then the suspect is confirmed and, due to the narrow scope of the test, a large number of circuits are suspended. Thus, the suspect space is greatly reduced and the inference engine moves on to the third mode, *Eliminate as many suspects as possible.*

If, on the other hand, the test passes, and if the most likely suspect changes, then the mode is not switched and this same operation is performed on the new suspect. If, however, the most likely suspect does not change, then the inference engine is switched to the third mode, *eliminate as many suspects as possible.*

4.3.3 Eliminate as many suspects as possible mode.

The *eliminate as many suspects as possible* mode is similar to the previous mode, in that the suspect is kept the focus of the search. This mode differs in that emphasis is placed on eliminating as many other suspects as possible, rather than confirming the fault. The test selection criteria are

1. the most likely suspect must always be in the test path and
2. the test path cannot contain more than 50% of the total circuits.

This way, if the test fails, as is predicted, then a large number of circuits are eliminated from the suspect list.

The inference engine remains in this mode until a solution is found, until no test meets the test selection criteria, or until the most-likely-suspect changes. In the first case, the solution is reported and the program terminates. In the next case, the inference engine is switched into the third mode, *eliminate the next most likely suspects.* In the last case, the mode is switched back to second mode. *confirm the suspected fault.*

4.3.4 Eliminate the next most likely suspects mode.

The object of the *eliminate the next most likely suspect*s mode is to eliminate the closest rivals to primary suspect. A test is chosen so that either the most-likely or next-most-likely suspect is in the test path, but not the other. Then, if the test fails, the suspended circuit is eliminated. If the test passes, then unless the test had an absolute extent-of-coverage value, neither suspect is eliminated. The belief factors of both circuits are adjusted accordingly and the operation is repeated.

As the next-most-likely suspects are eliminated from consideration, then *next*

next-most-likely suspect is paired against the most-likely suspect. The process continues until either :

1. a solution is found,
2. all the tests that can provide any useful information are run,
3. the belief in the most-likely suspect far outweighs its closest rival or
4. the most likely suspect changes.

Cases 1, 2 and 3 indicate the end of the diagnostic session. A best guess as to the cause of the fault is made at this time. In case 4, the inference engine is switched back to the second mode, *confirm the suspected fault,* and the process starts over.

4.4 Look-aside rule operation

As mentioned in the introduction, a look-aside rule is used to handle special cases where the model based method fails. Each test can have one or more look-aside rules attached to it. If a test fails, then all its attached rules are activated. If all the premises of a rule are satisfied, then the circuit specified in the conclusion is reported as the cause of the fault and the diagnostic session ends. Tests with multiple rules activate the rules one by one until either the rules are exhausted or a rule is satisfied. Refer to Appendix B for the rule syntax and definitions.

4.5 Criteria for a solution.

Experdite checks the suspect list entries to determine if the criteria to declare a solution are met after every test execution. If met, Experdite reports one of two possible solutions, either the circuit board is fully functional or the best guess as to the cause of the fault.

The solution criteria are as follows:

1. If the belief-factor of every circuit is positive, meaning each circuit believes it is operable and each circuit has been verified by at least one test with an *absolute* or *considerable* extent-of-coverage value, then circuit board is judged fault-free.

2. If all the premises of a look-aside rule are satisfied, then the circuit in the conclusion is reported to be the cause of the fault.

3. If only one circuit remains in the suspect list, then, obviously, that circuit is the most likely and only cause for the fault.

4. If all tests have been exhausted, then the most-likely circuit is reported as the prime suspect.

5. If the most-likely suspect has no tests remaining whose extent-of-coverage is greater than Minimal, and all the other suspected circuits within one *belief-factor-window*[2] of the most-likely suspect also have no outstanding tests with an extent-of-coverage value greater than Minimal, then the most likely suspected is declared to be the fault.

For example, suppose circuit A is the most likely suspect with a belief-factor of -8.0 and all tests associated with A with an extent-of-coverage value greater than Minimal have been run. Let circuit B be the only other remaining circuit in the suspect list. Suppose Circuit B has a belief-factor of -7.0, which is within one *belief-factor-window* of the most-likely suspect. Also, suppose test T2 checks circuit B *considerably* and has yet to be run. Then after test T2 is run, the *belief-factor* for each circuit is updated accordingly. At this point, a solution is reported because criteria 4 and 5 are satisfied.

[2] A belief-factor window is the area between an enumerated belief-factor range. For example the area between *bad* and *probably bad* is a window, or a value of 2.5.

4.6 Reporting the fault

Since no circuit is tested in complete isolation, making a determination of the fault often resorts to a best guess. Since each circuit continually calculates its own *belief-factor* based on the partial information gathered during the entire diagnostic session, Experdite's best guess simply falls out as the circuit with the most negative belief-factor.[3]

If the user desires, Experdite reports the next best guesses simply by traversing the suspect list in most negative *belief-factor* order. This provides the user with additional information in case the best guess is incorrect.

[3] In the case of belief-factors tying, the push is given the circuit with the smallest standard deviation.

5. DISCUSSION OF EXPERIMENTAL TEST RESULTS

5.1 Overview

In order to evaluate the effectiveness of this project's methodology, Experdite was implemented and applied to a real, production circuit board. The objective of the evaluation was to determine how accurately and timely this diagnostic system locates faults on the circuit board.

Faults were induced randomly on the circuit board. They represent characteristic defects, such as lines stuck high and low, shorted lines, open lines, etc. Faults with the potential of causing permanent damage to the system were disallowed.

5.2 Hardware system description.

Before the results are presented, a brief background on the hardware system. The circuit board used in the experiment is called the Picture Processor 2, or PP2, from the Tektronix 4230 series color graphic workstations. The PP2 is one of three major subsystems that comprise the workstation, as seen in Figure 7. Its purpose is to convert graphic display list commands generated by the Control Processor into explicit pixel address and data information. The PP2 writes the pixel information into the Frame Buffer. The Frame Buffer, in turn, stores and displays the graphic image. The Control Processor, CP, is the primary compute engine in the system. It provides the user interface and is the self-test dispatcher and controller for PP2.

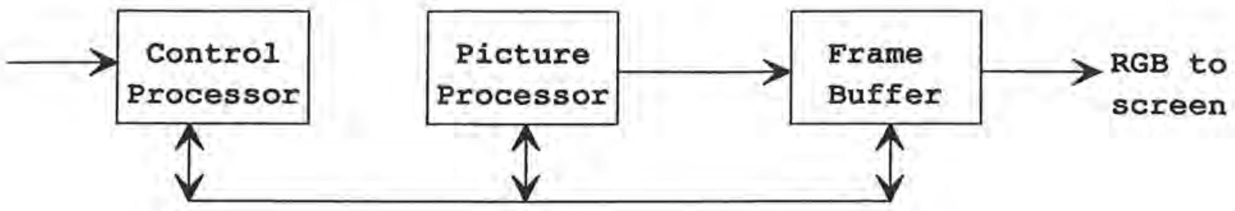

Figure 7. Hardware architecture for Tektronix 4230 graphic workstation.

The PP2 has two bit-sliced, micro-programmable graphic engines on the board. For this experiment, only one of the two bit-slice engines is modeled. This portion of the circuitry is made up of approximately 200 separate integrated circuits, which are divided into 32 distinct sub-systems. There are 22 distinct self-tests used to verify this portion of the PP2 hardware. The block diagram of the PP2 data path is shown in Figure 8 and of the PP2 control path in Figure 9[14]. The TDL listing for this circuit board is in Appendix C.

