A MODEL BASED DATA VALIDATION SYSTEM

By

James Richard Davis

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

December, 1995

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Janos Sztipanovits, for his support and guidance over the past two years. I also need to thank Dr. Gabor Karsai, who had a great deal of input into my research. If he had not answered all of my questions by forcing me to think the problem through and find my own solution, I would not be at this stage yet. Dr. Csaba Biegl and Dr. Karsai are responsible for the MultiGraph Architecture, without which this research would not have been possible.

Several other members of the Measurement and Computing Systems Laboratory have been a great help through my graduate studies. I would like to especially thank Dr. Ben Abbott, Dr. Ted Bapty, Dr. Amit Misra, and Dr. Akos Ledeczi for all the helpful hints along the way. Mike Moore and Jason Scott have suffered along with me. They also provided a great service by putting up with my stupid jokes.

My family deserves a great deal of credit. Without the encouragement from Mom, Dad, and Jenn, this would not have been possible. Someone had to keep telling me all the late nights would be worth it someday. My Grandfather has also been a source of inspiration for me. I need to thank him for all of his support throughout the years.

Artemis, thanks for pushing me at an early age. I am glad someone made me strive to be my best. Also, Ms. T., thanks for pushing me. Without your encouragement long ago, this would not have been possible.

Finally, I would like to thank Tom Tibbals, Don Malloy, and Greg Hollis of Sverdrup Technology, AEDC Division, for the insight into the problem. Many bad decisions might have been made along the path of this research without their input.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In this thesis, a model based data validation package is introduced. A data validation system takes input data and determines if there are problems with the data. This is done by comparing the data values to the data values generated by some test article system model. In order to facilitate this system design, several key areas were examined. This areas are discussed in detail in this thesis. Briefly, they are the data validation process, the MultiGraph Architecture, re–engineering of legacy software, diagnostic systems, and data dependency analysis. The major problem encountered during this research effort was the extraction of structural information from legacy codes for diagnostic purposes. This drove the development of a data dependency analyzer.

Problem Motivation

Arnold Engineering Development Center (AEDC) is a United States Air Force Base responsible for testing and development of aerospace systems. AEDC is the major testing center for the U.S. Air Force. Testing of both propulsion and aerodynamic systems of aircraft undergoing development occurs there. Supersonic and transonic wind tunnels support the aerodynamic testing of aircraft models; actual engines are tested, at simulated altitude, in one of several test cells available on base. Unlike most military operations, testing at AEDC is performed on both commercial and military aircraft. At AEDC, they also test missile and rocket engines along with performing some X-ray research for the military. By ground testing components under development, the USAF is able to save significant amounts of money. Even ground testing

is expensive, but with ground testing no prototype aircraft can be lost. Also, ground testing allows a test to be stopped if a significant problem arises. Obviously, this cannot be attempted in an actual flight test.

AEDC is always searching for more efficient, and more flexible, methods for testing. This search not only includes new physical tools, but also new software systems to enhance testing. As with any large plant, numerous software systems are used throughout AEDC. By improving these systems, AEDC can decrease the cost of testing and improve the test results supplied to their customer.

This thesis describes a software system built for AEDC to perform steady–state data validation. An existing data validation system has been in use at AEDC for many years. However, the system would be considered outdated by most software engineers. The rest of this chapter is devoted to describing the process of validating data and describing the data validation system AEDC wants to upgrade. Next, the problem of modifying **legacy codes** will be discussed. This is an important problem for this system, due to AEDC's constraints upon the new system. In Chapter III, general diagnostics systems will be discussed. A diagnostic system is one of the major additions required for the data validation system. Chapter IV will describe the modeling method used in producing the new data validation system. Also, the actual implemented system will be described here. In order to solve some problems with configuring the new system, a data dependency analyzer was written. It is outlined in Chapter V. Lastly, Chapter VI will examine the results of this research.

Validating Data

One of the first questions to arise in the course of this project was, "What is a data validation system?" A data validation system plays an extremely important role in the verification of aerospace systems. As part of these verification efforts, testing of

components and full systems must occur. These tests are useless if the data produced during a test are not complete and valid.

Other major users of AEDC's facilities are commercial companies. AEDC has a large investment in capital equipment. Commercial businesses buy testing time to test some of their key systems. For companies such as GE and Pratt-Whitney, testing time and facilities cost tens of thousands of dollars per hour. The only deliverables a company receives from AEDC after these test periods are the data recorded during the test session. If this data is incorrect, the test period was a failure; the test must be repeated.

One of the problems engineers face with testing aerospace systems is the system being tested is often viewed as reliable. An error that appears in a turbine engine test is usually first attributed to a faulty data system or a faulty diagnostics system. The actual test article is assumed to be correct and working. One of the goals of a data validation system is to enable an engineer to better determine the cause of an error, whether it be an acquisition system error or a test article failure.

One of the data validation systems used at AEDC takes steady-state data from a data acquisition system. Figure 1 diagrams the data validation system in use at AEDC. Based on the values of this data and the predicted behavior of the engine, the data is validated. If some data items are not within the expected ranges an error message is generated to inform the engineer of a suspected problem. It then becomes the engineer's responsibility to determine if a problem does exist. He must either: correct the problem; set the faulty data values to don't care states; or stop the test.

Another extremely important question is, "How do we know the values used for validating the data are correct?" Engine testing has produced, over a long period of time, a knowledge base capable of predicting certain parameters' values for an engine. Currently, no engine model can predict the engine's physical parameters before testing

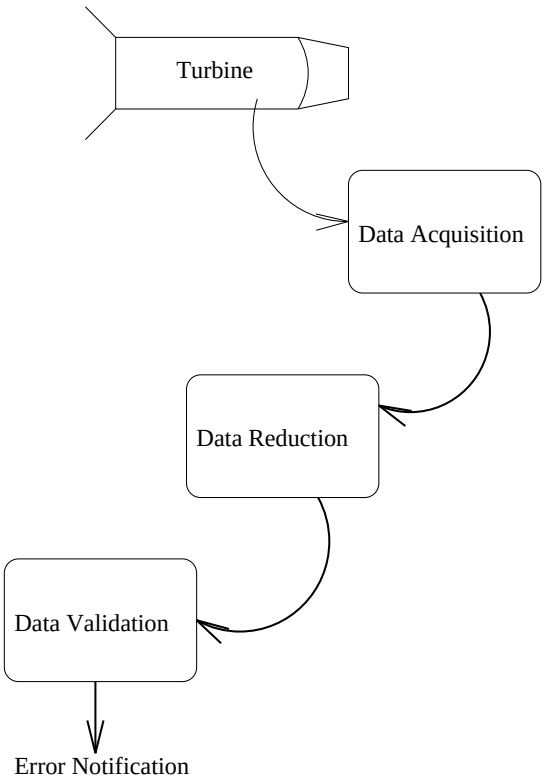

Figure 1. A Data Validation System

has taken place within an acceptable range. After some tests have been performed, adaptive modeling [7] can be used to predict engine behavior. Most of the knowledge about engine behavior is either stored in data recorded from past tests or is available only from an experienced test engineer.

It is the responsibility of the data validation system, in conjunction with the test engineer, to ensure all data transferred to the company sponsoring the test is valid. While the data validation program cannot automatically complete this job, it can enhance the engineer's ability to validate the data. It is important to remember the data validation system is a tool. It is not intended to (and cannot) assume the test engineer's responsibility for ensuring correct data. It can help the test engineer by alerting him to possible problems. This way, the test engineer can concentrate on

his other responsibilities until the need for human interaction with the data validation system arises.

One goal of test engineers is the development of better engine models for data validation. With the combined use of adaptive modeling and a model-based data validation system, a more accurate engine model can be built. This model can then be used to increase the accuracy of the data validation system. As this process iterates, a much more accurate data validation system can be produced. The advantages of a better validation system are evident. Mainly, an engineer can spend less time dealing with false alarms. He can then concentrate on other duties and on solving real problems when the data validation system recognizes them.

Original Data Validation System

Several data systems are in use at AEDC to help validate the test data before shipping to a customer. All data systems referred to in this thesis are data systems for propulsion testing. Their dynamic data system has recently been upgraded to the CADDMAS system that was developed by AEDC, Vanderbilt University, and the University of Tennessee, Space Institute [6]. However, the steady–state and transient data systems currently used are not state–of–the–art systems. Upgrading the current steady–state data validation system was the motivation for the research described in this thesis.

Steady–State Data Validation System

AEDC's current data validation system has been around for many years; the latest date in the source code's comments is around 1988. But, part of the code is Fortran 66, an outdated version of Fortran. The systems architecture is shown in Figure 1. First, several sensors, approximately three thousand, are placed in different parts of

the test article and in the test environment. A data acquisition system monitors these sensors and exports the sensor data to a computer running a data reduction program. Currently, the data reduction program runs on a Convex vector–processor. This program can be extremely large. In the past, this data reduction software has been as large as fifty thousand lines of source code (Fortran). The data reduction program performs numerical analysis on the data to produce information needed to further examine the data. The output of the data reduction program aids the validation of the raw data. After the data reduction process occurs, the reduced data is transferred to another computer running the actual data validation program. This program currently runs on a Digital Equipment Corporation VAX machine and is approximately five thousand lines of source code (also Fortran). Whenever some of the data fails the validation process, an error message is printed on a line printer. This message contains important information with which the test engineer should be able to diagnose engine or data system problems.

While this sounds like a reasonable implementation of a data validation system, there are several problems with the current system. These problems include:

Multiple errors tend to occur at the same time. Diagnosis of the errors becomes more difficult as more errors occur. Determining which errors are causing errors can be a difficult procedure. Also, a printout of fifty error messages can be difficult to analyze in the two minutes between steady–state data points.

Modification of the system is extremely difficult. Currently, the only method for modifying the system is to modify the source code. In order to modify the source, a programmer must be brought in to do the coding. After modifications, extensive testing should occur to ensure system consistency.

Time required for a modification of the system to take effect is unacceptably long. Due to the difficulty of making a modification, an unreasonable amount of time must pass between the realization that a modification is necessary and the actually modified system being brought into use. Usually, a system modification cannot be made during a test period. Therefore, any errors to be corrected by the modifications must be examined by the test engineer to ensure they are not actual errors at each data point. The modification procedure used will be presented later in this thesis.

Addition of new features to the system is virtually impossible. In order to add new features to the system, a complete design process must be undertaken. Even in order to add one more data check to the system, a programmer must be used to update the code. Then, the updated code must be tested prior to actual use. This process is similar to modifying the source code. The major difference is adding new features is a much more significant undertaking and should require much more testing before integrating into the system.

New test engineers have a difficult time diagnosing problems during a test. As engineers become more experienced in validating data, they become much more efficient in diagnosing problems based on empirical evidence. However, the system learning time a new engineer requires is extensive. This is one of the major concerns AEDC has with their current data validation system.

Some features of the current system need to be implemented in the new data validation system. The current system has many problems, but it still has features that AEDC will not allow to disappear in the new system. Two of the major features that must be kept in the new system are:

The current system works. An engineer that has been trained in the use of the current system can validate data on–line. If the new system makes errors in the validation process, the system will not be used. AEDC cannot afford to return faulty data to a customer. Also, the new system should be able to emulate the outputs of the old system. This will allow engineers, who have mastered data validation with the old system, to use the same methods for validating data with the new system.

Even though the reconfiguration of the current system takes a long period of time and much effort, the personnel at AEDC know how to reconfigure the system. If system reconfiguration becomes extremely difficult, they cannot afford to use the new system. One key element here is the software complexity (in terms of system reconfiguration) cannot become so complex as to render AEDC's programmers unable to manage the system.

Making modifications to correct the system's shortcomings without losing those features which are desirable is a significant problem. Several new capabilities are needed for the data validation system. Relevant system requirements include:

A graphical user interface (GUI) is desired. Ease of use for new engineers has become a critical problem that can be solved, in part, by using a GUI. This will help decrease the system learning period for a new engineer.

Large vector computers are no longer being purchased at AEDC. A move to a workstation environment is justified by the transitioning of other software to newly purchased workstations (Silicon Graphics Incorporated Indy 2). By eliminating the vector computer, the system is no longer tied to a proprietary system. This requires writing portable code.

System modification and configuration should be simplified. Ideally, each test engineer can set up the system to his liking. This reduces the time required for modifications to the system to take effect.

Additional information should be made available to the test engineer. A simple printout of an error message does not deliver enough information to an engineer. After receiving the error message, the engineer should be able to look at any and all data in the system. This allows the engineer to see the cause of an error more readily. By having access to all data, the engineer can more accurately determine the actual cause of an error.

Advanced diagnostic features need to be added to the system. A diagnostic system can reduce the number of possible causes of an error into a manageable set. With advanced diagnostics, the chances of mis–diagnosing an error during a test decrease dramatically.

Overall, the system needs to be more user friendly. The goal is to minimize the time difference between a new engineer and a more experienced engineer to validate the same data point. Help menus and better system documentation are a must.

Several different approaches to the development of a new data validation system were considered. After careful thought, a model–based [2] data validation system was decided upon. The more important issues behind this decision will be explained later in this thesis. However, in order to meet the needs of AEDC, we wanted to eliminate as much user training as possible. No one at AEDC wishes to learn a completely new system. Therefore, an effort was made to reuse as much of the old source code in the

new system. This also meant that, for the most part, system configuration changes **could** still be made as before. The next chapter shows the different paths that could be taken toward incorporating the old data validation source code in a modern system.

CHAPTER II

LEGACY SOFTWARE

With the evolution of the computer, industry has relied more and more on computers to perform complex and tedious tasks. Some businesses rely on codes written years ago (**legacy software**) for mission critical tasks. Software techniques and abilities have matured greatly since the design and implementation of some legacy software, but these legacy codes are still used for critical tasks. In order to meet the increased competition in the business world today companies need to have better facilities, including software, than their competitors. One method of improving software is to improve or replace the legacy software currently used by more efficient software [4].

Modifying Legacy Software

Several needs can be met by modifying legacy codes. The three major areas where modifying legacy software can most immediately impact a software system are:

Reuse or reconfiguration;

Addition of features;

Modernization.

Only by understanding the significance of these needs can one begin to understand the decisions made in this research effort. For a problem this complex every decision can effect the overall effectiveness of the system. Now, each area listed above is examined in greater detail.

Reuse or reconfiguration

Reusing existing software is an extremely desirable option for any company needing to expand its current operations. By reusing existing software, no new development projects are needed. While decreasing the time necessary to implement the new system, this method can also greatly reduce risk and cost. If only reconfiguration of the software must occur, no major risk is involved. This is due to the existing software having been previously verified. Reconfiguration should be a much less time and resource consuming task than completely designing and implementing a new system. However, if no low–level understanding of the system exists, a system redesign would be a better choice for system modification. Based on the reduced risk, cost, and time, most companies would rather reuse or reconfigure their existing software base than begin a new development project.

Addition of features

Software is an ever changing endeavor. What was a perfect solution to a problem three or four years ago no longer performs as needed. Consider the need to add a diagnostic subsystem to an existing software system. Two options exist for adding this subsystem: write new software or add these features to the existing software. Writing new software has obvious disadvantages. Adding the features to the existing software poses some new problems. Any data needed by the diagnostics must either be already incorporated into the existing software or modifications must be made to the original software to access the additional data. However, any modifications necessary to access additional data should be much simpler than designing a new software system just to implement the diagnostic subsystem. Again, a company is going to chose an option based on the overall cost of the options.