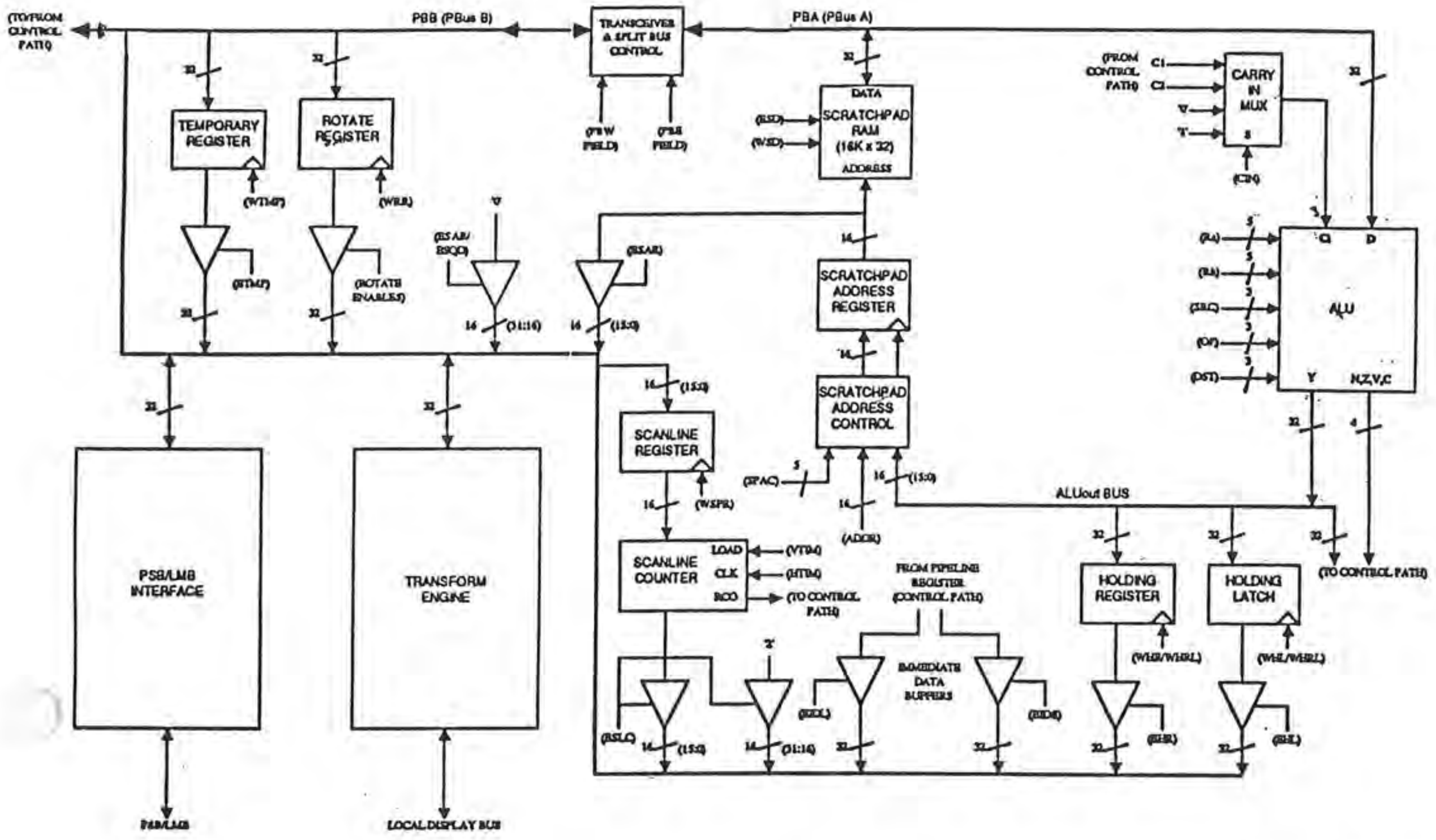

Figure 8. PP2 data path block diagram.

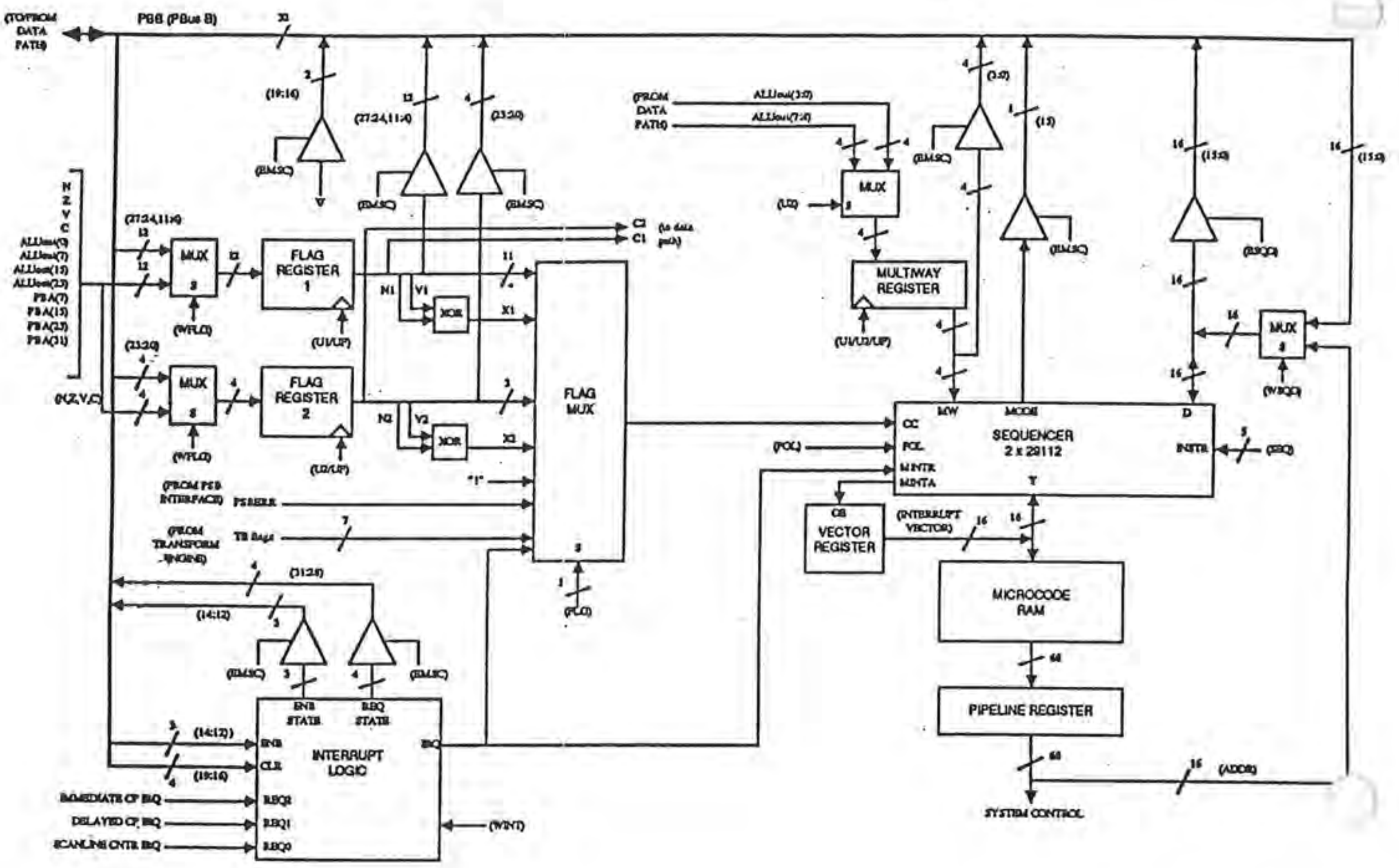

Figure 9. PP2 control path block diagram.

5.3 Experimental results.

In each experimental case, the following information is reported:

1. The induced fault and its expected symptoms.

2. The number of tests required for Experdite to make a decision.

3. Experdite's reported solution.

4. Accuracy of results, where an indication of whether the fault was identified correctly, completely erroneously, or identified as the second or third best guess.

Case	Induced Fault	Number tests requested	Reported best guess	Accuracy
1	ALU neg flag output stuck low	14	ALU flag output	Correctly identified
2	ALU Y output bit 11 stuck low	12	ALU Y output	Correctly identified
3	Scratch RAM addr Control output stuck low	8	Scratch Ram Addr input	Correctly identified
4	Scratch RAM output data lines shorted	4	Scratch Ram Output	Correctly identified
5	Holding Register always enabled	4	ALU D input register	3rd best guess
6	Holding Register always disabled	5	Holding Register	Correctly identified
7	Pbus bit 29 held hi	4	ALU D input register	3rd best guess
8	Disable Rotate register	9	Rotate Register	Correctly identified
9	Disable Temp register	7	Temp Register	Correctly identified
10	Pbus source mux disabled	7	Temp Register	2nd best guess
11	Multiway register output stuck low	7	Multiway Register	Correctly identified.
12	Flag bit into sequencer stuck low	5	Scratch RAM Output	Incorrectly identified
13	Misc input to flag mux stuck low	5	Flag reg output	Incorrectly identified
14	Disable flag reg output	5	Flag reg Output	Correctly identified.
15	Sequencer D input line stuck low	6	Scratch RAM output	Incorrectly identified

Case	Induced Fault	Number tests requested	Reported best guess	Accuracy
16	Disable sequencer mode switching	9	Sequencer Output	Correctly identified.
17	Disable holding latch	4	Holding Latch	Correctly identified
18	Disable pipeline MS data output	7	Temp	3rd best
19	Disable pipeline LS data output	7	Temp	3rd best
20	Scratch RAM Addr register carry bit stuck low	6	Scratch Ram Addr output	Correctly identified
21	Scratch RAM Addr control line stuck low	6	Scratch Ram	Incorrectly identified

5.4 Analysis of results.

The PP2 circuit board uses 22 different self-tests to check 32 distinct sub-systems on the board. The experiment consisted of inducing faults on the board and using Experdite to diagnose the fault. As shown in the experimental results in the previous section, Experdite is capable of diagnosing faults on a real, complex circuit board.

The experiment consisted of 21 separate test cases. Experdite correctly reported 17 of the faults when up to the third best guess is included in the total. Experdite was slightly less accurate, a total of 12, when counting reports which were correct on the first guess. Only four cases were reported entirely incorrectly. These statistics show the system is indeed capable of preforming diagnostic analysis of faults, though the accuracy is less than desired.

Two of Experdite's incorrect reports are due to an inaccurate TDL description of two pieces of the hardware. In this particular case, the faulty data was fed through a another circuit back to faulty circuit. This circular behavior proved to be difficult to accurately portray in the TDL. Therefore, when a TDL description is generated, extraordinary analysis must be made when describing feedback loops. Some thought how to cleanly handle such circuits in TDL and the inference engine is also needed.

The remaining two cases reporting incorrect faults had unusual test reports. By making these unusual test results into *test result* premises and combining them with a *measurement* premise, a look aside rule could be generated to correct the model based deficiency. A test result premise alone is inadequate due to the large number of circuits on the circuit board.

Experdite averaged 6.5 requests for a test to be run for it to determine a result. The minimum number of requested tests was 4 and the maximum was 14. The ability of Experdite to generate a solution in such a short order is advantageous in that it lessens the amount of time required to troubleshoot a circuit board.

The response time of Experdite, from entering the test result until Experdite recommended a new test or produced the final suspect list, was consistently less than a second. This is an adequate response time for human interaction.

6. SUGGESTIONS FOR FUTURE WORK.

This section covers suggestions for further work and ideas of what can be done to expand Experdite's capabilities.

1. The design and implementation of a circuit boards self-tests must take Experdite's operation into consideration. Feedback loops, which are difficult to model, should be minimized. Addition of a small amount of hardware and better test design could increase Experdite's accuracy rate, thereby offsetting any additional hardware costs.

2. The TDL model was intentionally made to simplify the modelling of a circuit board. The circuit/test behavior is the important entry in the TDL. The actual circuit behavior was irrelevant. But to increase the accuracy rate, the extent of coverage value must be expanded beyond a mere percentage. It is necessary to add some indication of <u>what</u> percentage of the circuit is checked by each test. For instance, consider two tests that both check 50% of the same circuit. The problem is, which 50% was checked. Did the tests overlap? Is 50% + 50% <= 100%?

3. Make better use of the cost/benefit ratio for each test. A standardized point system for each cost and benefit category needs to be defined. The TDL writer then has a basis of how to assign number to each category.

4. The production line staff collects statistics on the frequency and cause of the faults discovered during manufacture of circuit boards. Adding a *likelihood-of-failure* field to the TDL circuit description would be useful when determining the most likely suspect.

5. Add capability for the system to automatically record its actions, the results of tests and the fault causes so that it can generate its own rules. Essentially a learning

system.

6. Add a two faceted explanation functionality. First, provide the user with explanations of why and how a test is selected to be run. Second, to tell why and how the current most-likely suspect is selected.

7. Currently, the translated TDL knowledge base must be compiled with the inference engine to make a working system. A more general purpose solution would be to have the inference engine read in and interpret the knowledge base at run time.

8. The current user interface is marginal. A far better solution would be to have the Experdite talk directly to a self-test control program, bypassing the human altogether. This step would require a standard interface to all circuit board self-test systems.

9. Increase the capability of the rules to create a more typical back-chaining approach to their use. That is, when the first look aside rule is fired, if a premise is dependent on other rules, the rules will back-chain in a typical fashion. This feature will allow for other premise types, such as a circuit premise. A circuit premise is "if test circuit A is known to be good". Then, to resolve the question "is the circuit good", Experdite could back-chain through other rules and apply normal constraint suspension.

7. ANALYSIS OF SYSTEM AND CONCLUSIONS.

The first part of this section evaluates Experdite using the valuation criteria set forth by Pau[8]. The second half of this section discusses the practicality of the system, how well Experdite satisfied the project objectives and what is needed to make this a viable, "industrial strength" product.

7.1 Evaluation of Experdite's Test Description Language.

Pau[8] sets forth several evaluation points by which the effectiveness of a high level description language is judged. Each of these criteria are listed below, with a statement of how the TDL measures up.

1. Expressiveness: *Does the representation make distinctions between the concepts represented?* The TDL abstracts the most primitive concepts of a circuit and a test. In addition, TDL represents the notion of a relationship between a circuit and test. Therefore, the basic concepts of a circuit board necessary to do diagnoses are represented.

2. Computational efficiency: *Does the representation scheme allow for efficient computation of various inferences required for the task?* The TDL entries are translated into C++ objects representing each circuit, test and circuit/test relationship, as explained in Appendix A and B. Each object is self-contained as to its status, its ability to make inference decision and the data needed to make the decisions. Thus, when a list of objects are traversed during an operation, each object handles its own requirements. This simplifies the overall operation of the system, remove much overhead and extraneous control functionality.

When the TDL is converted to C++ classes, they are currently held in simple linked lists. A linear search is used during inferences. The small number of items in the lists allowed for efficient computing. However the lists are generated as objects, and can be altered to more efficient algorithm should the knowledge base grew significantly. The changes would be invisible to the remainder of the system.

3. Modifiable: *Is the representation scheme easily modifiable?* Each circuit, test and circuit/test relationship is a wholly contained entity unto itself, as explained in Appendix A. Thus, the high level of abstraction allows easily changes to a piece of the knowledge base, while leaving the remainder of the knowledge base is oblivious to the change.

4. Conciseness: *Is the representation scheme compact, clear, and at the right level of abstraction?* The TDL abstracts a circuit board to the most basic components, the circuit, the self-test and the relationship between the two. The TDL writer must use their own judgement when assigning a judgement to the extent a test checks a circuit and the cost/benefit ratio. While a test's extent-of-coverage value is reasonably straight forward determine, the writer would benefit from a more concise guideline for assigning cost/benefit ratios.

5. Uniform representation: *Can different types of knowledge be expressed with the same general scheme?* The TDL is capable of representing any system that has discrete parts that are tested with tests capable of reporting back at least a pass or fail status. For example, a TDL description of an automobile engine was used during development of the Experdite software. Thus TDL and the inference engine are capable of representing different domains, but only if they fit into the category described above.

6. Easy retrieval and access: *Is the representation scheme such that the desired knowledge can be easily accessed?* The TDL is a high level description language. Its syntax, described in Appendix B, provides a straight forward representation for each element of described by the TDL. Therefore, it is easy to read the TDL description and

determine what circuits make up the board, what tests execute on the circuits, how each test and circuit relate to one another and any special case rules known about the hardware.

7. Multiple level representation of knowledge: *Does the scheme offer different levels of abstraction about the same concept.* The TDL does not meet this criteria. Its primitives are abstracted to the most basic units. It currently has no previsions to handle any other level of abstraction.

In summary, Experdite's Test Description Language satisfy the majority of Pau's representation schemes evaluation criteria. The TDL falls short in that it is limited to representing systems with components and self-tests and that it is limited to only the highest level abstraction. Regarding the first limitation, the number of different domains that are applicable is numerous, so Experdite is still valuable. Regarding the second limitation, if lower level abstractions are added to the TDL, it would allow Experdite to diagnosis to a finer grain, but sacrifice the ability to easily generate, understand and modify the knowledge base.

7.2 Evaluation of Experdite architecture.

Pau[8] asserts a set of standard requirements to evaluate an expert diagnostic system. Each of these benchmarks and how Experdite measured follows.

1. *Minimum non-detection probability* - Before declaring a solution, Experdite requires all but "invisible" circuits to be either tested to a satisfactory level, or be declared good through constraint suspension. Invisible circuits are handled by look-aside rules. Therefore, the system has the capability of detecting nearly every fault.