Modernization

First, we must answer the question, "What is modernization of software?" In the last few years, some areas of software engineering have improved greatly. We consider adding a graphical user interface (GUI) to a piece of software as modernizing the software. The advantages of a GUI have been apparent since the introduction of the MacIntosh. The major advantages a GUI provides are increased user efficiency and lower learning curve for the software [21]. Other modernizations of legacy software include new control methods (fuzzy logic, neural networks) and improved diagnostic methods. The advantages of modernizing legacy code are many. Some of these advantages are: (1)increased knowledge extractable from the system; (2)improving the effectiveness of the system user; (3)improving system and user efficiency; and (4)enabling new users to become sufficient in the operation of the system in a much shorter period of time.

Our experience in working with legacy codes has indicated that these are the four biggest advantages of modifying legacy systems. Industry has a great deal of money and time invested in legacy software systems and they would prefer to modify the systems rather than discard all of the knowledge gained from these systems. Just the continued use of these systems proves their usefulness and value to industry [4]. If the systems were not performing a valuable operation, they would have already been replaced by a better system. Modifying legacy codes is obviously a very attractive proposition for companies who have a large previous investment in legacy codes. However, modifying these legacy codes is not a trivial matter [23].

Methods for modifying legacy codes

During this effort, several different methods for modifying legacy codes were identified and examined. As with any significant software development undertaking, all of

the issues were dealt with on a system wide basis. After examining all of the identified methods for modifying legacy software, three different methods were considered. We feel these three paths provide a sufficiently broad base of reasonable methods for revitalizing legacy codes. These paths are:

Porting the software;

Reengineering the software;

Providing a "wrapper" around the software.

All three methods require some allocation of resources in order to complete the modifications. Although this is a common problem with any software system, some companies cannot afford spending significant resources to modify an already acceptable system. However, making these modifications can greatly increase productivity. Compared to the resources needed to completely design a new system from scratch, all of these methods are cheap. One must note while these methods can solve many problems, there are certain situations where the best method for modifying a legacy system is a complete redesign. Only knowledgeable engineers can make the choice as to which path to take for a given project.

Porting the software

Porting software is the process of transferring a software package to another platform: either a new compiler, new operating system, or a new hardware system. This is not a new process in the software engineering community. However, porting a software system is usually only done to move the software to a new, updated computer system. Porting can solve some problems with legacy code, but many problems may be unresolved.

The main problems that can be solved by porting the software are performance related. If the system has become too slow to meet it's requirements, porting to another, updated computer system can resolve these problems. Some usability issues can be resolved, but only if the operating system of the new system is more user friendly than the previous system. This is a problem with the operating system of the computer system and not the legacy software.

Among the problems not solved are: improved reconfiguration; addition of features to the system; and modernization of the code. While the method used to reconfigure the system may be modified during the process of porting the software, solving this problem is another effort and not part of the porting process. Another issue not solved by simply porting the software system is the addition of features to the system. In the porting process, the original code is only modified if their is a problem on the new computer system with the original code. Thus, only changes necessary to keep the legacy code functioning properly are made to the software. Lastly, no modernization of the software occurs in the porting process. Even if the new operating system is GUI based, the original code will still run with the old interface; albeit, the interface will run in a window.

The key factors of porting legacy codes include the time necessary to port the code, the cost of porting the code, and consistency of the software. By consistency, I refer to the fact the new software must be tested alongside the old software to ensure no problems were introduced into the system during the porting process. There is also the possibility that the new software will have to be ported to another computer system sometime in the future. It is easy to see that a vicious, expensive cycle could develop here. Lastly, porting a legacy code does not solve many of the problems discussed earlier.

Reengineering the software

Reengineering software involves going back to the original design documents and rewriting the software to encompass any new features, new methods, or new modernizations that need to be added to the system. This is not re–implementing the legacy system since new features are added and legacy system problems are corrected. Redeveloping a working software system (legacy system) is not a normal endeavor for most companies. While all of the problems of the legacy system can be corrected in the new system, there are some problems that can either be unresolved or created by the redevelopment operation [5].

This process is commonly known as **reverse engineering** software. Working from the actual application, an engineer works to determine the actual methods used in the application. Commercial tools are available to aid in this process. One common reverse engineering tool used is *Software Refinery* [19]. No single tool currently available is sufficient for reverse engineering software in general. Each system to be reverse engineered must be examined individual to determine the best method for reverse engineering.

Problems solved by the redevelopment process include easier reconfiguration of the software, adding new features to the software, and modernizing the legacy system. The problem of easy reconfiguration of the system is not guaranteed to be solved by the redevelopment process, but if careful consideration is given to the ease of configuration of the system improving the legacy system can be accomplished. New features can be brought into the design process of the software, and therefore, can be correctly implemented in the software without hacking the system. A clean design is one of the key advantages of redeveloping a legacy system. Modernizing the software can easily be accomplished. As in adding features, modernizing the code needs to be considered in the design stage of the development process.

Some of the problems of reengineering a legacy system include the increased cost and time of the development process. In the other methods for modifying legacy systems the design process is much shorter than in the redevelopment processes. A shorter design process will require less time by the software engineers; this translates directly into less money. If the original documents from the legacy system's design still exist they can be used to significantly reduce the time required to redevelop a software system. By using the information contained in the design documents the group responsible for redeveloping the software can eliminate part of the design process. In the case of missing legacy system development documents, the group must either re–enact the entire design process or examine the source code to determine the existing system design.

Several key issues involved with redeveloping a legacy code include the cost of the system and the new additions that will be added to the system. If the additional value of the new features and the updating of the old system outweigh the cost, in time and dollars, of redeveloping a system, redevelopment is a reasonable option. However, it can be cheaper to implement a new design. Another area that must be watched is the consistency of the new system. There is no guarantee that the new system will perform as well as, let alone better than, the legacy system. Redevelopment, without proper system testing and verification, is a risky endeavor.

"Wrapping" the software

The last option we considered is providing a **wrapper** around the legacy software. Wrapping existing software is somewhere between porting and redevelopment as far as risk and cost are concerned. Writing a wrapper around a legacy code involves using the core routines of the legacy system as. These routines are interfaced to new input and output routines. See Figure 2 for a representation of wrapping legacy code. Also,

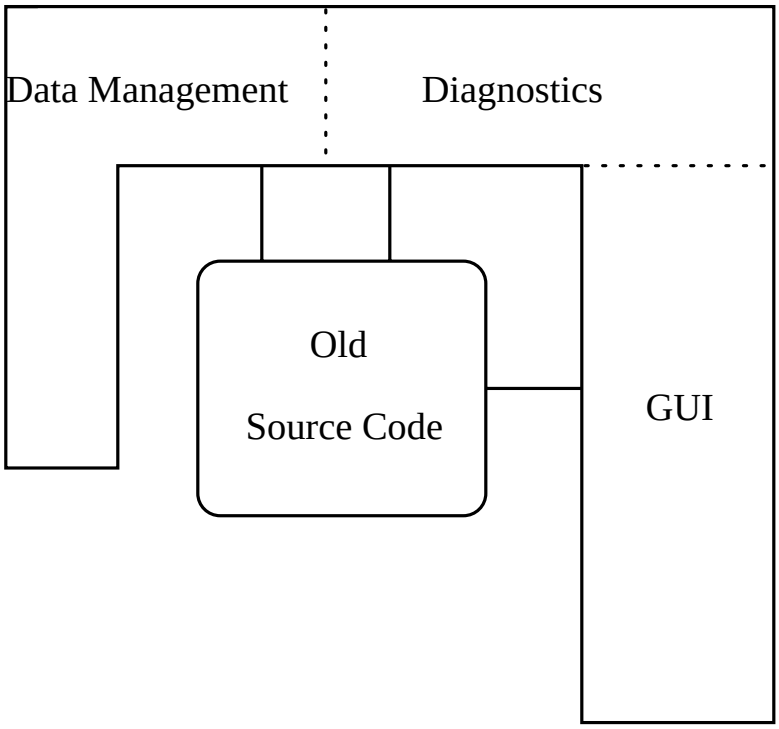

Figure 2. "Wrapping" Existing Software

interfacing to the internal data structures of the legacy code can be beneficial as it can allow easy incorporation of new features. By reusing the core routines of the original system, testing to ensure the new and old systems are consistent can be minimized. If the new system is on a new operating system, the core routines of the legacy system must be ported to the new operating system. This adds some complexity to the issue of adding a wrapper around the legacy system core routines [4].

Several requirements must be met by the existing code for wrapping to be a viable option. First, if a new operating or computer system is needed, the old source code must be able to run on the new system with a minimum of modifications. Also, a good "black–box" model of the existing source code must be available. The input and output routines of the legacy code must be well defined. This can be defined in either a design document or the source code itself. Without the input/output specifications of the legacy system, the interface points between the wrapping software and the legacy

software are not well defined. If either condition is not met by the legacy software, providing a wrapper for the source code is not a good option.

Once again, the time and money required to provide a wrapper for a legacy system are greater than simple porting of the code. However, the ease of adding features is on the order of adding the new features in the redevelopment of a new system, as long as the interfaces to the old core routines were implemented well. Interfacing to the legacy software is a vital issue. If not done well, the new system may be unusable or not offer any significant improvement over the old system.

Comparison of the proposed methods

While all of these methods have their advantages and disadvantages, a comparison of the proposed methods is beneficial. Depending on the legacy system and the goals of the modification project each modification method could be used. The project should define which method is best suited for the present needs. The key areas where a modification method must perform are system modification cost, system performance after modification, and additional usefulness of the system after modification. Each modification method should be examined as to the costs and benefits in each of these areas. Then, the best modification method for providing improvements over a legacy code can be chosen.

CHAPTER III

DIAGNOSTICS

A diagnostic system can be considered a mapping of inputs to outputs. The diagnostic system takes a set of inputs, and if there is an error recognized in the inputs, notifies the system user of the error and of possible error causes. For a given set of inputs, the system must assert the corresponding system outputs. A diagnostic system receives some type of system input. The system then decides if a problem exists. If an error exists, some subset of the system outputs are asserted. This is used to alert the system operator of a possible failure in the system.

A large amount of research has been performed in the area of diagnostics. Diagnostic systems for many different problem domains have been designed and implemented. A data validation system can be considered one type of diagnostic system. It is given a set of inputs and then determines if a problem exists. If one exists, the system must be able to determine the cause of the problem and alert an operator to the problem. This is the task property of a diagnostic system. Several different types of diagnostic systems have been built. Using the knowledge gained from past research projects can only improve the performance of a diagnostic subsystem. Due to the similarities of data validation and a general diagnostic system, and the need for adding diagnostics to the new data validation system, several different diagnostic methods will be examined.

Over the past few years several methods of diagnostics have been identified. Figure 3 shows a diagram of different diagnostic methods (Figure 3 was taken from [32]). Methods higher in the figure are generally considered capable of better, or more complete, diagnostics. A method can be transformed into a better method if

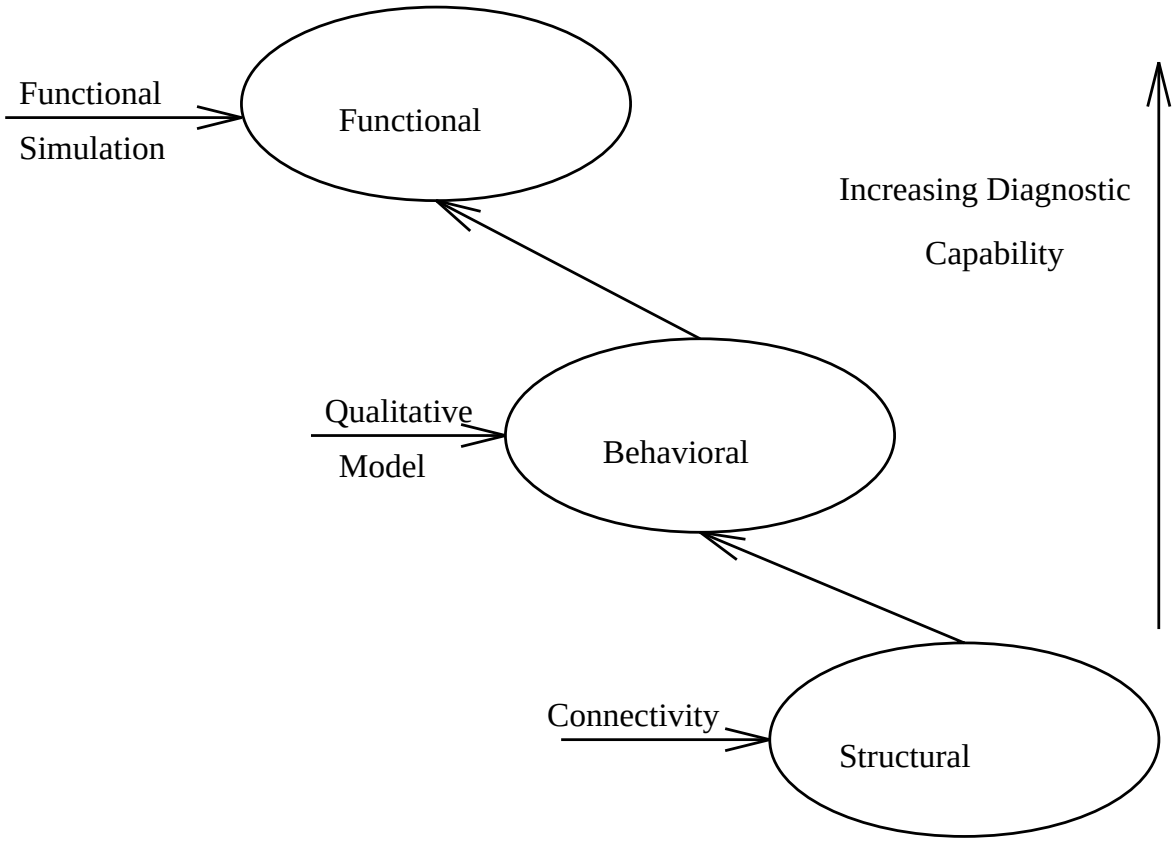

Figure 3. Diagnostic Levels

enough information can be added to the diagnostic information already available. A brief description of each of the different methods will be given. Of course, some of these methods are not currently applicable to the data validation process. Some of these methods require information that is not available, not tractable in size, or not obtainable. Only certain methods are candidates for immediate use, but the available choices will grow as the information about the data validation problem grows.

Levels of Diagnostics

Three basic strategies of information representation for diagnostics exist. One key element linking these strategies is knowledge present in one level can be used to derive the information needed in a higher level diagnostic system. The ability to manipulate

the available knowledge into what is needed for a particular diagnostic system enables us to choose the best form of diagnostics for a given system. Otherwise, the diagnostic system used would be determined by the information currently available and nothing else. By transforming the information readily available into a form usable in a higher-level diagnostic system, system flexibility and expandability are improved. Now, for a look at the major classes of diagnostic representation [31].

Structural

Structural diagnostics is the simplest representation method available. The information needed for structural diagnostics includes the connectivity of the system being examined. For an electrical system, this would be the actual component interconnections. In a software system, this would entail the data dependency of the source code's variables, for example. From this information, the diagnostic system can help isolate the cause of a problem [13] [32].