2. *Minimum false alarm probability* - Each circuit retains its own belief

regarding whether it is good or bad, and to what degree. This belief-factor is acquired over time. Also, the criteria used by Experdite when declaring a solution requires all circuits must have a belief-factor surpassing certain thresholds. Combining the accumulative belief-factor and the criteria for declaring a solution, false alarms are minimized

3. *Minimum detection/test selection time* - Based on the experimental executions of Experdite, the test selection time is faster than can be measured with a stop-watch. Such a short response time is more than adequate for a user interaction. The limiting and most time consuming factor in running the system was the human involvement. This strengthens the argument suggested in the future work section to remove the human interface and have Experdite interact directly with the circuit board self-tests.

4. *Deep knowledge and specific historical knowledge data integration* - Experdite allows for input of both types of knowledge through its behavior and structure model description and its rule generation capabilities.

5. *Capable of handling uncertainties.* Uncertainties in Experdite revolve around the self-tests and how completely a test checks every circuit in its path. A test may pass vectors through a faulty circuit and not trigger the fault condition. Also, since self-tests can indicate only that some circuit in its path failed or all passed, then the exact fault cause is uncertain. Experdite handles the uncertainties by having each circuit accumulate its own belief-factor, based on the extent a test checks the circuit and the pass/fail result of the test. This way, if a test misses a fault in a circuit, other tests will trigger it. The accumulated belief-factor evens out the hits and misses so Experdite can make a correct report.

6. *Expansibility and maintenance.* Maintenance on specific circuit board knowledge bases is easy due to the high level abstraction of the TDL. Also, Experdite, as

a whole, takes advantage of the inherent ease of expansion and maintenance afforded object oriented systems. Therefore, Experdite satisfies this requirement in both the TDL and the inference engine.

In summary, the architecture of Experdite satisfies the requirements set forth by Pau[8] for a diagnostic knowledge based systems.

7.3 Evaluation of Experdite to the specific manufacturing needs.

7.3.1 Analysis of how well Experdite met its primary goal.

The primary goal of Experdite is to provide a practical expert diagnostic system for use by manufacturing personnel. In particular, Experdite is intended to help the junior technician at the board verification station. In the manufacturing scenario which includes Experdite, a junior technician would first use Experdite to narrow the search space to a few square inches of the circuit board and to a few ICs. Then, s/he would be limited to 5 minutes to locate a workmanship error in the search space. If the fault is not be found within the allotted time, the board is sent off-line to the senior technician for repair.

The experimental evidence, detailed in section 5, shows Experdite is capable of correctly identifying a faulty circuit to an accuracy rate of 81, when considering the top three reported suspects. However, if only the most likely suspect is considered, the accuracy rate drops to a 57%.

Since the junior technician's time to find the fault is so limited, it is likely s/he will have time to look only in the area of the most likely suspect. Thus, the 57% accuracy rate is unacceptably low. Further work is required to increase the accuracy rate of Experdite for it to be viable system for the junior technician.

The senior technician at the repair station is not under such tight time constraints and is more likely to consider the second or third guesses reported by Experdite. In conversations with the senior technicians, they expressed a great eagerness to have a tool such as Experdite, even if the accuracy is not 100%. Their concern is that with new circuit boards, each more complex than previous boards, constantly being added to the manufacturing line repertoire, their productivity is decreasing. The senior technicians felt any tool which could increase their productivity is desirable. Of course they would prefer a 100% accuracy, but they stated that using their own expert knowledge in conjunction with Experdite would be an significant in increasing their productivity.[13]

Alas, the product line Experdite was intended for has been recently abandoned and the manufacturing line shut down. This occurred just as the implementation of Experdite was being completed. So, there is no feed back or evaluation from actual users. There has been no time to modify Experdite to be useable on another line.

7.3.2 Analysis of how well Experdite met its secondary goals.

As part of the practical requirements for the manufacturing line, Experdite was to address three special needs of the manufacturing line.

1. The need for an expert diagnostician to be available immediately when a new circuit board enters production.

2. The need to have a general purpose inference engine that can be specialized to any circuit board. This requires a straight forward method of generating a specific circuit board's knowledge base.

3. The need to localize and minimize changes in the knowledge base when reflecting modifications to the hardware circuitry. That is, the knowledge base should be designed such that a change to the hardware need be reflected only in the specific part of the knowledge base corresponding to the changed circuit. The remainder of the

knowledge is untouched.

Experdite address the immediate expert problem by changing from rule based, qualitative type knowledge base to a quantified, behavioral model base. This means the knowledge base uses the expertise of the circuitry designer, rather than the expert troubleshooter. Thus, an expert is available much earlier in time, allowing for a diagnostic system to be built before the board enters production.

The two part design of Experdite, a TDL description of a circuit board and the general purpose inference engine, satisfies the need to have one general system be applicable to many different circuit boards. If, as part of a circuit board's design, the TDL description for a circuit board is generated, then every board in production has its own expert diagnostic system.

The requirement to easily reflect modifications to the hardware is addressed with the TDL. Each circuit, test and circuit/test relationship is a wholly contained entity unto itself, as explained in Appendix A. Thus, when hardware is changed, only the TDL sections affected by the hardware change need to be modified. The high level of abstraction allows for easy changes and the remainder of the knowledge base is oblivious to the change.

A. APPENDIX A - INFERENCE SHELL PROGRAM DESIGN AND IMPLEMENTATION CONSIDERATIONS

A.1 Overview

The design of this system uses an object oriented approach. That is, the system is designed such that, with very few exceptions, all the elements of the system are defined as objects. These objects include circuits, tests, circuit/test relationship, rules, lists, belief-factors and inference engine modes. The only exception is the main control loop of the inference engine.

Each circuit in the hardware system is defined as a *circuit object*. A circuit object is autonomous in that it maintains its own belief-factor with a belief-factor object, its own list of associated tests and its own universal visibility factor. A universal visibility factor is an indication of how many other circuits this circuit is connected to and how many tests include the circuit in their path. Each circuit object wholly contains methods to determine the best test to be run on itself, the ability to update its own belief-factor and report its own status.

Every test in the system is defined as a *test type object*. A test object maintains its own status, is able to invoke its own internal methods to request action, gather data and report the result of its execution.

Each circuit-test relationship is defined as an object. For every test associated with the circuit, this object describes the relationship between the two. The object contains the all important extent-of-coverage and the cost/benefit ratio values.

Rule type objects are made up of premise objects and a conclusion object. The premise objects are divided into two virtual objects, a test premises and measurement premises. The premises are described above in section 3.3 on rule generation. If all premise objects are satisfied, then the method to apply the conclusion object is initiated. The conclusion object reports the solution.

The list type object provides generic list handling operations for all lists within the system. This includes the suspect list (circuit objects), associated test lists (circuit-test relationship objects), rule object lists, premise object lists, and test object lists.

Belief-factor objects are used only by circuit objects. They calculate the belief-factor value and update themselves with the new value. This object also is capable of comparing itself to another belief-factor. The advantage of the belief-factor being an object is that if its algorithm ever requires a change, the change would be invisible to the remainder of the system.

The so called meta-rule objects are wholly contained objects that control the operation of each inference engine mode. Each meta-rule does the appropriate test selection and result analysis, as described in the inference engine mode of operation section, section 4, above. These objects are also capable of setting up the next meta-rule to be fired, based on its exit criteria. Having the operational modes as objects allows for quick and easy changes to the inference engine operation, without affecting any other mode.

B. APPENDIX B - TEST DESCRIPTION LANGUAGE IMPLEMENTATION

B.1 Overview

The intent of this appendix is twofold. First, the presentation of the Test Description Language syntax and grammar. Second, the description of the TDL to C++ Translator structure and implementation.

As stated in a section 3, the Test Description Language, or TDL, provides a simple mechanism to model the structure and behavior of the circuitry and the self-tests for a digital circuit board.

A TDL description of a circuit board consists of four basic pieces of the system: the circuit, the test, the relationship between the circuit and test, and the look-aside rule. Thus, when writing a description, the author follows the TDL syntax to describe each circuit and test associated with the circuit board, each circuit/test relationship and every rule.

For example, consider circuit in Figure 10, which has three circuits, A, B and C, and three tests, T1, T2 and T3. Test T1 passes vectors through circuits A and C, test T2 through circuits A, B and C and T3 through circuits B and C. Thus, the TDL description for such a system consists of a separate entry for each circuit A, B and C, for each test T1, T2 and T3, for each circuit/test relationship A/T1, C/T1, A/T2, B/T2, C/T2, B/T3 and C/T3, and for any pertinent rules.