A forest structure can easily be built from this structural dependency information. By making each component of the system a node in the forest, connecting the nodes according to the physical connections of the system builds an accurate system interconnection representation. Whenever an error occurs, a search on this forest structure will identify all possible paths the error could have followed. It is important to have a correct forest, otherwise the diagnostic system will produce erroneous results.

This information can be used by an engineer to quickly diagnose the root cause of an error. Giving the system operator a reduced set of possible error causes allows the operator to concentrate on the status of the components that map to these causes. Checking fewer components for errors will obviously help reduce the time needed to find and correct the true cause of an error.

Behavioral

The next method of diagnosis uses a **behavioral** model of the system. By basing the diagnosis of any errors on system behavior and not just system interconnectivity, a more concise diagnosis solution can be achieved. This is realized as a smaller set of possible paths the error could have taken. By qualitatively modeling the system, error causes identified by a structural model can be eliminated if they do not actually affect the component of the system that has experienced an error [33] [32].

Now, the question of how to transform a structural system representation into a behavioral representation must be answered. This allows a smaller set of error paths to be generated than with only structural diagnosis. Behavioral diagnostics is best used to augment the results of a structural diagnosis. Only by studying the behavior of the system can a behavioral model be determined. This can only be accomplished by either qualitative simulation or consolidation. Using this information a mapping between system behavior and system structure can be created.

Once again, a positive error cause can not be determined from a behavioral diagnostic package alone. As in structural diagnosis, a reduced set of possible errors is given to the system operator that allows for less human diagnosis effort. Using behavioral diagnosis will improve on the results of a structural diagnosis by giving a potentially reduced set of possible error causes.

Functional

Moving to a higher level of diagnosis requires teleological reasoning. Teleological reasoning involves viewing the system not only from a structural or behavioral view, but also from the aspect that views the **functionality** of the system. The diagnosis system is aware of the intended functionality of the system being examined. With this additional information, an even smaller set of possible error paths is delivered

from the diagnostic system. Transforming the behavioral model of a system into the functional model allows the diagnosis system to also examine the relationship between system errors and system functionality [32] [30].

As in the structural and behavioral diagnostic systems, an exact cause of an error can not be determined. Instead, an even further reduced set of possible causes can be identified to the system operator. In some cases the set of error causes can be limited to only a small percentage of the original possibilities through functional diagnosis, thereby improving upon the efficiency of the diagnosis process.

Improving Diagnostics

Choosing a diagnostic system is easiest if the choice is based on the amount and type of information available. Methods for moving from one diagnostic system to another have been mentioned in this thesis. While the actual methods are better left to the experts in this field, using the algorithms they develop allows many different choices for a diagnostic system. Information necessary for any method can be used in developing a diagnostic system of another, possibly better, methodology. Over a period of time a diagnosis system can be developed for a problem which performs with great accuracy.

Using a simple diagnostic system can facilitate the gathering of more detailed error–and–cause information. By using a modular approach to building a diagnostic system, a simple diagnostic method can be used at first. Information gathered using this simple method can be used in developing a more advanced system. When the more advanced system is ready, it can replace the simple system. This process can iterate until a very precise diagnostic system is available. In the meantime, the system under scrutiny does not have to operate without a method for aiding in diagnosing error causes.

Diagnostic System for Data Validation

For the steady–state data validation problem, a set of **errors** are identified by the validation system. Based on these errors, a set of possible causes needs to be determined. System users can make more informed decisions if they know the actual cause of an error. Using a diagnostic system can generate a set of possible error causes for examination by the system engineer.

Currently, only structural diagnostics is a viable choice for the data validation system. Only the source code for the original system is available; no documents used to design this system exist. Due to this, the only system information available is the structural information included in the source code. Extracting the data–dependency graph from the source code is analogous to building a structural model of the system. While work is underway on developing a model-based diagnostic routine for data validation, it is not ready for production use at this time. Several other diagnostic systems are better overall choices due to their improved performance. However, using structural diagnostics is a vast improvement over the original data validation system. Using these diagnostic routines can greatly improve the efficiency of the system. One must keep in mind the overall goal of the data validation system when making a major design decision. Using one diagnostic routine without considering the future migration to another could cripple this project. This is why diagnostics was a major consideration in the development of the data validation package. By designing a method to allow an easy change from one diagnostic routine to another, flexibility and the ability for system growth and maturation was ensured.

CHAPTER IV

NEXT GENERATION DATA VALIDATION

A new data validation package has been constructed to overcome the shortcomings of the previous system. The complexity involved in a data validation system is obvious. In order to facilitate managing the system complexity, increase system flexibility, and enable easy system modification a software system known as the **MultiGraph Architecture** (MGA) [2] is used. Using MGA to develop the new data validation system greatly enhanced system development. The following sections of this thesis will describe MGA and the new data validation (DATVAL) system in detail.

MGA

The MultiGraph Architecture was developed at Vanderbilt University over a period of several years. A diagram of the overall system structure is shown in Figure 4. The basic concepts behind the design of MGA involve domain–specific system modeling. By modeling a system in terms familiar to the system user, some of the complexity of the system can be hidden from the user. This approach allows the system user to modify and rebuild complex software systems with relative ease.

Typical software engineering practices are not practical for applications which require frequent, sometimes major, modifications. The latency between modifications to a conceptual state and an implementation of a new software system encompassing these changes must be small. The DATVAL system requires modifications made during a test to be available in just a few test points. This turn-around time is often less than ten minutes. Using MGA facilitates meeting these requirements.

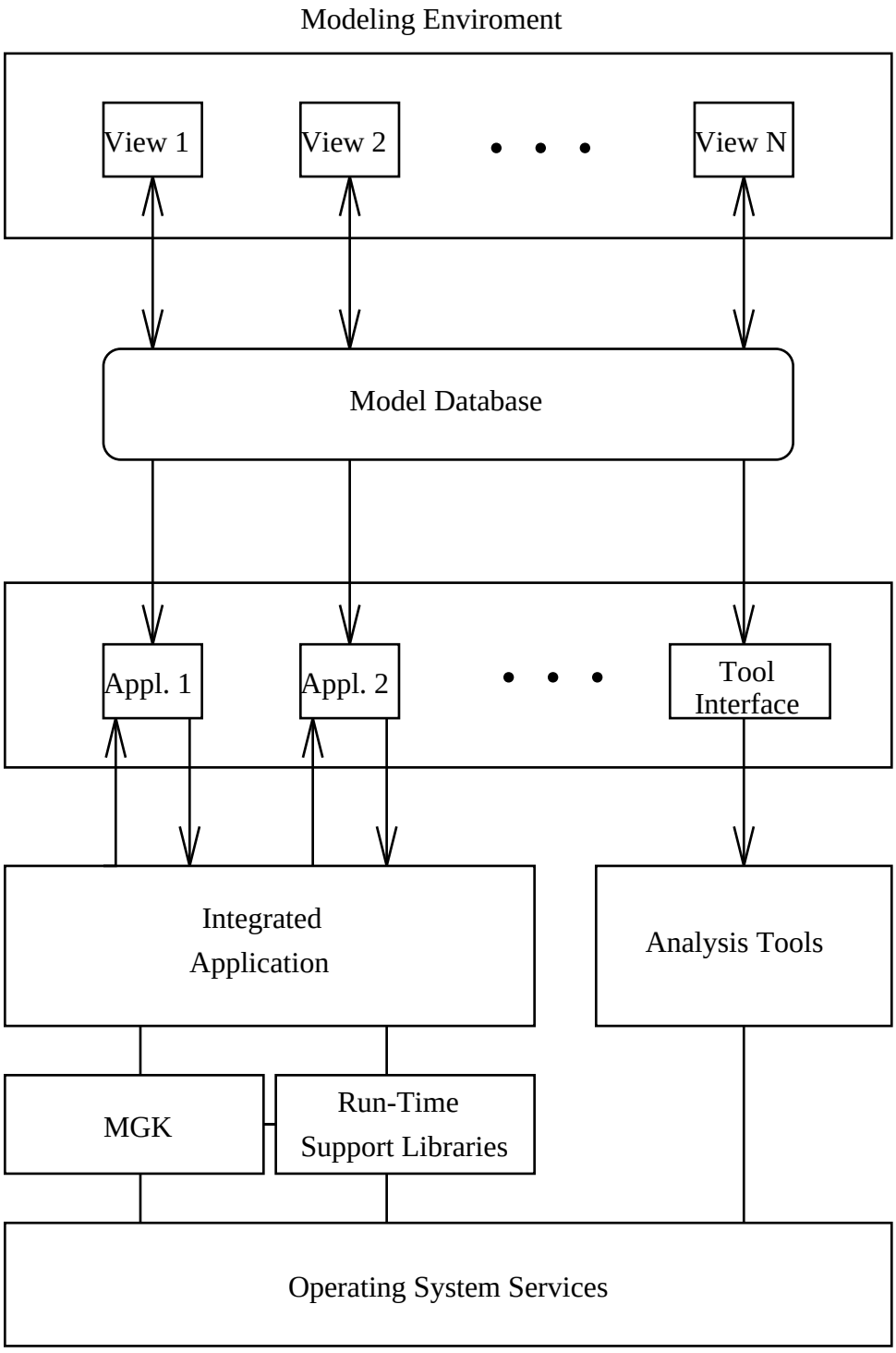

Figure 4. MultiGraph Architecture

Some of the key elements of MGA will now be looked at in more detail. The more important features of MGA include:

Graphical Modeling: Using diagrammatic models allows for easy system changes. The system engineer works with icons from his domain of expertise. This allows the engineer to work with familiar items, in a familiar way. This is one method of abstraction that removes the system user from the computer engineering aspects of a system [3].

Multiple Aspect Modeling: One major concern when dealing with graphical system modeling is complexity management. In MGA, multiple–aspects of the same model can be viewed. This allows the user to only deal with those features of the model he is concerned with. Hierarchy is supported in the MGA modeling environment as another means of managing complex model structures.

Domain Specific Model Interpreters: Domain specific model interpreters take the models produced by the graphical model builder and convert the models to *MultiGraph Kernel* (MGK) commands. These interpreters must be written for each modeling domain and are usually written by a software engineer who is also familiar with the problem domain to be modeled. After development, system models are interpreted into scripts executable on the MGK [15].

Domain Independent Analysis Tools: Analysis tools are often very general in nature. Since all model interpreters produce MGK scripts, analysis tools designed for MGK specific scripts can be used in many diverse problem domains. By building analysis tools designed to analyze MGK scripts, the same analysis tool can be used for many different domains. Using domain independent tools allows code to be reused in many different disciplines.

Domain Independent Kernel: The MGK is domain independent. Many diverse domains are modeled with MGA and all produce scripts that run on the MGK. A domain independent kernel allows for all modeled domains to use the same basic execution environment. This has benefits in debugging, code reuse, and information exchange between applications.

Modular Kernel: The MGK has been re–implemented as a micro–kernel. See Figure 5 for a diagram of the MGA when utilizing the new MGK. This new approach allows the kernel to support many high–level features and still retain the ability to remain small in size. Only those feature sets needed by the current domain are actually loaded into the kernel used by their produced applications [1].

The overall system interaction enables the multi-aspect, domain specific modeling approach to be applied in many diverse areas. Figure 6 shows the different domains MGA has been successfully applied in. All of the domains have some features in common, but are vastly different. MGA has been used to produce: CADDMAS (Computer Assisted Dynamic Data Measurement and Analysis System), a high-speed turbine engine diagnostic and monitoring system [6]; IPCS (Intelligent Process Control System), a chemical engineering control and monitoring system [11]; MIRTIS (Model Integrated Real–Time Imaging System), a real-time image processing system [8]; DTOOL, a fault diagnosis and isolation tool [10]; and a discrete manufacturing monitoring, diagnostic, and control system.

Based on the previous success of MGA, we chose MGA as the methodology to be used in developing DATVAL. The overall flexibility of MGA had a great deal to do with this decision. In fact, the prototype demonstration system for DATVAL was produced using IPCS with some minor modifications. This allowed a quick prototype

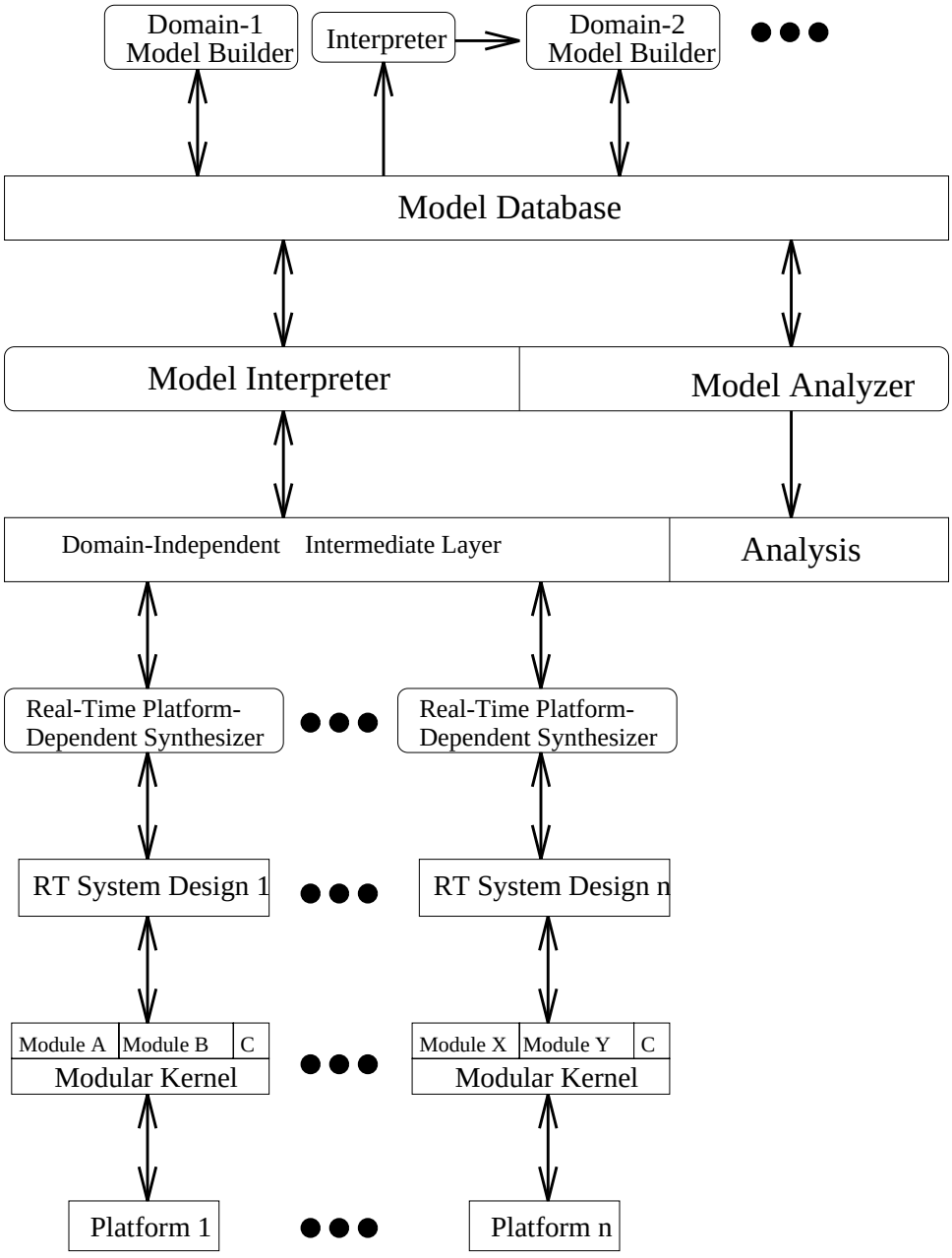

Figure 5. The New MultiGraph Micro–kernel

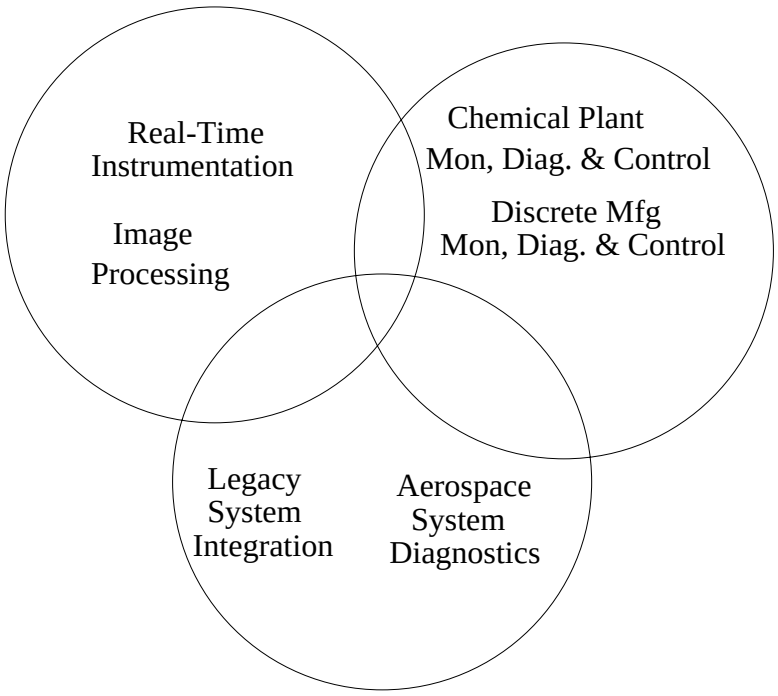

Figure 6. MGA Domains

system to be built. After a successful session of demonstrations, a production–quality DATVAL was approved. MGA was the only logical course of action at this point; after all, it had been used to produce a successful prototype system.

Prototype Data Validation System

As a first step in developing the data validation system, a prototype was designed and implemented. In order to facilitate the quick generation of the prototype, *Intelligent Process Control System*, a MGA tool intented for use in the chemical industry, was used to construct the data validation system prototype. This prototype was demonstrated at AEDC in actual turbine engine tests. After the successful demonstrations, work began on a production quality data validation system.

Figure 7 shows one of the models used to generate the data validation run–time system. During the demonstration phase of the system, the engineers at AEDC decided graphical modeling was not necessary, and not wanted, for this problem. Therefore,

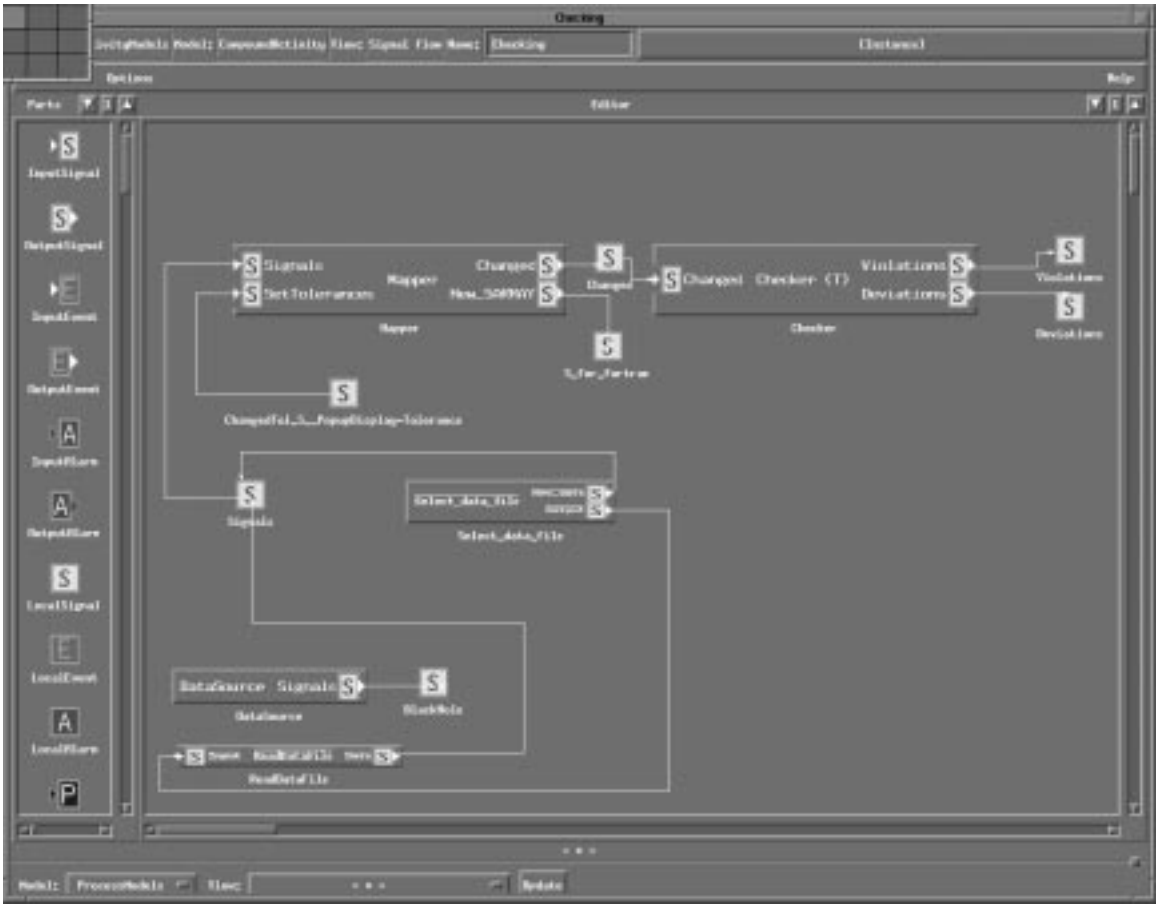

Figure 7. Prototype Modeling Environment

in the production system a new modeling method was devised. The production run–time system re–used many features of the prototype system and the model editor built upon features of the prototype modeling environment.

The next sections will discuss the production data validation system in detail. Since the same problem was solved with both generations of the new data validation system, the prototype system will not be discussed further. It should be noted, while James Davis designed and implemented the prototype system, the current data validation system, with the exception of the data dependency analysis work, was developed by Dr. Gabor Karsai.

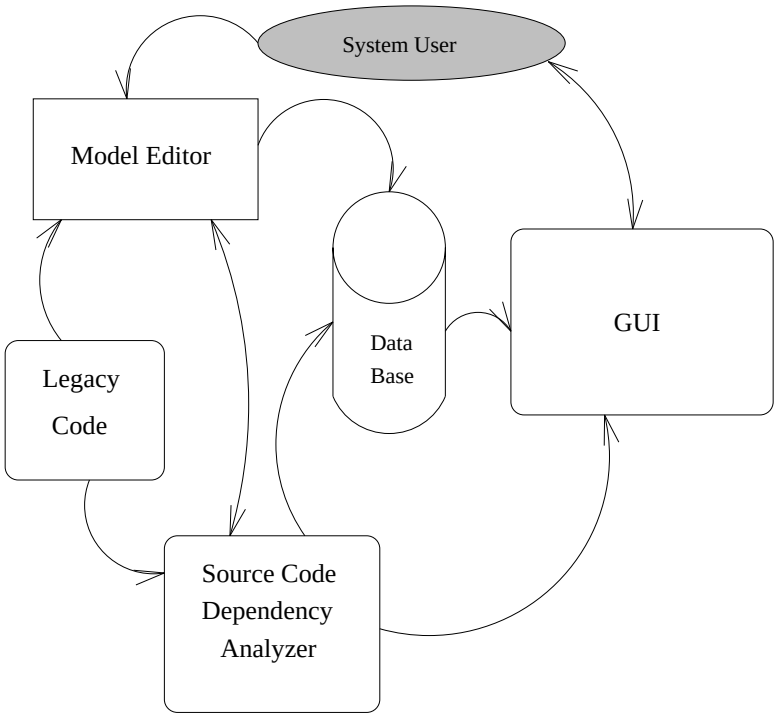

Figure 8. Data Validation Development System Interaction

Production Data Validation System

A block diagram of the production data validation modeling system is shown in figure 8. The system user has two interface points: the model editor (DatEdit) and the GUI (DatVal). From the model editor, the legacy code routines are incorporated in the system. The user can then make configuration modifications; these modifications will be described later in this chapter. The model editor then stores the system models in the model database. This database is shared by the model editor, GUI, and source code analyzer. The model editor can interface with the source code dependency analyzer to determine the software structure of the validation system. This information is returned to the model editor and is stored in the database. The GUI has access to this information and uses the dependency analyzer's output for system configuration at run–time. Using these tools, the user then builds the GUI (DatVal) to be used for the actual data validation process.

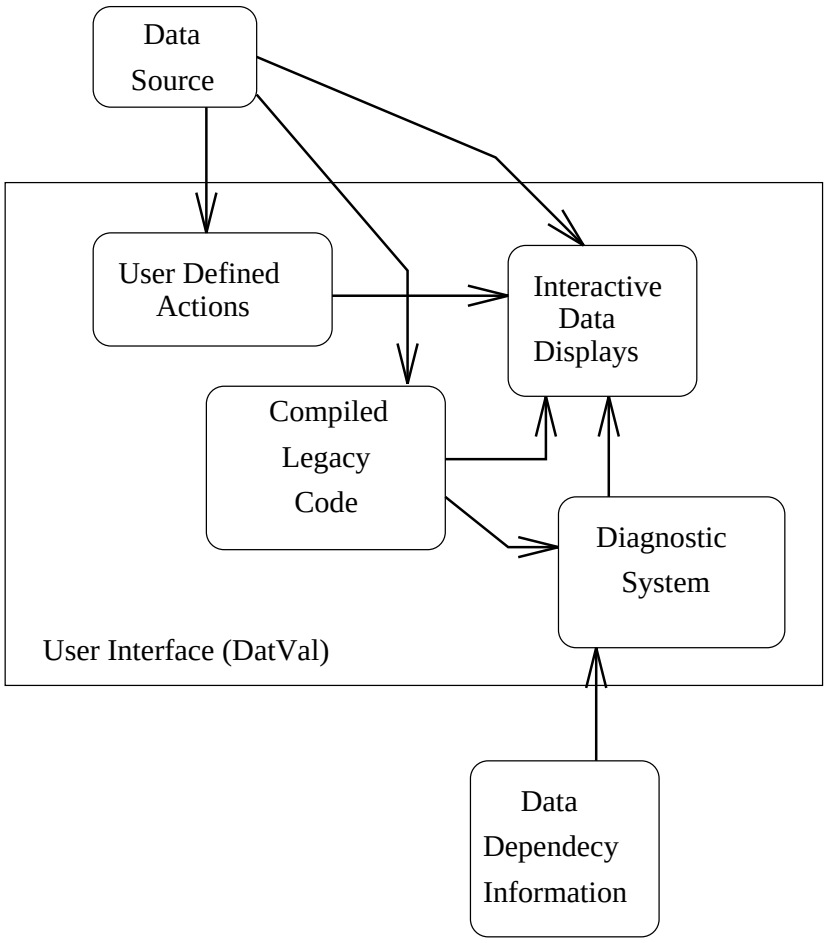

Figure 9. Data Validation Run–Time System

Figure 9 diagrams the interaction of the seperate parts of the run–time system. All aspects of the run–time system are access through the system GUI. Data is transferred into the run–time system where it is analyzed by the legacy code, the user defined code, and the diagnostic system. All of these components are specified with the modeling enviroment. The interactive data displays allow access to all data in the system. The diagnostic system recieves a list of errors identified by the legacy code and a structural description of the system that was generated by the data dependency analyzer. This system then alerts the user to possible error causes.

The interaction of all three elements (model editor, dependency analyzer, and GUI) is important to the structure of the system. The model editor allows the system user

to input information necessary for system operation. Structural information stored in the legacy code is extracted by the dependency analyzer. The GUI allows interactive use of the system and allows access to all system information. The overall system composed of these parts is the data validation system now in use at AEDC.

DatEdit

DatEdit is the data validation system model editor. It is used to model and automatically build **DatVal** (the runtime system). Unlike some of the previous MGA modeling environments, DatEdit is not a graphical editor. One of the key lessons learned from the prototype system was the engineers at AEDC did not want to learn a new method for modifying the data validation package. They especially did not want to leave behind the current method of configuring the system to move to a graphical representation. Since the most often made change to the system was the mapping of input variables to different internal variables, the use of a graphical modeling package was not warranted. However, the new modeling environment is much more complex, although easy to use, and has much more functionality than the previous system. This system can be configured in the same manner as the legacy system. Fortran code will be mentioned throughout this chapter and refers to sections of the original data validation package. Utilizing the existing Fortran code allows for more flexibility in system configuration.

Figure 10 shows the main user interface to DatEdit. From this interface, the user can open, save, and create both projects and tests. A project is used to define different engine test programs at AEDC. AEDC performs many concurrent tests; projects allow the engineers to keep test projects separate. A test is used to define a unique identifier to the current test article and conditions. Tests allow for quick test condition recall. Multiple projects exist to allow many different DatVal configurations

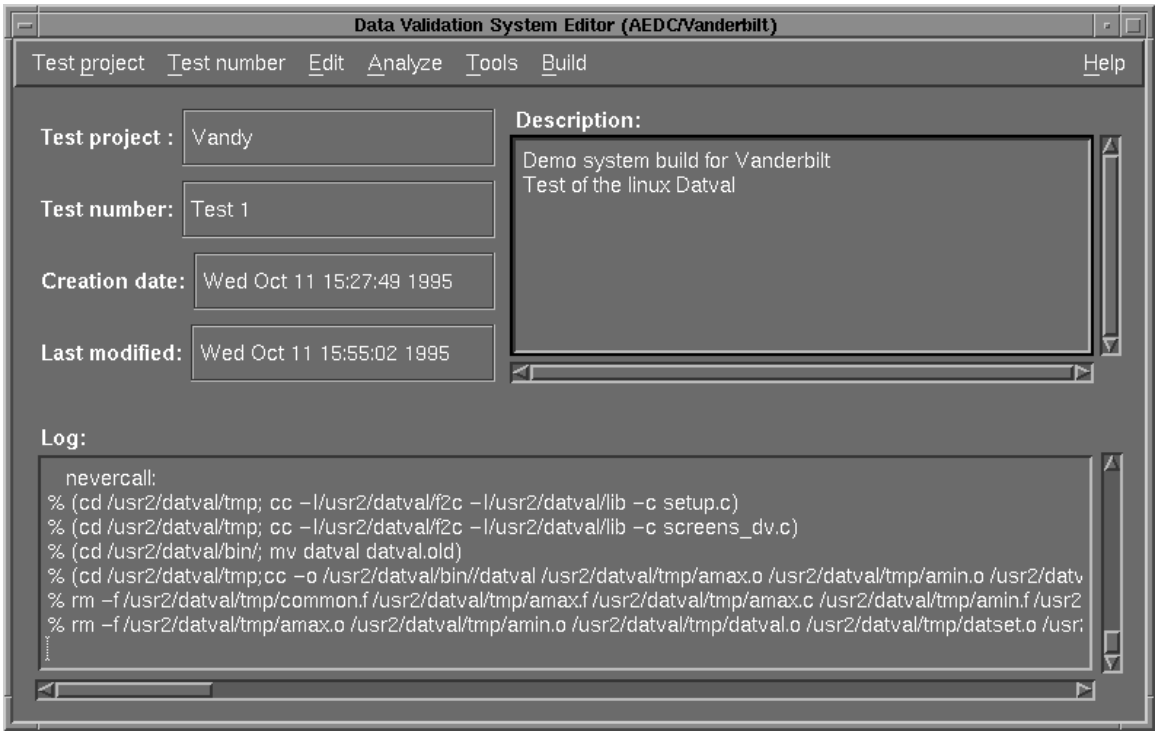

Figure 10. DatEdit Main Interface

to be saved at one time. The need for multiple tests came from allowing each engineer to model a run–time system to meet his specific needs. The user can also move into the edit menu, which allows the user to edit the system to meet the current needs. If the engineer wants to automatically configure the system, he can use the **Analyze Modules** selection to configure the system and the diagnostics subsystem. A set of tools allows the user to easily update nomenclature files (files containing detailed information about the legacy system variables), Fortran subroutines, and images (graphic files used for backgrounds). A tool for starting the image editor is also included in the main user interface. Last, the build option allows the user to build a new DatVal from the current configuration. A feature of this system is the ability to modify the configuration on-line, build a new DatVal, and then start the new DatVal while stopping the previous version. This allows for a seamless transition to a new configuration.