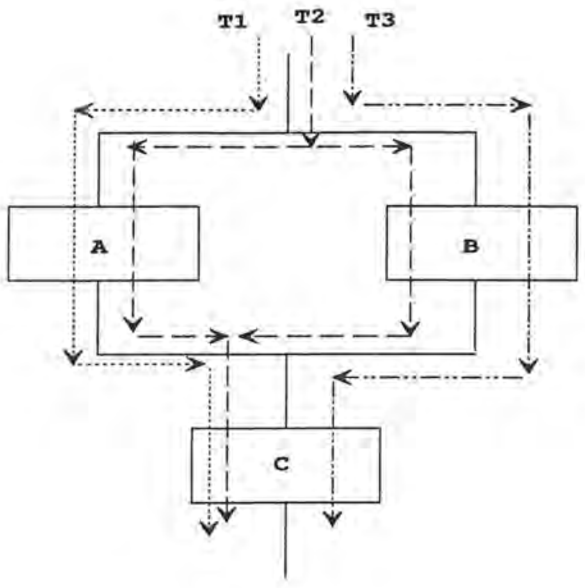

Figure 10. TDL Example

B.1.1 TDL syntax and grammar

A description of the syntax for the circuit, the test, the circuit/test relation and the rule follows.

B.1.1.1 Circuit entry.

The TDL syntax for a circuit starts with the circuit identification token, *CKT*, followed by the name of the circuit, as shown here.

CKT circuit-name.

For instance, the TDL definition for a circuit named "Adder-1" is:

```
CKT Adder-1.
```

B.1.1.2 Test entry.

In a similar manner, the TDL syntax for a test begins with the token, *TEST*, followed by the test name, as shown here.

TEST test-name.

As an example, consider the test named Adder-1-test. The TDL entry is

```
TEST Adder-1-test.
```

B.1.1.3 Circuit/Test relationship entry.

The circuit/test relationship TDL syntax starts with a *REL* token followed by a circuit name. The next items are the test relationship descriptions for each test that passes even a single vector through the circuit. The test relationship consists of the test name, followed by the *extent-of-coverage* value and the *cost/benefit ratio* value that describes the test and circuit association. The circuit and test names must match previously defined circuits and tests. The syntax for a circuit/test relationship entry is:

REL circuit-name (test-name, coverage, cost/bene)

or

REL circuit-name ((test-name, coverage, cost/bene)$^+$)

For example, suppose `Adder-1-test1` and `Adder-1-test2` are tests that check the circuit `Adder-1` with *extent-of-coverage* values of *considerable* and *minimal*, and *cost/benefit* ratios of 2 and 1, respectively. Therefore, the TDL relation entry is

```
REL Adder-1 ((Adder-1-test1, considerable, 2),
             (Adder-1-test2, minimal, 1))
```

B.1.1.4 Rule entry.

A TDL entry for a rule always start with a *RULE* token, followed by the name of the test to which the rule is associated. Only previously defined tests may be used in a rule definition. One or more premises are listed next, followed by the conclusion. The rule syntax is

$$\texttt{RULE test-name ((premise)}^{+}\texttt{; conclusion)}$$

A premise begins with the premise type token, *TP* or *MP*, indicating a test or measurement type premise respectively. A text string containing the body of the rule follows the token. The syntax of the test and measurement premises are :

TP, "test premise string"

and

MP, "measurement premise string"

The body of a test premise describes a specific test result. To satisfy the premise, the result from the actual test execution must match the test result registered in the premise. For example, suppose test T1 reports bits 0-3 are stuck high. Let this be an indication of a specific fault condition existing in the hardware. The premise would then read

```
TP,"Bits 0-3 always high".
```

The rule handler in the inference engine uses the string to generate a conditional statement to the user. The user confirms or denies the condition, which satisfies or rejects the premise.

The measurement premise body specifies the circuit to be measured, what to measure and the expected condition. An example of the measurement premise is

```
MP, "Pin 3 on circuit A1 oscillating at 16MHz".
```

Again, the inference engine poses a question to the user that is constructed around the string portion of the premise. A positive reply from the user, based on his/her measurement of the hardware, satisfies the premise.

The conclusion is the name of the faulty circuit. If all rule's premises are satisfied, the circuit listed in the conclusion is declared faulty.

The following example is the complete rule, using the above premise examples.

```
RULE Adder-1-test1(
     (TP,"Bits 0-3 always high"),
     (MP, "Pin 3 on circuit A1 oscillating at 16MHz");
     Adder-1)
```

B.1.2 Context free grammar

The set of productions defining the TDL Context Free Grammar is listed here. Following traditional syntax, upper case letters identify tokens and lower case characters for non-terminals.

```
start:    exprs

exprs : /* empty */
      | exprs expr;

expr: test_expr
    | ckt_expr
    | rel_expr
    | rule_expr;

ckt_expr: CKT ckt_name;

test_expr: TEST test-name;

rel_expr: REL ckt_name ( test_coverage_expr)

test_coverage_expr:
           test_name,test_coverage,cost_bene_ratio
         | (test_coverage_expr)
         | (test_coverage_expr),test_coverage_expr

rule_expr : RULE test-name ((premise_expr);
                             conclusion)

premise_expr: premise
            | premise , premise_expr
```

```
premise: TEST_PREMISE, STRING_ID
       | MEAS_PREMISE, STRING_ID

conclusion : ckt_name

test_name : ID

ckt_name : ID

cbr : number

number : DIGIT
       | number DIGIT

test_coverage:  ABSOLUTE
              | CONSIDERABLE
              | MODERATE
              | MINIMUM
```

B.1.3 Translation from TDL to C++

The circuit board TDL description is applied to *a TDL translator* to generate a knowledge base for use by the inference engine. The translator parses the TDL description and creates a knowledge base consisting of a set of C++ classes. A C++ class is produced for each CKT, TEST, REL or RULE entry in the TDL description. The knowledge base is then compiled with the inference engine shell code. The result is an expert diagnostic system specific to the described circuit board.

The TDL translator consists of two main parts, the lexical analyzer and the parser, each described below.

B.1.3.1 Lexical Analyzer.

The lexical analyzer converts the stream of characters from the TDL description into tokens and feeds them into the parser. The lexical analyzer also captures specific character strings and/or number values needed by the parser[15].

The UNIX *lex* utility is used to build the lexical analyzer. The input to *Lex* is a table of regular expressions and their corresponding rules. *Lex* produces a program fragment containing the lexical analyzer. The parser incorporates the fragment as its system front end[16].

The input to *lex* consists of two main parts, the *definitions* section and the *lexical rules* section. The *definitions* section contains the set of valid regular expressions for the Test Description Language. The *rules* section defines the actions taken upon matching a regular expression. There is a one to one correspondence between a regular expression and a rule. Descriptions of each section follow.

The *definition* section, listed below, contains the regular expressions that defines the TDL. Each expression is case insensitive.

```
letter [A-Za-z]
digit [0-9]
ckt_token_id [cC][kK][tT]
test_token_id [tT][eE][sS][tT]
rel_token_id [rR][eE][lL]
absolute_token_id [aA][bB][sS][oO][lL][uU][tT][eE]
consid_token_id
[cC][oO][nN][sS][iI][dD][eE][rR][aA][bB][lL][eE]
moderate_token_id [mM][oO][dD][eE][rR][aA][tT][eE]
minimal_token_id [mM][iI][nN][iI][mM][aA][lL]
rule_token_id [rR][uU][lL][eE]
test_premise_token_id [tT][pP]
measurment_premise_token_id [mM][pP]
```

Upon recognition of one character in each set of braces for any one definition, the lexical analyzer designates a token id to the parser. For instance, "ckt", "Ckt" or "CKT" are all possible strings that identify a *ckt_token_id.*

The *rule* section, shown in the following list, gives the lexical rule definitions for each regular expression. Upon recognition of a regular expression, its associated rule is invoked. Most of the rules simply return a token identifying the matched regular expression. Some cases require setting of variables or copying of input strings.

```
{ckt_token_id} return(CKT);

{test_token_id} return(TEST);

{rel_token_id} return(REL);

{absolute_token_id} {
                      coverage_value = ABSOLUTE;
                      return(ABSOLUTE);
                    }

{consid_token_id} {
                    coverage_value = CONSIDERABLE;
                    return(CONSIDERABLE);
                  }

{moderate_token_id} {
                      coverage_value = MODERATE;
                      return(MODERATE);
                    }

{minimal_token_id} {
                     coverage_value = MINIMAL;
                     return(MINIMAL);
                   }

{rule_token_id} { return (RULE)}

{test_premise_token_id} { return (TEST_PREMISE)}

{measurement_premise_token_id} {
                      return (MEASUREMENT_PREMISE)
                              }

[A-Za-z][A-Za-z0-9_]* {
                         strcpy (str_ptr, yytext);
                         return(ID);
                       }

\"[A-Za-z][A-Za-z0-9_ ]* {
                         strcpy (str_ptr, yytext);
                         return(STRING_ID);
                         }

[0-9] {
        value = yytext[0] - '0';
        return(DIGIT);
      }
    ")" return(RIGHT_PAREN);
    "(" return(LEFT_PAREN);
    "," return(COMMA);
```

B.1.3.2 Parser

The parser groups the incoming stream of tokens from the lexical analyzer into grammatical phrases, as defined by the language[15]. Then, after an semantic check on the phrases, the parser generates a set of C++ classes reflecting the original TDL data.