In the editor sub-menu there are several selections the user can choose from. These include:

Fortran Modules: This is where the user can insert the standard Fortran code used in the system. These are a set of standard subroutines and an include file listing all of the *common block* variables. This is where the legacy code is incorporated in the system.

Standard Checks: From here, the user can modify the standard checks used in the data validation system. These standard checks are usually defined in the Fortran modules, but can be modified here, leaving the original source code unchanged. The standard checks originate in the legacy code. They remain unchanged for most projects and tests.

Temporary Checks: A new feature of DatVal is the inclusion of **temporary** checks. These checks are included by the user to perform simple to extremely complicated computations. These checks are not part of the standard checks.

Item Number/Parameter Names: This allows the user to modify the standard nomenclature files used to map array indices to variable names. This is a reference only. It is not used in the actual source code, but is used when displaying data.

Subroutines: Also a new feature of Datval, the user can add subroutines (Fortran) that can be called from either temporary checks, Fortran modules, or post/pre filters.

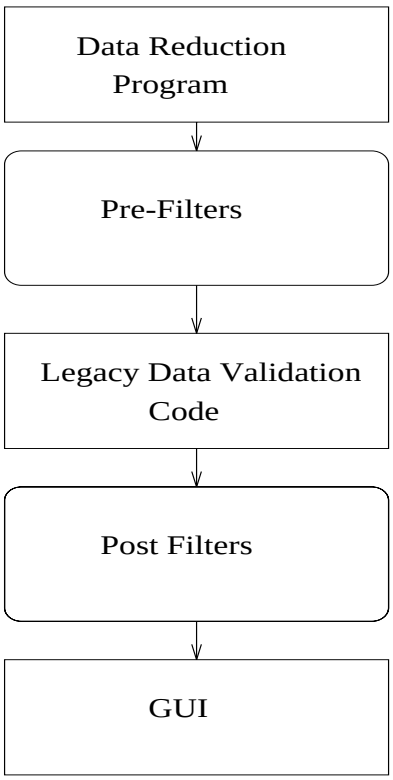

Figure 11. Filters Flowchart

Filters: Filters are segments of Fortran source code that are executed either before (pre–filter) or after (post–filter) the Fortran modules are executed. Filters can be used for further data manipulation than was previously available (see figure 11).

Critical Parameters: A user can input a list of critical parameters that are used in DatVal error displays. A critical parameter is any variable used in the system designated as critical by the system user. If a check uses a critical parameter, the display for that check is highlighted.

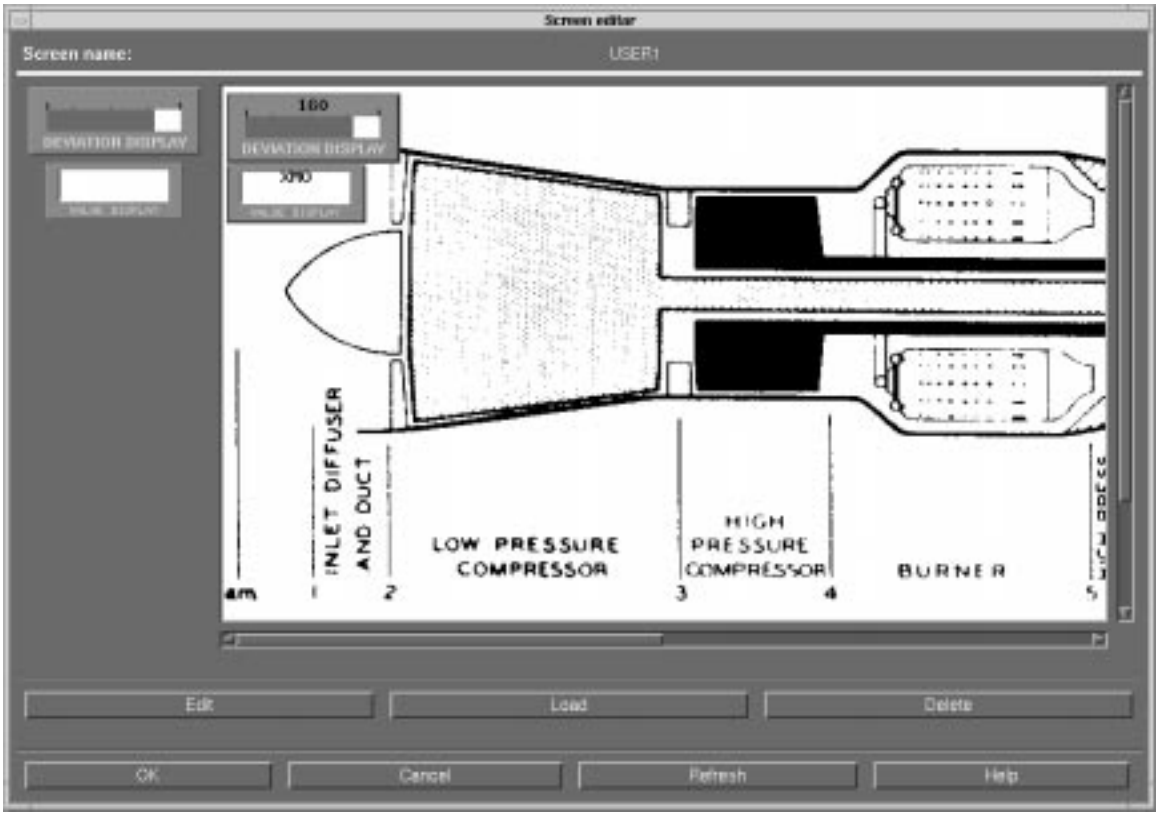

Figure 12. DatEdit User Screen Editor

User Screen: Here, a user can specify a new screen layout that meets his needs. No longer are engineers restricted to common screen layouts that do not meet their specific needs. A sample user screen layout session is shown in Figure 12. This can be used to show the relation between some data item and a physical system component. The background images used in the user screens are often CAD drawings of the test article.

Dependency Graph: The data dependency graph for the current configuration can be viewed. See Figure 13 for an example dependency graph. The data dependencies present in the system are shown by the dependency graph. This graph is a representation of the structure of the system. By clicking on an

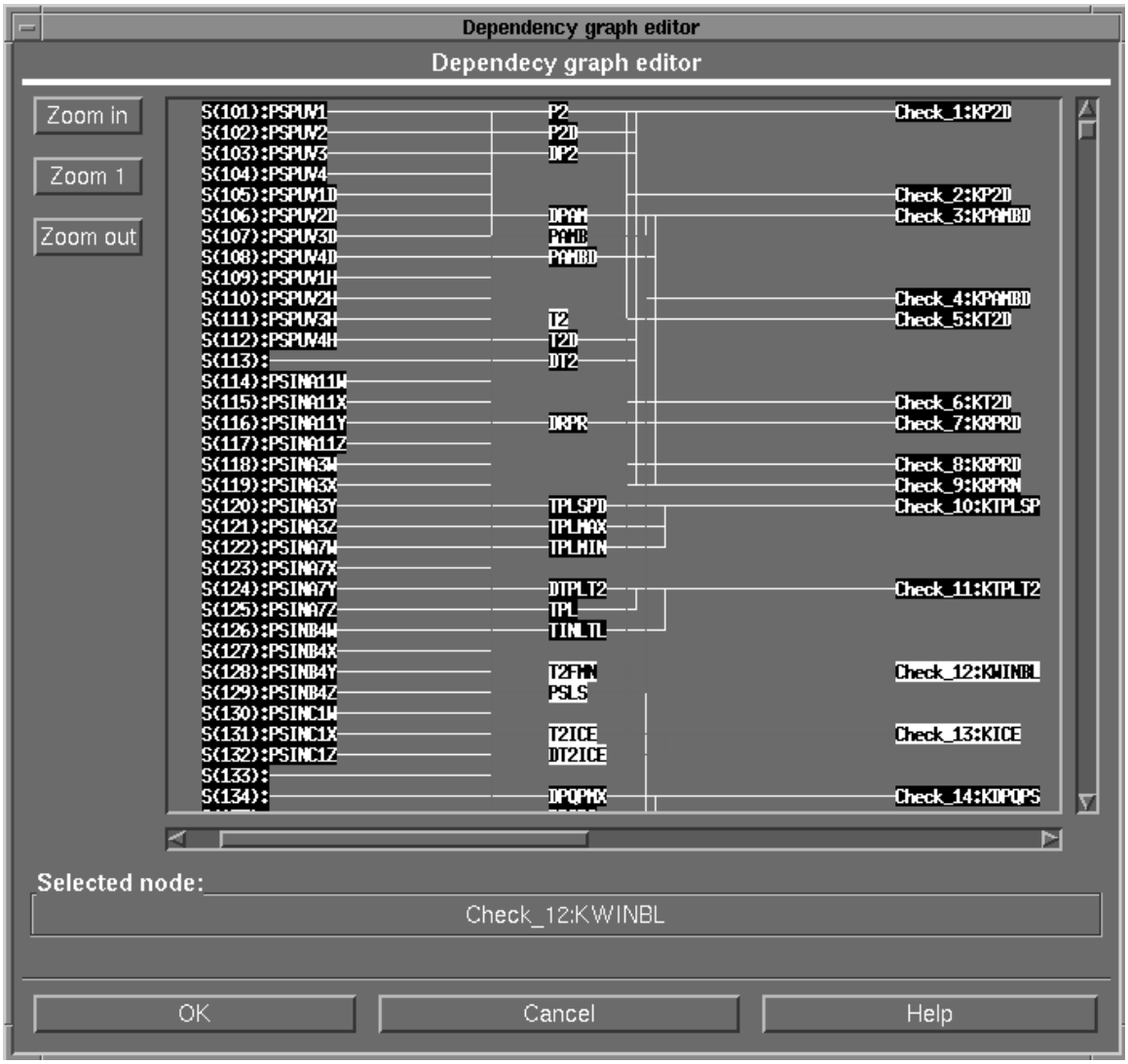

Figure 13. DatEdit Dependency Graph

input variable or a check (output) variable, the display highlights all variables connected to the selected one. This is useful as a graphical display of system connectivity.

Fortran code is used throughout the system. This was due to a requirement from AEDC that the system configuration was to continue to be performed using Fortran source code. We decided to use Fortran throughout the system, but to give users another method for performing some system configuration. In order to interface the

pre–existing Fortran source code with our GUI code (C and C++) we use AT&T's F2C Fortran to C converter. This utility converts Fortran source code to C source code, allowing easy integration of the Fortran into the new data validation system [22].

However, this requirement also created the need for an automatic system configuration tool. This tool needs to configure the basic data validation system and the diagnostic subsystem. This tool will be described in Chapter V. This tool has been demonstrated to work to our satisfaction and shows great promise in fulfilling all of the automatic configuration needs.

DatVal

DatVal is the runtime system of the data validation system. It is produced automatically from the models built using DatEdit. Most test engineers at AEDC only **need** to come in contact with DatVal. Since DatVal is used in the actual test cell, it is the most important piece of the system to the users at AEDC. Without a properly working DatVal they must resort to the old, out–dated method of validating data. Based on feedback from the test engineers, returning to the old data validation system would cause great difficulty. This is mainly due to the many new, performance–enhancing features included in DatVal.

The new DatVal had to meet several requirements handed down from AEDC. For the runtime system, the boundary conditions required by AEDC were:

Workstation Environment: This system must run on a workstation under Xwindows. Specifically, the system was implemented on an Silicon Graphics Incorporated Indy 2 workstation.

Interface must access Fortran data: Since some code was written in Fortran, the user interface must be able to access data used in the Fortran code. This was solved in DatEdit. Please see the section on DatEdit for further details.

Diagnostics: A diagnostic system must be implemented in the current system. Some rudimentary diagnostics were implemented. The description of the diagnostics system follows in this chapter.

No need to maintain foreign code: AEDC desired a system that would allow them to keep evolving the data validation process without resorting to rewriting foreign (i.e. non–AEDC written) code. While some code will not need to be re-written, some aspects, such as modifying the system for a new data acquisition system, will require modifications of the system source code.

Basic DatVal Features

DatVal overcomes many of the shortcomings of the original system. To begin, a graphical user interface (GUI) is used instead of a line printer for system output. Figure 14 shows the main user interface. This interface was designed to be small in size so other programs can be run on the same workstation if the need arises.

Several key pieces of information are shown on the main user interface. Some of this information includes: total number of errors reported for the current data point; the current data point and the time the current data point was recorded; the current tolerance set in use; and any error messages printed to the main user interface. From this interface, the user can select from several different options. The user can display errors by either deviation value or error message. Other options include viewing any specific check or any user–defined screen. These options are selected from the applicable pull–down boxes. For the steady–state data validation system, data is

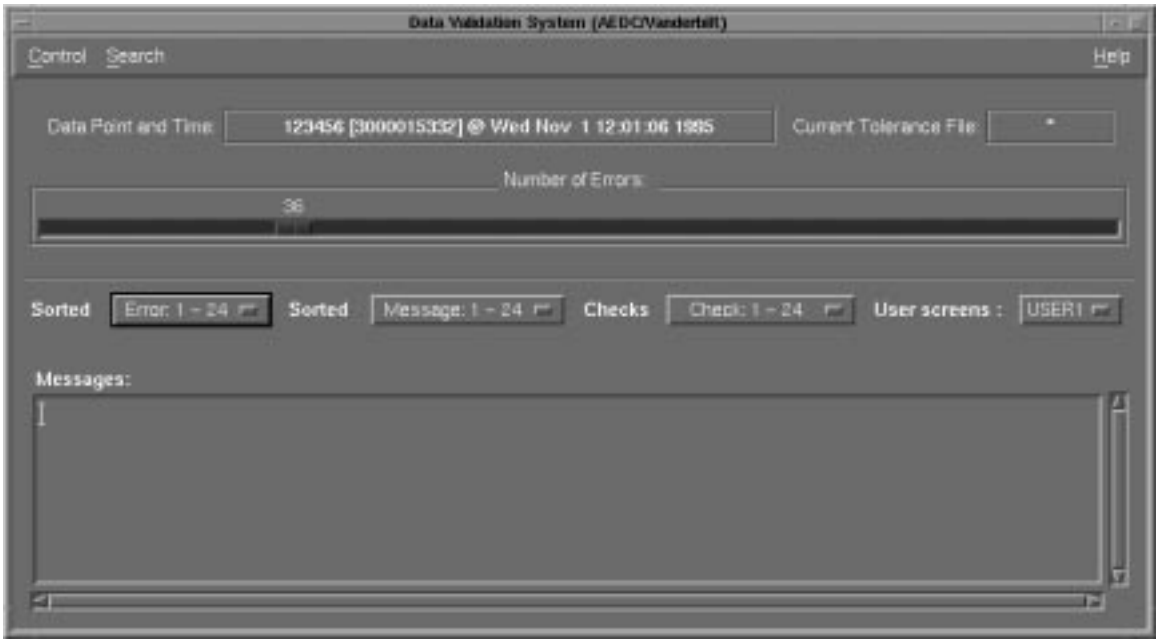

Figure 14. DatVal Main User Interface

received in discrete quantities. These quantities are referred to as **data points**. From this interface the user can activate one of many screens. These include: errors sorted by a calculated severity; messages produced by these errors (also sorted); checks (with and without errors) in a predefined order; and any user defined screens. This simple interface also allows the user to search for a given parameter (variable used for calculation of errors) or a given sensor number (an array index that correlates directly to an input sensor).

While the main user interface relays a great deal of information to the user, in-depth study of problems that arise requires the use of some other features of DatVal. Figure 15 shows one of the most important features of DatVal: the ability to view errors sorted by severity. Each check produces a *deviation* value. This deviation is a calculation of how close a given check is to its tolerance. A check is defined in the system as a possible error condition. The components of this error condition are evaluated to determine if the error has occured. A tolerance is a quantity used to

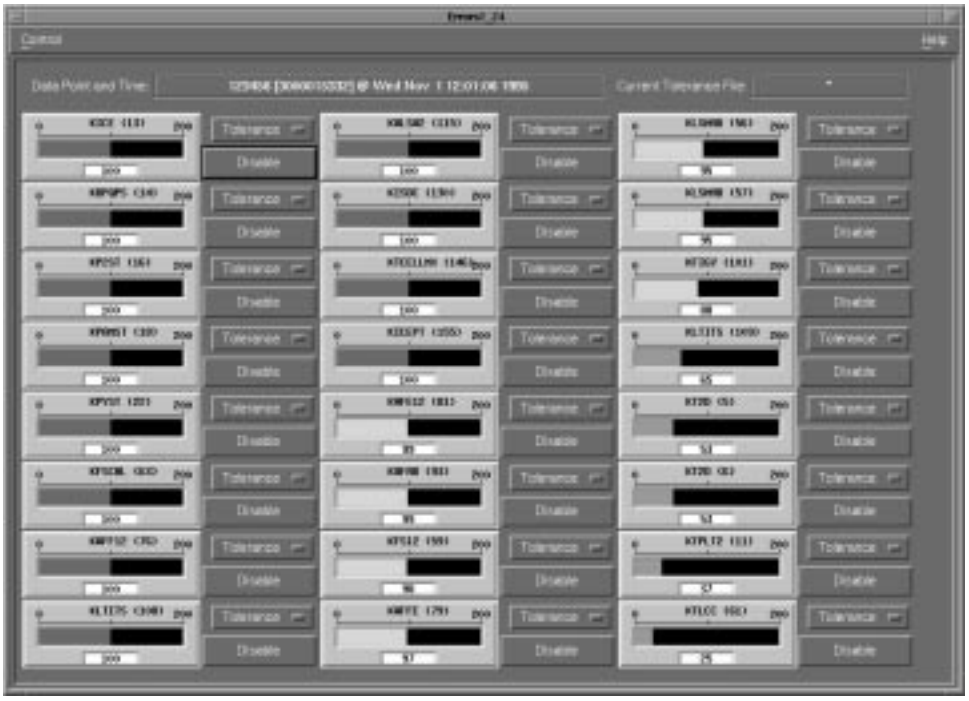

Figure 15. Sorted Errors Display

determine if an error condition exists. By modifying the tolerance associated with a check, the behavior of the check can be changed. The user interface displays this deviation value on a bar-graph display. It can display values between 0% and 200%. In addition to showing the deviation of a check, the deviation display bar changes from green (0% to 75%) to yellow (75% to 100%) to red (100% or over) depending on the deviation value.

In addition, the **sorted errors** display shows the user the current data point and recording time and the current tolerance set. The user also has the option of disabling a check. This does not stop the check's computation, but prohibits the display of the check. A popup window with a list of disabled checks appears whenever a check is disabled. By clicking on a check name in this popup window, a check is re-enabled.

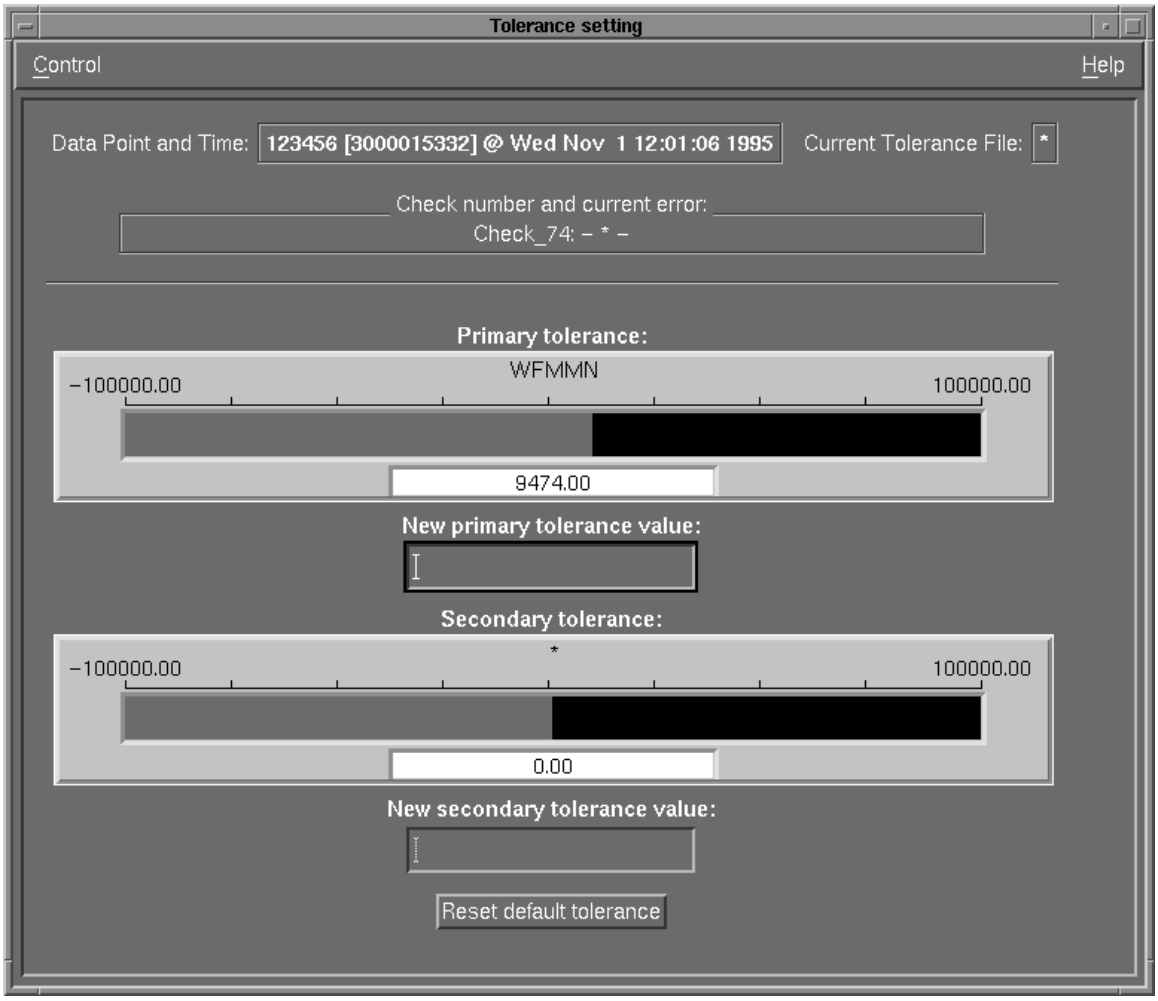

Figure 16. DatVal Tolerance Popup Window

Corresponding to each check is a box that activates a pull-down selection list of possible popup windows a user can request. This box defaults to the **Tolerance** selection. The possible popup windows and their descriptions follows.

Tolerance: This selection displays the current tolerance value(s) for the selected check. Figure 16 shows the popup window used for viewing and modifying tolerances. From this popup, the user can modify the tolerance value(s), reload the default values, or load a specific set of tolerances.

Figure 17. DatVal Parameter Popup Window

Internal Variables: Selecting this button will show the user a window listing all of the variables, and their current values, used in calculating the selected check. Input variables may or may not be listed, depending on system configuration. By selecting a variable (by clicking on its display), the variable is highlighted. Then, the user can select other checks to examine and if the selected variable is used in the newly selected check, it will be highlighted automatically.

Parameters: This popup window is very similar to the Internal Variables popup. However, this window is used to show the input parameters to the check. The selection procedure described above also works for this popup window. See Figure 17 for a sample window.

Info Message: Selecting this button will result in a popup window with the error message the check generates in a text box being displayed. This is the same message used by the old data validation system and is mainly used in order to keep all of the functionality of the previous system.

Diagnostics: By selecting the diagnostics button, the user can invoke the DatVal diagnostic subsystem. This system will be discussed later in this chapter.

The next choice from the main interface is the **sorted messages** window. Figure 18 shows a sample sorted error messages display. By selecting this window, the user again gets errors sorted by severity. But, the errors are displayed as error messages. This is the same message that was used in the old data validation system. Once again, this feature is included to keep available all features of the previous system. No popup windows can be requested from this menu.

By selecting one of the **checks** buttons, the user can view each check in the same manner as the sorted errors. These check displays are shown in groups of twenty–four checks. The look and feel of this windows matches that of the sorted errors window. The difference is this window has the checks sorted in a predefined order. This order is defined in the modeling environment and is input to the run–time system at system initialization.

The last user selection from the DatVal main interface is the **user screens.** These screens are defined by the user with the modeling environment. Figure 19 shows an example user screen. The user can use a background picture to visually relate variable values to physical quantities of the test article. Also, deviation displays, error message displays, and variable value displays can be added to the screen. These displays are shown in the foreground of the user screen. The user has unlimited flexibility in the number and placement of these displays.

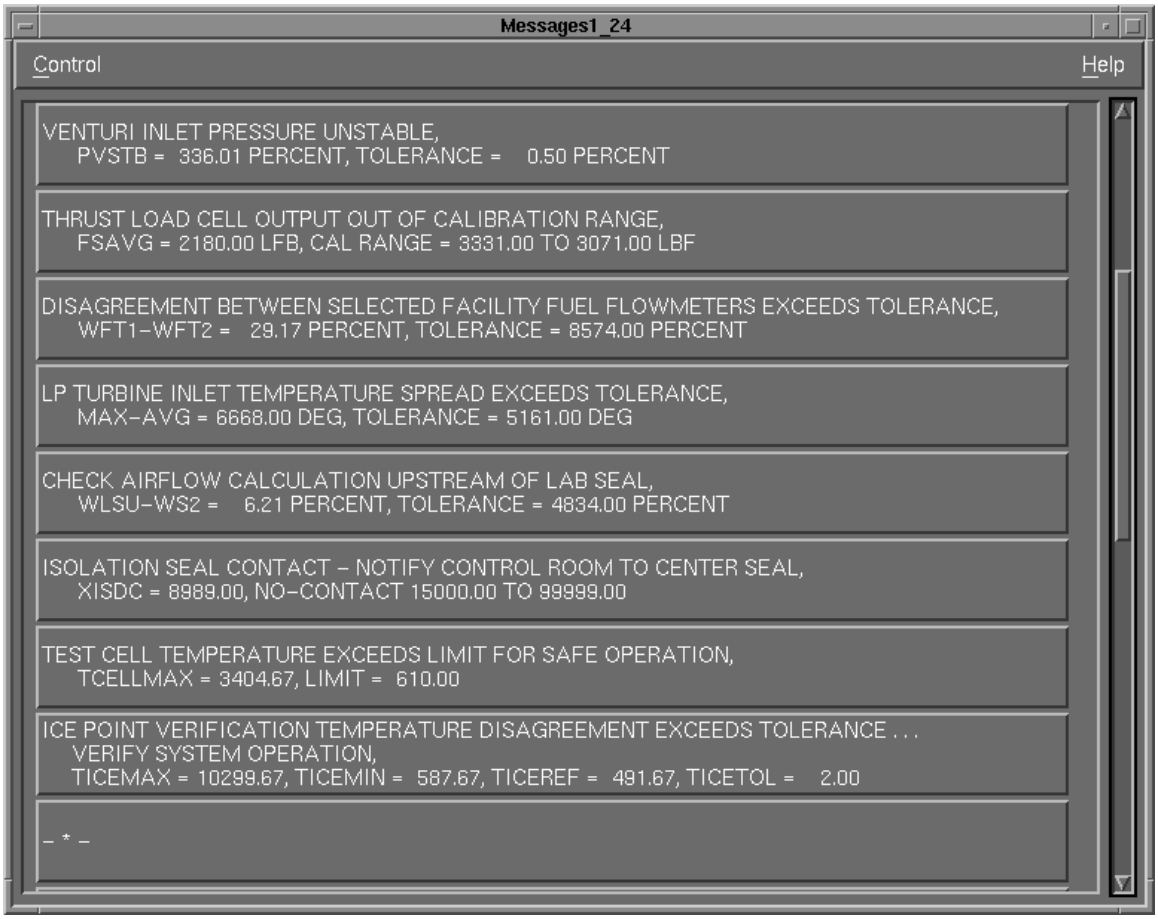

Figure 18. DatVal Sorted Error Messages Window

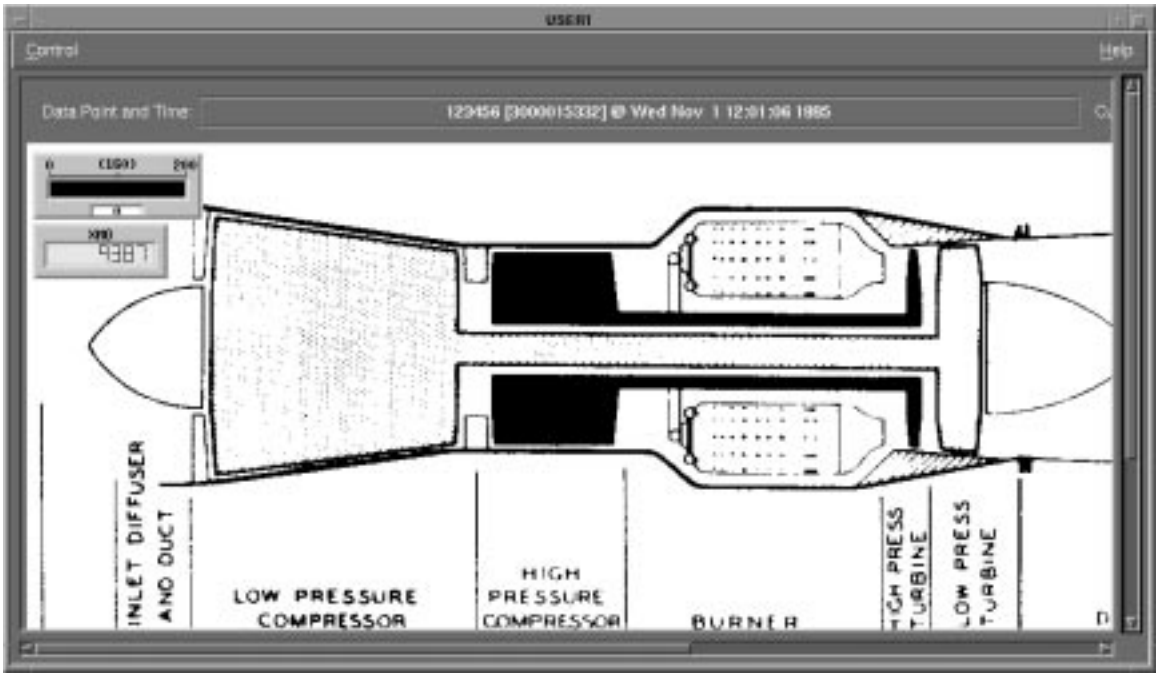

Figure 19. DatVal User Screen

Using this display methodology allows the system user to visualize much more data than with the previous system. Great enhancements to the data validation process were achieved by allowing the test engineer to examine more data and different types of data. Overall, the new data validation system has been well received at AEDC. We feel this is due to solving the majority of problems in the previous data validation system.