The UNIX *yacc* utility is used to create the parser. *Yacc* takes the user defined specification for a language and produces a program capable of parsing the specified language[17].

Three main sections, the *declarations*, the *rules* and the *programs*, make up the *yacc* format. The *declarations* section contains the definitions of the tokens used in the *rules* section and the C data structures used in the *programs* section. The *rules* section contain the grammer rules specifying the language to be parsed. Actions are invoked upon recognition of a rule. The action invocation results in function calls to the C routines in the *programs* section.

B.1.3.3 Yacc rule section for TDL.

This section lists the TDL grammer, including the action items. The grammer is labeled as `grammer`, while an action is indicated as *ACTION*

```
start :
      {
        INIT_LISTS();
      }
```

```
         exprs
         {
           GENERATE_CODE();
           RECORD_NUM_CKTS();
         }

exprs :
         /* empty */
         | exprs expr;

expr :
           test_expr
         | ckt_expr
         | rel_expr
         | rule_expr;

ckt_expr :
         CKT ckt_name
         {
           INIT_CKT(CKT_STR);
         };

test_expr :
         TEST test_name
         {
           INIT_TEST(TEST_STR);
         };

rel_expr :
         REL ckt_name LEFT_PAREN test_coverage_expr
         {
           ASSOCIATE_TESTS_WITH_CKT(CKT_STR);
         };

test_coverage_expr :
         {NUM_VALUE = 0;}
         test_name COMMA test_coverage COMMA cbr
         {
           INIT_RELATION(CKT_STR, TEST_STR, COVERAGE_VALUE,
                                NUM_VALUE);
         }
         | LEFT_PAREN test_coverage_expr RIGHT_PAREN
         | LEFT_PAREN test_coverage_expr RIGHT_PAREN COMMA
           test_coverage_expr;

rule_expr :
         RULE test_name
         {CURRENT_RULE = INIT_RULE(TEST_STR);}
         LEFT_PAREN
            LEFT_PAREN
              premise_expr
            RIGHT_PAREN SEMI_COLON
            conclusion
         RIGHT_PAREN
```

```
          {FINISH_RULE (CURRENT_RULE);};

premise_expr :
          premise
        | premise COMMA premise_expr;

premise :
        TEST_PREMISE COMMA STRING_ID
        {
           STRCPY(PREMISE_STR, STR_PTR);
           ADD_TO_PREMISE_LIST(TP_TYPE, PREMISE_STR);
        }
        | MEASUREMENT_PREMISE COMMA STRING_ID
        {
           STRCPY(PREMISE_STR, STR_PTR);
           ADD_TO_PREMISE_LIST(MP_TYPE, PREMISE_STR);
        };

conclusion :
        ckt_name
        {
           GENERATE_CONCLUSION (CKT_STR);
        };

test_name : ID {STRCPY(TEST_STR, STR_PTR);

ckt_name : ID {STRCPY(CKT_STR, STR_PTR);

cbr : number;

number :
        DIGIT
        {
          NUM_VALUE = VALUE;
        }
        | number DIGIT
        {
          NUM_VALUE = (NUM_VALUE*10) + VALUE;
        };

test_coverage :
          ABSOLUTE
        | CONSIDERABLE
        | MODERATE
        | MINIMAL;
```

B.1.3.4 Example.

This section shows an example of a TDL description and the C++ object knowledge base that results from the translation. Consider the circuit shown in Figure

11. The system has three circuits, A, B and C, and three tests T1, T2 and T3. The test coverage and the cost benefit rations for each circuit/test relation is shown in Table 3.

Circuit	Test	Extent-of-checking	Cost/ Benefit
A	T1 T2	Considerable Moderate	1 2
B	T2 T3	Absolute Moderate	3 1
C	T1 T2 T3	Considerable Moderate Moderate	8 1 1

Table 3. TDL example circuit setup.

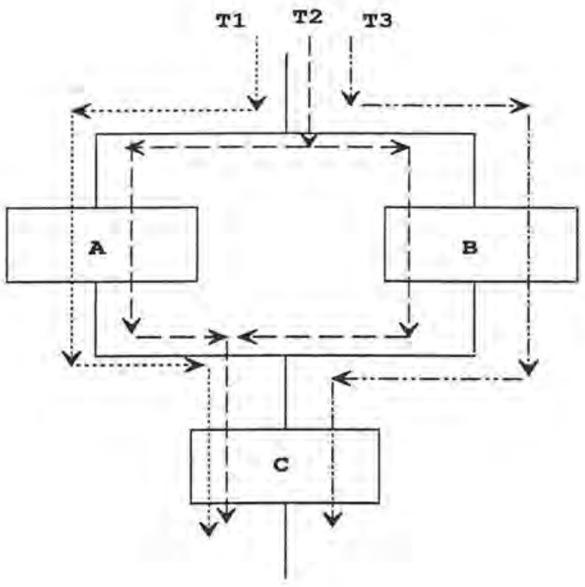

Figure 11. Circuit for TDL walk through example.

Using the syntax definition above, the TDL description of this circuit follows.

```
CKT A
CKT B
CKT C

TEST T1
TEST T2
TEST T3

REL A ((T1, Considerable,1),(T2, Moderate,2))
REL B ((T2, absolute,3),(T3, Moderate, 1))
REL C ((T1, Considerable,8),
       (T2, Moderate,1),
       (T3, Moderate,1))
```

After feeding the TDL description to the translator, the resulting output is as follows.

```
ckt_class * A;
char * A_str = "A";
ckt_test_relation_list * A__test_suite;

ckt_class * B;
char * B_str = "B";
ckt_test_relation_list * B__test_suite;

ckt_class * C;
char * C_str = "C";
ckt_test_relation_list * C__test_suite;

test_class  * T1;
char * T1_str = "T1 test";

test_class  * T2;
char * T2_str = "T2 test";

test_class  * T3;
char * T3_str = "T3 test";

ckt_test_relation * A__T1__c_t_rel;
ckt_test_relation * A__T2__c_t_rel;
ckt_test_relation * B__T2__c_t_rel;
ckt_test_relation * B__T3__c_t_rel;
ckt_test_relation * C__T1__c_t_rel;
ckt_test_relation * C__T2__c_t_rel;
ckt_test_relation * C__T3__c_t_rel;


void ie_initialize_ckt_desc_and_list (void)
```

```
{
   A = new ckt_class(A_str,
                     A__test_suite,
                     2);
   B = new ckt_class(B_str,
                     B__test_suite,
                     2);
   C = new ckt_class(C_str,
                     C__test_suite,
                     3);

   /*
   * Init the suspect list with all ckts in the
system
   */
   suspect_list->insert((char *) A);
   suspect_list->insert((char *) B);
   suspect_list->insert((char *) C);

}/* ie_initialize_ckt_desc_and_list */



void ie_initialize_test_suites (void)

{

A__test_suite = new ckt_test_relation_list;
B__test_suite = new ckt_test_relation_list;
C__test_suite = new ckt_test_relation_list;

}/* ie_initialize_test_suites */



void ie_initialize_test_and_list (void)

{

   /*
    * Initialization of tests in system
    */

   T1 = new test_class(T1_str, 2);
   T2 = new test_class(T2_str, 3);
   T3 = new test_class(T3_str, 2);

   /*
   * Initialization of the tests in system
   */
   all_test_list->insert((char *) T1);
```

```
    all_test_list->insert((char *) T2);
    all_test_list->insert((char *) T3);

}/* ie_initialize_test_and_list */



void ie_initialize_ckt_test_relation (void)

{
  A__T1__c_t_rel = new ckt_test_relation(
                                  T1,
                                  CONSIDERABLE,
                                  1);
  A__T2__c_t_rel = new ckt_test_relation(
                                  T2,
                                  MODERATE,
                                  2);
  B__T2__c_t_rel = new ckt_test_relation(
                                  T2,
                                  ABSOLUTE,
                                  3);
  B__T3__c_t_rel = new ckt_test_relation(
                                  T3,
                                  MODERATE,
                                  1);
  C__T1__c_t_rel = new ckt_test_relation(
                                  T1,
                                  CONSIDERABLE,
                                  8);
  C__T2__c_t_rel = new ckt_test_relation(
                                  T2,
                                  MODERATE,
                                  1);
  C__T3__c_t_rel = new ckt_test_relation(
                                  T3,
                                  MODERATE,
                                  1);

    /*
    * Test Suite definitions for all ckts
    */
    A__test_suite->insert((char *) A__T1__c_t_rel);
    A__test_suite->insert((char *) A__T2__c_t_rel);
    B__test_suite->insert((char *) B__T2__c_t_rel);
    B__test_suite->insert((char *) B__T3__c_t_rel);
    C__test_suite->insert((char *) C__T1__c_t_rel);
    C__test_suite->insert((char *) C__T2__c_t_rel);
    C__test_suite->insert((char *) C__T3__c_t_rel);

}/* ie_initialize_ckt_test_relation */
```

C. APPENDIX C - TDL DESCRIPTION OF EXPERIMENTAL SYSTEM

This appendix lists the TDL description of the PP2 board described in section 5.2.

```
CKT alu_carry_in_ckt
CKT alu_d_in_ckt
CKT alu_flag_out_ckt
CKT alu_reg_ckt
CKT alu_y_out_ckt
CKT flag_mux_no_emsc_ckt
CKT flag_reg_out_ckt
CKT flag_reg_in_mux_out_ckt
CKT flag_reg_in_pbus_ckt
CKT flag_reg_in_alu_ckt
CKT flag_reg_ctrl_ckt
CKT hold_latch_ckt
CKT hold_reg_ckt
CKT int_reg_ckt
CKT multiway_reg_ckt
CKT pbus_ckt
CKT pipe_data_seq_d_ckt
CKT pipe_data_eids_ckt
CKT pipe_data_eidl_ckt
CKT rotate_ckt
CKT seq_d_io_ckt
CKT seq_mode_ckt
CKT seq_stack_ckt
CKT spar_esar_ckt
CKT spar_ram_in_ckt
CKT sp_ctrl_reg_pass_thru_ckt
CKT sp_ctrl_rega_ckt
CKT sp_ctrl_regb_ckt
CKT sp_ctrl_regc_ckt
CKT sp_ram_addr_ckt
CKT sp_ram_data_ckt
CKT tmp_reg_ckt

TEST alu_carry_in_tst
TEST alu_flag_tst
TEST alu_reg_tst
TEST flag_mux_tst
TEST flag_reg_tst
TEST hold_latch_tst
TEST hold_reg_tst
TEST multiway_tst
TEST pbus_tst
TEST rotate_tst
TEST seq_d_io_tst
TEST seq_mode_tst
TEST seq_stack_tst
TEST spar_tst
TEST sp_actrl_rega_tst
TEST sp_bctrl_regb_tst
TEST sp_cctrl_regc_tst
TEST sp_addr_ram_tst
TEST sp_data_ram_tst
TEST int_reg_tst
TEST tmp_reg_tst

REL alu_carry_in_ckt ((alu_carry_in_tst, absolute, 1),
                      (sp_addr_ram_tst, minimal, 1),
                      (sp_data_ram_tst, minimal, 1))

REL alu_d_in_ckt ((alu_carry_in_tst, minimal, 1),
                  (alu_flag_tst, considerable, 1),
                  (alu_reg_tst, minimal, 1),
                  (flag_reg_tst, minimal, 1),
                  (hold_latch_tst, considerable, 1),
                  (hold_reg_tst, considerable, 1),
                  (multiway_tst, minimal, 1),
                  (pbus_tst, considerable, 1),
                  (spar_tst, moderate, 1),
                  (sp_actrl_rega_tst, minimal, 1),
                  (sp_bctrl_regb_tst, minimal, 1),
                  (sp_cctrl_regc_tst, minimal, 1),
                  (sp_addr_ram_tst, minimal, 1),
                  (sp_data_ram_tst, considerable, 1))


REL alu_y_out_ckt ((alu_reg_tst, minimal, 1),
                   (flag_reg_tst, minimal, 1),
                   (hold_latch_tst, considerable, 1),
                   (hold_reg_tst, considerable, 1),
                   (pbus_tst, considerable, 1),
                   (spar_tst, moderate, 1),
                   (sp_actrl_rega_tst, moderate, 1),
                   (sp_bctrl_regb_tst, moderate, 1),
                   (sp_cctrl_regc_tst, moderate, 1),
                   (sp_addr_ram_tst, moderate, 1),
                   (sp_data_ram_tst, considerable, 1))

REL alu_flag_out_ckt ((alu_carry_in_tst, minimal, 1),
                      (alu_flag_tst, absolute, 1),
                      (sp_addr_ram_tst, minimal, 1),
                      (sp_data_ram_tst, moderate, 1))

REL alu_reg_ckt ((alu_carry_in_tst, minimal, 1),
                 (alu_flag_tst, minimal, 1),
                 (alu_reg_tst, absolute, 1),
                 (flag_reg_tst, minimal, 1),
                 (spar_tst, minimal, 1),
                 (sp_actrl_rega_tst, minimal, 1),
                 (sp_bctrl_regb_tst, minimal, 1),
                 (sp_cctrl_regc_tst, minimal, 1),
                 (sp_addr_ram_tst, minimal, 1),
                (sp_data_ram_tst, moderate, 1))

REL flag_mux_no_emsc_ckt ((flag_mux_tst, absolute, 1),
                (sp_addr_ram_tst, minimal, 1),
                (sp_data_ram_tst, minimal, 1))

REL flag_reg_out_ckt ((alu_carry_in_tst, minimal, 1),
                       (alu_flag_tst, minimal, 1),
                      (flag_mux_tst, considerable, 1),
                      (flag_reg_tst, absolute, 1),
                      (sp_addr_ram_tst, minimal, 1),
                      (sp_data_ram_tst, minimal, 1))

REL flag_reg_in_mux_out_ckt (
                     (alu_carry_in_tst, minimal, 1),
                     (alu_flag_tst, minimal, 1),
                     (flag_mux_tst, absolute, 1),
                     (flag_reg_tst, considerable, 1),
                     (sp_addr_ram_tst, minimal, 1),
                     (sp_data_ram_tst, minimal, 1))

REL flag_reg_in_pbus_ckt (
                     (flag_mux_tst, considerable, 1),
                     (flag_reg_tst, absolute, 1))

REL flag_reg_in_alu_ckt (
                    (alu_carry_in_tst, considerable, 1),
                    (alu_flag_tst, minimal, 1),
                    (sp_addr_ram_tst, minimal, 1),
                    (sp_data_ram_tst, moderate, 1))

REL flag_reg_ctrl_ckt (
                (alu_carry_in_tst, considerable, 1),
                (alu_flag_tst, considerable, 1),
                (flag_mux_tst, considerable, 1),
                (flag_reg_tst, considerable, 1),
                (multiway_tst, considerable, 1),
                (sp_addr_ram_tst, moderate, 1),
                (sp_data_ram_tst, moderate, 1))

REL hold_latch_ckt ((hold_latch_tst, absolute, 1),
                     (sp_addr_ram_tst, moderate, 1),
                     (sp_data_ram_tst, considerable, 1))

REL hold_reg_ckt ((alu_reg_tst, considerable, 1),
                   (hold_reg_tst, absolute, 1),
                   (pbus_tst, considerable, 1))

REL int_reg_ckt (int_reg_tst, absolute, 1)

REL multiway_reg_ckt (multiway_tst, absolute, 1)

REL pbus_ckt ((alu_carry_in_tst, minimal, 1),
              (alu_flag_tst, minimal, 1),
                (alu_reg_tst, considerable, 1),
                (flag_mux_tst, minimal, 1),
                (flag_reg_tst, moderate, 1),
                (hold_latch_tst, considerable, 1),
                (hold_reg_tst, considerable, 1),
                (multiway_tst, minimal, 1),
                (pbus_tst, absolute, 1),
                (rotate_tst, considerable, 1),
                (seq_d_io_tst, moderate, 1),
                (seq_mode_tst, minimal, 1),
                (seq_stack_tst, minimal, 1),
                (spar_tst, moderate, 1),
                (sp_actrl_rega_tst, moderate, 1),
                (sp_bctrl_regb_tst, moderate, 1),
                (sp_cctrl_regc_tst, moderate, 1),
                (sp_addr_ram_tst, moderate, 1),
                (sp_data_ram_tst, considerable, 1),
                (int_reg_tst, moderate, 1),
                (tmp_reg_tst, considerable, 1))

REL pipe_data_seq_d_ckt (
                   (seq_d_io_tst, absolute, 1),
                   (seq_stack_tst, considerable, 1))

REL pipe_data_eids_ckt (
                   (seq_d_io_tst, considerable, 1),
                   (spar_tst, considerable, 1),
                   (sp_actrl_rega_tst, minimal, 1),
                   (sp_bctrl_regb_tst, minimal, 1),
                   (sp_cctrl_regc_tst, minimal, 1),
                   (sp_addr_ram_tst, minimal, 1),
                   (sp_data_ram_tst, minimal, 1),
                   (int_reg_tst, minimal, 1))

REL pipe_data_eidl_ckt (
                   (alu_carry_in_tst, minimal, 1),
                   (alu_flag_tst, considerable, 1),
                   (alu_reg_tst, minimal, 1),
                   (flag_mux_tst, considerable, 1),
                   (flag_reg_tst, minimal, 1),
                   (hold_latch_tst, considerable, 1),
                   (hold_reg_tst, considerable, 1),
                   (multiway_tst, minimal, 1),
                   (pbus_tst, considerable, 1),
                   (rotate_tst, considerable, 1),
                   (sp_data_ram_tst, considerable, 1),
                   (tmp_reg_tst, considerable, 1))

REL rotate_ckt (rotate_tst, absolute, 1)

REL seq_d_io_ckt ((seq_d_io_tst, absolute, 1),
                  (seq_stack_tst, considerable, 1))

REL seq_mode_ckt ((seq_mode_tst, absolute, 1))
REL seq_stack_ckt ((seq_d_io_tst, considerable, 1),
                   (seq_stack_tst, absolute, 1))

REL spar_esar_ckt (
                  (spar_tst, absolute, 1),
                  (sp_actrl_rega_tst, considerable, 1),
                  (sp_bctrl_regb_tst, considerable, 1),
                  (sp_cctrl_regc_tst, considerable, 1))

REL sp_ram_addr_ckt (
                  (sp_addr_ram_tst, absolute, 1),
                  (sp_data_ram_tst, considerable, 1))


REL sp_ram_data_ckt ((sp_addr_ram_tst, moderate, 1),
                     (sp_data_ram_tst, absolute, 1))

REL spar_ram_in_ckt (
                 (spar_tst, absolute, 1),
                 (sp_actrl_rega_tst, considerable, 1),
                 (sp_bctrl_regb_tst, considerable, 1),
                 (sp_cctrl_regc_tst, considerable, 1),
                 (sp_addr_ram_tst, considerable, 1),
                 (sp_data_ram_tst, considerable, 1))

REL sp_ctrl_reg_pass_thru_ckt (spar_tst, absolute, 1)

REL sp_ctrl_rega_ckt (sp_actrl_rega_tst, absolute, 1)

REL sp_ctrl_regb_ckt (
                       (sp_bctrl_regb_tst, absolute, 1),
                       (sp_addr_ram_tst, considerable, 1),
                       (sp_data_ram_tst, considerable, 1))

REL sp_ctrl_regc_ckt (sp_cctrl_regc_tst, absolute, 1)

REL tmp_reg_ckt ((alu_carry_in_tst, minimal, 1),
                 (alu_flag_tst, minimal, 1),
                 (alu_reg_tst, considerable, 1),
                 (flag_mux_tst, minimal, 1),
                 (flag_reg_tst, moderate, 1),
                 (hold_latch_tst, considerable, 1),
                 (hold_reg_tst, considerable, 1),
                 (multiway_tst, minimal, 1),
                 (pbus_tst, considerable, 1),
                 (rotate_tst, considerable, 1),
                 (seq_d_io_tst, moderate, 1),
                 (seq_mode_tst, minimal, 1),
                 (seq_stack_tst, minimal, 1),
                 (spar_tst, moderate, 1),
                 (sp_actrl_rega_tst, moderate, 1),
                 (sp_bctrl_regb_tst, moderate, 1),
                 (sp_cctrl_regc_tst, moderate, 1),
(int_reg_tst, moderate, 1),
(tmp_reg_tst, absolute, 1))
```