Diagnostics

As stated earlier, the diagnostic subsystem used in DatVal can be easily changed to use another diagnostic model at any time. This section is intended to give a brief description of the currently implemented diagnostics system. This system was implemented as a first step in the evolution of the diagnostic system.

Whenever a user selects the **diagnostics** popup window, the diagnostic subsystem is invoked. This diagnostic subsystem is a structural diagnostic system. Based on the

structural (data dependency) information available from the current DatVal configuration, a set of possible error causes is determined. For each error identified by the data validation system, each data item included in the dependency set for that check is identified. Each time a data item is identified as contributing to an error, a counter is incremented. Sorting the data items by counter value, after all errors have been evaluated, gives an ordering of possible error causes. Most error causes can trigger multiple errors. The system user can then start the search for the true error cause based on the ordered set of possible error causes.

The user then receives a popup window listing the system inputs involved in the most errors. By enumerating the number of errors each input is involved in, the user has an idea as to what degree each input effected the current set of errors. While this does not generate the exact cause of a set of errors, it does give the user a reduced set of inputs with which to begin the search for the real error cause. This can greatly reduce the amount of time an engineer must spend searching for the source of system errors.

By giving the system user a reduced set of possible error causes, the diagnostic system can reduce the time necessary to correct an error. While better diagnostic systems are realizable, this system performs a valuable operation. Any method that can reduce the time needed to correct an error will aid the test engineer.

Since this system uses a structural diagnostic package, improvements to the diagnostic system can be made. As information as to the functionality of the system becomes available, a functional diagnostic system could be used to augment the results of the structural diagnosis. Extending the system to a behavioral model would also require keeping the structural diagnostic system. Each system could augment the current diagnostic system by further reducing the set of possible error causes. However, the structural diagnostic system is still needed. Some error causes might

not be diagnosable with a functional or behavioral diagnostic system, based on the system information available. These error causes could still be diagnosed with a structural diagnostic system.

While the diagnostic system used in DATVAL is not optimal, it is a valuable asset to DATVAL. Compared to the legacy system, which did not have a diagnostic system incorporated, adding this diagnostic capability is a significant improvement. In the future, as more advanced diagnostic capabilities are realized, the diagnostic system used in DATVAL can be augmented to include the new diagnostic techniques. This modular approach allows system flexibility and the possibility for system growth.

CHAPTER V

DATA DEPENDENCY ANALYSIS

Though DatVal has been successful at AEDC, some problems exist with the current system. In order to perform the diagnostics, a **dependency graph** mapping the system inputs (sensors) to the system outputs (checks) is needed (see figure 13). This graph must also include any internal variables used in calculating the system checks. While these nodes in the graph are not necessary for diagnostics, they are necessary for displaying each check's input variables and variables used in the deviation calculation. But, additional information is needed for an automatic configuration of the system. To fully configure the system, some tool must be able to extract the internal variables, the input variables, the tolerance(s), and the standard error message for each check [16].

Some method for automatically configuring DatVal was obviously needed. Otherwise, many man hours would be required for each and every DatVal configuration change. Also, several people would need to be trained in how to make the modifications necessary for updating the DatVal configuration. Due to the manpower required to reconfigure DatVal, an automatic process for updating DatVal had to be devised.

Somewhere the information about the DatVal configuration we needed was stored. First, it was necessary to examine AEDC's internal documentation and the old data validation source code. During this stage, we discovered the only source of the required information was in the old data validation code. No other source of the information existed; due to the age of the original system, all of the original design documents were not available.

Available Tools

Once the problem of automatic configuration appeared, several existing packages were examined. The packages considered include FORGE, AT&T's F2C, Omega, Sage++, extensions of Sage++, several parallelizing compliers, and custom perturbation analysis tools. Each of these tools was examined independently. We were searching for a tool that performed the necessary data dependency analysis and either allowed access to the data dependency data or the source code to the tool could be obtained. This would allow modifications to the tool which could output the needed dependency information. A more detailed analysis of the tools we examined follows.

FORGE: Forge is a commercial package designed for automatic parallelization of pre–existing Fortran code. Forge has been used successfully at AEDC for other projects, so it seemed a viable alternative at first. Forge is a commercial package, so we would not have access to the internal data structures that contain the data dependency information. Another problem with Forge is the basic Forge design. It is designed to mainly work on the parallelization of loops. Information as to if Forge does a data dependency test on other variables is not available. Forge does not deal with *common block* variables very well. This is the major problem with Forge: the original data validation software contains many common blocks. This last problem could not be overcome without a major effort [17].

Omega: Omega is another parallelization package for Fortran code. The source code for Omega is available, so access to internal data structures posed no great difficulty. As with Forge, Omega is designed to concentrate on parallelizing loops. No information is available as to parallelization of source code without time-consuming loops. Omega only works with Fortran source code. While this is not a problem for this project, it would be preferable if the data dependency

analysis suite could have interchangeable front ends. Based on these shortcomings, it was decided Omega did not fit our needs on this project [27].

F2C: As mentioned earlier in this thesis, AT&T's F2C Fortran to C source code converter is used throughout DatVal for converting existing Fortran code to C code. Since F2C is being used in the system already, it would be beneficial if we could also use F2C to extract the data dependency information. But, F2C just performs a code conversion, no parallelization or data dependency analysis is performed. Therefore, we chose not to use F2C for this section of the DatVal [22].

Sage++: Sage++ is a language restructurer designed for converting old source code into pC++ source code. pC++ is a parallel C++ language implemented by the authors of Sage++. Sage++ has multiple front ends; that is, several different language parsers exist for Sage++. These parsers convert source code into an object database representation of the source. See Figure 20 for a block diagram of how Sage++ works. Sage++ was not used due to one major problem: the basic tools available for Sage++ only look at loops for parallelization. The data dependency information for non-loop variables could not be extracted from Sage++ [24].

Sage++ Extensions: Several extensions for Sage++ exist. The most prevelant of these tools are Omega for Sage++ and TAU. However, all of the extensions had the same shortcomings as Sage++. For this reason, none of these extensions were used in the system [25] [27] [28].

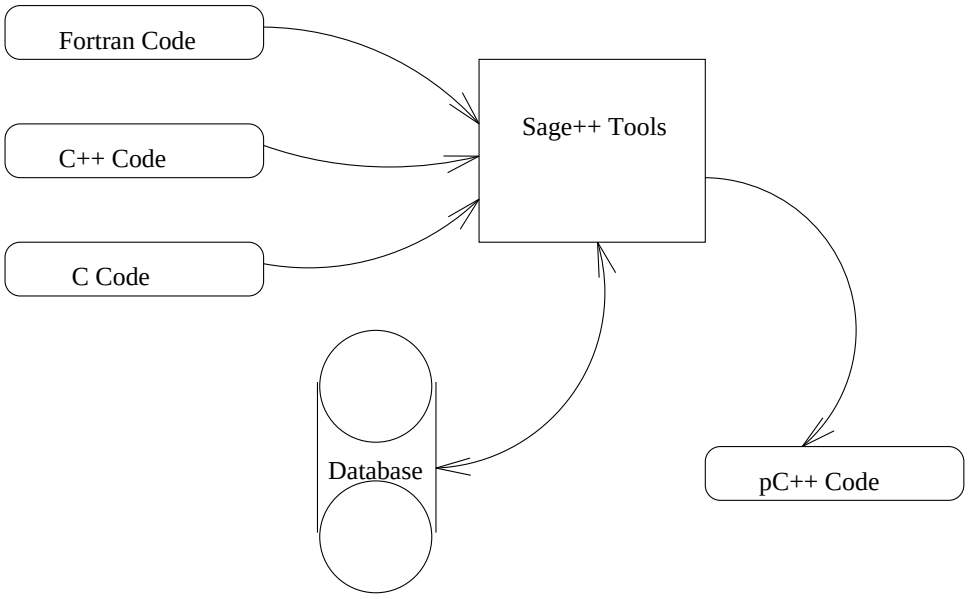

Figure 20. Sage++ Language Restructurer

Parallelizing Compilers: Both commercial and freeware parallelizing compilers were examined. Every compiler either only parallelized loops and nothing more, or did not make available the information needed for the data dependency analysis [26] [18] [20].

Perturbation Analysis: Performing a perturbation analysis consists of modifying the inputs to a software system and then recording how the outputs of the system change. However, we have over 2500 inputs to the system and these inputs can either be very sensitive or very insensitive. To fully analyze the previous data validation system required an extremely large search space to be examined. Due to this requirement, the option of performing a perturbation analysis was discarded.

After examining all of these tools, it became apparent that no available tool would suffice for performing the needed data dependency analysis. Faced with either making significant modifications to an existing software package or writing a data dependency

analysis tool from scratch, we chose the latter. It appeared much easier to start with a clean design than to hack on another, already finished, software package.

Using a source code dependency analyzer has significant advantages when dealing with *data dependent dependencies*. These are the dependencies found in, for example, a for loop with an input variable used as a condition variable. Utilizing a source code analyzer, a **conservative** method for extracting data dependent dependencies can be implemented. A conservative method will ensure that **all** possible dependencies are included. This is achieved by using the maximum, reasonable value for the condition variables. However, when an individual data point is analyzed, the presence of all of these dependencies cannot be guaranteed. It is fuller representation to build a more dense dependency graph and ensure no dependencies are ignored than to build a less dense graph. Extra dependencies would appear to the system user as possible system faults that are not actually present. While not desirable, this is a much better alternative to **not** alerting the user to a possible error cause.

Source Code Dependency Analyzer

After deciding to build a source code data dependency analyzer (SCDA) from scratch, the first step was to decide what type of parser to write for the system. This was necessary since we were going to design a source code dependency analyzer. The only binary analysis method available is a perturbation analysis. Since writing a specific parser would limit the analyzer to only source code of one type, a better solution was found. Sage++ [24] was chosen to be the parser for the dependency analyzer. By using Sage++ to parse the code, the analyzer could be built to work on the object database Sage++ uses to store parsed code. This not only allows multiple languages to be used, thanks to Sage++ and it's multiple language parsers, but eliminated

the need to write a Fortran parser from scratch. This step was instrumental in the resulting analyzer design.

Several types of programming statements were identified as important to a data dependency analysis [16]. These key statements are ones which actually have input into the assignment of values in a program. The statements are:

Assignment: Assignment statements were the easiest. The variable on the left hand side of the assignment obviously depends on **all** of the variables on the right hand side. The only problem is when the right hand side contains a function call. This is a special case, but the left hand side depends on the variables used to find the result of the function call.

Control Flow: All control flow statements (if, for, do) have some control parameter(s). Every statement contained in the body of the control flow section must depend on the parameter of the control flow statement. This is simpler than first expected, since the only statements that actually record a dependency is an assignment statement. So, for each assignment statement in the body of the control flow section, the variables used as parameters in the control flow statement must be included as dependencies.

Loops: Loops are usually considered control flow statements, but they require special treatment. All loops must be unrolled so as to allow accurate dependency information extraction.

There are several other situations that require special attention. *Common blocks* must be dealt with in a special manner. All common block variables must be kept in a linear array. This is due to Fortran using a pass by reference parameter passing convention. In a large amount of code, a common block variable is passed as a reference

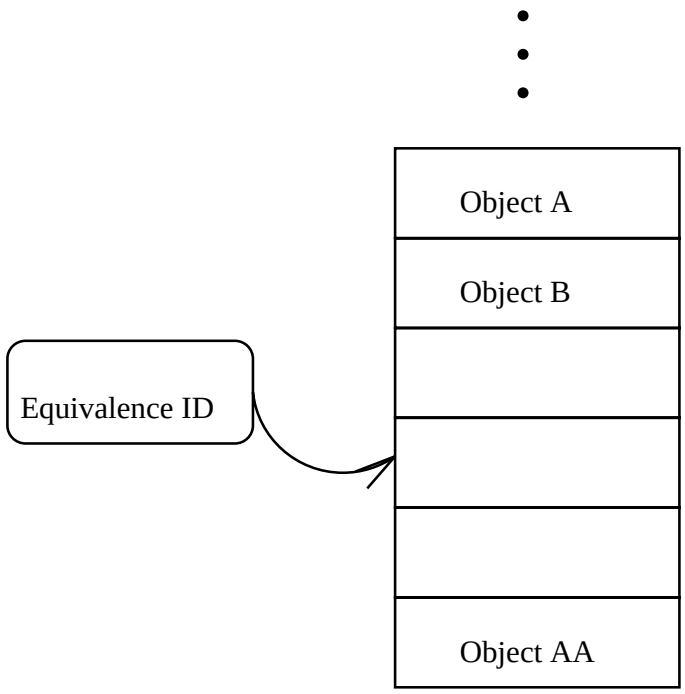

Figure 21. Equivalence Indirection

and is used to access many different elements in the common block. If the common block variables are not dealt with in a linear array, consistency between the source code and the dependency graph cannot be guaranteed. *Equivalence* statements also require special attention. An equivalence is usually used to reference another variable. Equivalence statements require a reference to the actual variable the equivalence statement refers to. Equivalence statements are often used to access sections of data in a linear fashion (i.e. an array). This indirection allows equivalence statements to be dealt with efficiently, effectively, and accurately. See Figure 21 for a graphical explanation of this indirection. One other special situation involves *temporary arrays*. These arrays are often used to group variables before issuing a function/subroutine call involving the variables. This is dealt with in a method similar to the equivalence statements. The difference is each element of the array must be a pointer to another variable. Utilizing this method ensures the correct variable will be accessed through the indirection.