BIBLIOGRAPHY

[1] Buchanan, B. and Shortliffe, E.. Rule-Based Expert Systems. Reading, Massachusetts: Addison-Wesley Publishing Company, c. 1984

[2] Johnson, R. and Schwartz, T.. "Hubble scope gets expert help.", Electronic Engineering Times, May 28, 1990, P 27.

[3] Davis, R.. "Diagnostic Reasoning Based on Structure and Behavior", Artificial Intelligence, Vol 24, 1984, P 347-410.

[4] DeKleer, J and Williams, B.. "Diagnosing Multiple Faults", Artificial Intelligence, Vol 32, 1987, P 97-130.

[5] Yen, J.. "GERTIS: A Dempster-Shafer approach to diagnosing hierarchical hypotheses.", Communications of the ACM, Vol 32, Number 5, May 1989, P 573-585.

[6] Tektronix Graphic Workstation Division statistics. John Brownlee. Manufacturing Engineer. Graphics Workstation Division, Tektronix, Wilsonville, OR. Sept 1989.

[7] Fink, P., Lusth, J., and Duran, J.. "A general expert system design for diagnostic problem solving." Proceedings IEEE Workshop of Principles of Knowledge-Based Systems, Dec 3-4, 1984. P 45-52.

[8] Pau, L.F.. "Survey of expert systems for fault detection, test generation and maintenance." Expert Systems, April 1986. Vol 3, No 2. P 100-111

[9] Marques, T.. "A Symptom-Driven Expert System for Isolating and Correcting Network Faults." IEEE Communications Magazine, March 1988, Vol 26, No 3. P6-13.

[10] Horvitz, E.J.. "Reasoning Under Varying and Uncertain Resource Constraints". Automated Reasoning, Introduction and Applications. Wos, L et. al.. P111-116. Englewood Cliffs, NJ: Prentice-Hall Publishing Company, c. 1984

[11] Davis, E.. "Constraint Propagation with Interval Labels". Artificial Intelligence. 1987, No 32. P281-351

[12] Novak, T., Meigs, J.R., Sanford, R.L.. "Development of an Expert System for Diagnosising Component-Level Failures in a Shuttle Car". IEEE Transactions on Industry Applications. July/Aug 1989, Vol 25, No 4. P691-698.

[13] Shafto, D., Phillips, R., Interviews with Senior Technicians, Graphics Workstation Production Line, Tektronix, Inc. Wilsonville, Oregon. Sept, 1989.

[14] Redfield, G., Brown, G.. "4230 Picture Processor Functional Description". Tektronix Inc. Graphics Workstation Division Internal Documentation. July 1987.

[15] Aho, A.V., Sethi, R., Ullman, J.D.. Compilers Principles, Techniques, and Tools. Reading, Massachusetts: Addison-Wesley Publishing Company, c. 1986.

[16] Lesk, M.E., Schmidt, E.. "Lex - A Lexical Analyzer Generator" UNIX Programmer's Supplementary Documents. Volume 1. 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, April, 1986.

[17] Johnson, S.C.. "Yacc - Yet Another Compiler-Compiler" UNIX Programmer's Supplementary Documents. Volume 1. 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, April, 1986.

ACKNOWLEDGEMENTS

The author wishs to thank Mark Strand for rescuing a PP2 from the ashes of GWD. Thanks also to Ron Phillips and Dave Shafto for allowing me to pick their brains about how a technician thinks. A special Thanks to Dr. Bill Bregar for his guidance and ideas. Finally, a special and heartfelt thanks to Janet Maxwell-Tilp for her excellent proofing skills and unwavering support.

ACKNOWLEDGEMENTS

The author wishes to thank Mark Strand for rescuing a PP2 from the ashes of GWD. Thanks also to Ron Phillips and Dave Shafto for allowing me to pick their brains about how a technician thinks. A special Thanks to Dr. Bill Bregar for his guidance and ideas. Finally, a special and heartfelt thanks to Janet Maxwell-Tilp for her excellent proofing skills and unwavering support.