The Algorithm

In order to perform the data dependency analysis, an algorithm for detecting dependencies and recording them had to be developed. As stated earlier, the Sage++ object database is used to store the information available in the system source code. This allows the dependency analyzer to operate on the objects in the database using Sage++ library calls. Using these calls the database can be traversed in a statement–by–statement manner. This allows each statement to be examined in the exact order it will execute in. This is important due to the extensive use of temporary lists of dependencies throughout the tool. All information needed to perform the analysis is extractable from the object database using the Sage++ library calls.

The algorithm follows:

Find common blocks. A list of all common block elements is built. The reasons for this were already explained. An object is built using each variables name as an identifier. These objects are in an array to ensure proper ordering.

Equivalence statements. For each equivalence statement, an object used for redirection is created. These objects are identified by the equivalenced variable's name.

Examine each statement. Each statement is then examined in order and the dependency information is stored internally.

Invoke output routine. The dependency information must be stored to disk. This allows DatVal to read in the configuration at run-time, without having to re-run the data dependency tool.

Examining each statement should be discussed further. Examining each statement is a complex procedure. In fact, it is best described by the following method:

A. Loops: If the statement is a loop, unroll the loop. This is done by examining the body of the loop. This same algorithm is then applied to each statement with one exception: elements that depend on the loop variable are appropriately updated. Currently, nested loops are not supported, but are not used in the original Fortran source code. Data dependent loops are not unrolled. To ensure a complete data dependency graph, data dependent loops should be unrolled using a conservative approach. This allows the maximum number of dependencies to be extracted. This has not yet been implemented.

B. Control Flow: For each control flow statement, build a temporary list of all variables used in the statement. Then, recursively apply the algorithm described in this chapter, adding the variables in the temporary list to each dependency list stored. Nested control flow statements are supported up to the limit of total system memory.

C. Assignment Statements: For each assignment statement, record the necessary dependency information including any temporary lists of dependencies.

D. Function Calls: Function calls are a special case. Every function call must be evaluated to determine which input variables effect the returned value. This list of variables is then added to the list of dependencies for the variables depending on the function call's result. Currently, function calls are examined first and a map of their inputs to outputs is stored in memory. Thus, the function call does not have to be examined each time; instead, a table lookup can determine the correct dependencies.

```
Common Block Declarations

var dvars adp2 adpam adt2   rpr dplsu wfet ducdpavg dlcdpavg dscdpavg;
var const1 p2tol pamtol t2tol rprtol tpttol;


Checks Section

check 2 kp2d :
names { p2 p2d dp2 }
sensors { 360..399 153 157..160}
tol {p2tol }
message {
"P2 HIGH OUT OF TOLERANCE, LOWER P2 BY %7.2f PSIA" adp2
};
```

Figure 22. Samples From a Configuration File

E. Subroutines: Subroutines are sometimes dealt with in a method similar to function calls. This is when the subroutine does not contain any references to common block variables. If a subroutine does contain references to common block variables it is expanded in–line, similar to a macro, and examined as if it were normal source code.

After this analysis procedure runs, a configuration file is stored to disk for use by DatVal. The output format can be changed to match the current target application for the dependency analyzer. This allows great flexibility in the possible uses of the dependency analyzer. For DatVal, the output format is a text file divided into two sections: a list of common block variables and a list of the information necessary for configuring each check in DatVal. A brief example taken from each section is shown below in Figure 22. The actual configuration file produced has been as large as fifty kilobytes.

Performance Analysis

The performance of the data dependency analyzer has been promising in the two most important aspects: accuracy and speed. The tool needs to be fast to allow configuration changes during a test session. Currently, execution of the tool in the background on a Pentium 90 requires approximately fifteen seconds for the current DatVal Fortran source code. This is for an approximately five thousand line Fortran program and includes the time necessary for Sage++ to parse the original source code. This extracts approximately 750 data dependencies. This is well within the time constraints for in–test configuration and analysis. In the area of accuracy, on the current configuration, the tool finds all data dependencies in the source code. This has been verified by hand using the above algorithm. This does not verify the algorithm, but does verify the current implementation. No known problems with the algorithm exist and they can/will be corrected as needed.

There are two pieces of configuration information that cannot be extracted from the source code. These items are the tolerances and the error messages. Tolerances are seen as ordinary variables to the dependency analyzer. The only identifying aspect of the tolerances are their names, which can change. Due to this, the system user must input the names of each tolerance using DatEdit. This is not needed if the user does not want the ability to modify tolerances on–line. F2C does not convert error messages in the original Fortran code into usable C source code. If the user wants the error messages displayed he must enter the messages into DatEdit. Since the error messages change very seldom, this is not seen as a major shortcoming.

CHAPTER VI

CONCLUSIONS

While several problems were solved during the course of the work, some problems remain unresolved. Other problems have solutions, but better solutions are thought possible. In this chapter, additional research is outlined and then the work completed is analyzed.

Future Research

Several new areas related to this research need to be explored. Some of the areas where work related to this research should continue are:

Generalized SCDA: The source code dependency analyzer needs to be generalized. Some areas of the existing SCDA are closely related to the data validation problem. While a truly general SCDA is not conceivable in the near future, a SCDA capable of handling most common programming constructs can be devised. This would allow the SCDA to be used in many diverse areas, such as code parallelization. Currently, the SCDA contains some DATVAL specific code. However, generalizing the SCDA would not be a difficult task. The biggest hurdle is adding the data dependent dependency analysis to the code. This does not appear difficult.

Parallelization: Also related to the SCDA is the topic of code parallelization. By using the SCDA a person could determine the overall dependencies in their code, not just in loops. This would allow for pipelined structures to be built. This could greatly enhance performance in areas where traditional parallelization methods are not applicable.

Diagnostics: Currently, the diagnostic system used in DatVal is very rudimentary due to the structural information of the system being the only currently available information. A higher–level diagnostic package could greatly enhance the usefulness of DatVal.

Other areas of research could be explored, but these three areas are thought to be the most beneficial areas of enhancement to DatVal. Improving upon DatVal does more than improve on one software system. Data validation as a whole is an important problem with many possible uses in industry. A large percentage of the research performed for this project can be applied to other problems.

Lessons Learned

Several valuable lessons have been learned throughout the course of this research project. While legacy code does not always perform as expected, it represents a huge investment in both time and money to industry. This is why many companies will not entertain the thought of reengineering the vast amounts of legacy code they own. Reengineering a complex software system requires a large allocation of resources. Only if a system has unacceptable performance is a company likely to reengineer the software. If the code performs at a minimal exceptable level it will not be replaced. Due to this, methods for revitalizing legacy code need to be devised.

Using MGA as a means of handling the complexity of a data validation system has been a success. Based on the success of MGA in other problem domains this success was expected. By enabling the user to easily modify a complex system, MGA removes the system user from the details necessary to implement a system. MGA allows non–computer engineers to build complex computer systems. In fact, the users of DatVal at AEDC are generally mechanical engineers with very limited experience with computer systems. We feel MGA has been a major cause of the success of DatVal.

One of the methods for revitalizing legacy code that has been shown to be successful is "wrapping" the legacy code with new, modern source code. This allows many new features to be added to the legacy system without the risk of modifying the legacy code. These additional features can be as complex as a full diagnostics package or related to usability, such as adding a GUI. This also allows the legacy system to be implemented in a "modern" language which should ensure the code can be maintained with less effort.

One feature often needed to be added to legacy codes is a more advanced diagnosis system. Legacy systems use diagnostic systems that are outdated both in system design and user interaction. Several of the problems associated with diagnostics and legacy codes can be solved using the methods described in this thesis. At AEDC, we have successfully incorporated a more advanced diagnostic system in the data validation system. In addition, we have left the possibility of replacing the current diagnostic system with a newer, better system with minimal effort. Currently, engineers at AEDC are in the process of designing a new diagnostics system. When complete, they will interface this new system into DatVal. This should allow the data validation system to grow gradually, instead of requiring major modifications.

One of the last items discussed here was a source code data dependency analyzer. This tool shows great promise for aiding the revitalization of legacy codes. Whenever

information is only available in the source code, a source dependency analyzer can extract this information and make it available to engineers or to other software systems. Using this information as a basis, more information can be extracted from the system while the system is in use. After a period of time, the equivalent of the information used to design the legacy system should be available to the system users. Without a doubt this can greatly enhance the use of any legacy system. Adding this knowledge base to the engineers' existing knowledge base will result in more efficient system use.

Overall, this research has been well received at AEDC. We feel this effort was a success based on the use of the system and the lessons learned from the effort. Continuation of this work shows promise as there is a large amount of legacy code in use in industry today. Using the techniques described here, new systems can be implemented that have the same functionality of the legacy system, but also add new, needed features. This will help industry to be more efficient and more effective.

REFERENCES

1. Bapty, T. Abbott, B.: "Portable Kernel for High-Level Synthesis of Complex DSP-Systems," *Proceedings of the International Conference on Signal Processing Applications and Technology*, Boston MA, 1995.

2. Sztipanovits, J., Karsai, G., Biegl, Cs., Bapty, T., Ledeczi, A., Misra, A.: "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proceedings of the International Conference on Engineering of Complex Computer Systems*, Ft. Lauderdale, Fla., October 1995.

3. Karsai,G.: "A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming," *IEEE Computer*, pp. 36-44., March 1995.

4. Bennett, K.: "Legacy Systems: Coping With Success," *IEEE Software* , pp. 19-23, January, 1995.

5. Sneed, H.M.: "Planning the Reengineering of Legacy Systems," *IEEE Software*, pp. 24-34, January, 1995.

6. Bapty, T.A., Abbott, B.: "Parallel Signal Processing for Turbine Engine Testing," Final Report for USAF-UES SRP, Contract no. F49620-88-C-0053, July 22, 1991.

7. Landau, Y.D.: *Adaptive Control*, Marcel Dekker, Inc., New York, 1979.

8. Moore, M.S., Karsai, G., Sztipanovits, J.: "Model-based programming for parallel image processing," *Proc. of the 1st IEEE International Conference on Image Processing*, 1994.

9. Abbott, B., Bapty, T., Biegl, C., Karsai, G., Sztipanovits, J.: "Model-Based Approach for Software Synthesis," *IEEE Software*, pp. 42-53, May, 1993.

10. Misra, A., Sztipanovits, J., Underbrink, A., Carnes, R., Purves, B.: "Diagnosability of Dynamical Systems," *Proc. of the Third International Workshop on Principles of Diagnosis*, pp. 239-244, Rosario, WA 1992.

11. "Research on Intelligent Process Control Systems," Dept. of Electrical Engineering, Vanderbilt University, Technical Report #88-003, 1988.

12. Karsai, G.: "Hierarchical Description Language (HDL) User's Manual," Dept. of Electrical Engineering, Vanderbilt University, Technical Report #87-004, 1987.

13. Sztipanovits, J., Bourne, J.R., "Architecture of Intelligent Medical Instruments," *Journal of Biomedical Measurements Informatics and Control,* London, UK., Vol.1, No. 3, pp. 140-146, 1987.

14. Sztipanovits, J., Biegl, C., Karsai, G., Bourne, J., Mushlin, R., Harrison, C., "Knowledge-Based Experiment Builder for Magnetic Resonance Imaging (MRI) Systems," *Proc. of the 3rd IEEE Conference on Artificial Intelligence Applications,* Orlando, FL, pp. 126-133, 1987.

15. Ledeczi, A., Bapty, T., Karsai, G., Sztipanovits, J.: "Modeling Paradigm for Parallel Signal Processing," *The Australian Computer Journal*, vol. 27, No. 3, pp. 92-102, August, 1995.

16. Banerjee, Utpal: *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Boston, 1988.

17. Friedman, R., Levesque, J., Wagenbreth, G.: "Fortran Parallelization Handbook," Applied Parallel Research, April 1995.

18. Malony, A., et. al.: "Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers," *Proceedings of the 8th International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

19. Wilkening, D., Loyall, J., Pitarys, M., Littlejohn, K.: "A reuse approach to computer–assisted software reengineering," *Proceedings of the Fourth Systems Reengineering Technology Workshop*, pp. 83-90, February, 1994.

20. Bodin, F., et. al.: "Distributed pC++: Basic Ideas for an Object Parallel Language," *Scientific Programming*, Vol. 2, Num. 3, Fall, 1993.

21. Merlo, E., et. al.: "Reengineering User Interfaces," *IEEE Software*, pp. 64-73, January, 1995.

22. Feldman, S., Gay, D., Maimone, M., Schryer, N.: "A Fortran-to-C Converter," AT&T Bell Labs, Technical Report No. 149, 1995.

23. Weide, B., Heym, W., Hollingsworth, J.: "Reverse Engineering of Legacy Code is Intractable," Ohio State University and Indiana University Southeast, Technical Report OSU-CISRC-10/94-TR-55, October 1994.

24. Bodin, F., et. al.: "Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools," *Proceedings of Oonski*, Oregon, 1994.

25. Brown, D., Hackstadt, S., Malony, A., Mohr, B.: "Program Analysis Environments for Parallel Language Systems: The TAU Environment," *Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing*, Townsend, Tennessee, pp. 162-171, May 1994.

26. Mohr, B.: "A Portable Dynamic Profiler for C++ based Languages," available on the WWW, 1992.

27. Pugh, W.: "The Omega Test: a fast and practical integer programming algorithm for dependence analysis," *Communication of the ACM*, August 1992.

28. Mohr, B., Brown, D., Malony, A.: "TAU: A Portable Parallel Program Analysis Environment for pC++," *Proceedings of CONPAR94 - VAPP VI*, University of Linz, Austria, pp. 29-40, September, 1994.

29. Bodin, F., et. al.: "Implementing a Parallel C++ Runtime System for Scalable Parallel Systems," *Proceedings of the Supercomputing 1993 Conference*, Portland, Oregon, November, 1993.

30. Nawab, H., Lesser, V., Milios, E.: "Diagnosis Using the Formal Theory of a Signal-Processing System," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol.SMC-17, No. 3, 1987.

31. Milne, Robert: "Strategies for Diagnosis," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol.SMC-17, No. 3, pp. 333-339, 1987.

32. Chandrasekaran, B., Milne, R.: "Reasoning about structure, behavior and function," *SIGART Newsletter*, no. 93, pp. 4-59, July 1985.

33. Milne, R: "Fault diagnosis through responsibility," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, Calif., Aug. 1985.

ELECTRICAL ENGINEERING

A MODEL BASED DATA VALIDATION SYSTEM

JAMES RICHARD DAVIS

Thesis under the direction of Professor Janos Sztipanovits

A model based data validation system has been developed using the MultiGraph Architecture. Several key issues were examined in this research including: the MultiGraph Architecture, data validation, diagnostic systems, legacy software reengineering, and data dependency analysis. An existing legacy data validation system was used as the basis for this research. Several issues had to be resolved dealing with the utilization of the legacy code. As part of the system, a diagnostics package was added to the legacy software. In order to accomplish the addition of a diagnostics system, structural information had to be extracted from the existing system knowledge base. A source code data dependency analysis tool was implemented to extract the needed information directly from the legacy code.

Each issue faced in the process of this research is described in detail. An overview of how MultiGraph was used to control the complexity of the data validation system is given. The algorithms for performing the data dependency analysis are outlined. Lastly, the actual implementation of the new data validation system is discussed.

Approved_____ Date_____