1994 NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM 111768

JOHN F. KENNEDY SPACE CENTER
UNIVERSITY OF CENTRAL FLORIDA

S/7-63
33977
P- 24

ENHANCEMENTS TO THE KATE MODEL-BASED REASONING SYSTEM

PREPARED BY:	Dr. Stan J. Thomas
ACADEMIC RANK:	Associate Professor
UNIVERSITY AND DEPARTMENT:	Wake Forest University Department of Mathematics and Computer Science
NASA/KSC	
DIVISION:	Engineering Development
BRANCH:	Artificial Intelligence
NASA COLLEAGUE:	Peter Engrand
DATE:	August 12, 1994
CONTRACT NUMBER:	University of Central Florida NASA-NGT-60002 Supplement: 17

Acknowledgements

I would like to thank Peter Engrand of NASA as well as Charlie Goodrich, Bob Merchant and Steve Beltz of INET for their cooperation and technical support this summer. I would also like to extend both compliments and thanks to Dr. Anderson, Dr. Hosler and Ms. Stiles for their professional management of the summer faculty program.

Abstract

KATE is a model-based software system developed in the Artificial Intelligence Laboratory at the Kennedy Space Center for monitoring, fault detection, and control of launch vehicles and ground support systems. This report describes two software efforts which enhance the functionality and usability of KATE. The first addition, a flow solver, adds to KATE a tool for modeling the flow of liquid in a pipe system. The second addition adds support for editing KATE knowledge base files to the Emacs editor. The body of this report discusses design and implementation issues having to do with these two tools. It will be useful to anyone maintaining or extending either the flow solver or the editor enhancements.

Summary

The Knowledge-based Autonomous Test Engineer (KATE) system is a model-based software system which has been developed in the Artificial Intelligence Laboratory at the Kennedy Space Center over the last decade. It is designed for monitoring, fault detection, and control of launch vehicles and ground support systems.

This report commences with a brief introduction to the fundamental principles behind the operation of KATE. Emphasis is placed on the structure and importance of KATE's knowledge-base. We then describe two software efforts implemented this summer to enhance the functionality and usability of KATE. The first addition, called a flow solver, adds to KATE a tool for modeling the flow of a non-compressible liquid in a system of pipes. The program was developed in C++ in such a way that it can be called from within KATE or used independently as a tool for solving flow problems. The second enhancement is a collection of Emacs-LISP functions which comprise a major editing mode for working with KATE knowledge base files. These functions add domain-specific features to Emacs in order to ease the task of building KATE models.

The body of this report discusses design and implementation issues having to do with these two software systems designed and prototyped this summer. It will be useful to anyone maintaining or extending either the flow solver or the editor enhancements.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

I

AN INTRODUCTION TO MODEL-BASED REASONING

1.1 BASIC PRINCIPLES

The basic concepts of model-based reasoning are very simple. A computer simulation model of a physical system is constructed from a knowledge-base representing the components of the system and their interconnections. The physical system, which is assumed to have numerous sensors, is put into operation. As the physical system operates, sensor readings are compared to their predicted values from the simulation model. As long as there are no significant discrepancies between predicted values and actual sensor readings, nothing is done. When a significant discrepancy occurs, the model-based reasoning system can carry out whatever actions are needed to alert a human that a problem has occurred. This aspect of model-based reasoning is simply known as monitoring.

If model-based reasoning systems were only capable of monitoring, they would be of limited utility. Fortunately, model-based reasoning systems such as KSC's Knowledge-based Autonomous Test Engineer (KATE) have other abilities. Among the most interesting is failure diagnosis. Once a significant discrepancy has been identified in the monitoring stage, KATE utilizes its internal representation of the physical system in an effort to identify failures which could have led to the conflict between predicted values and actual sensor values. A significant difference between this reasoning process and traditional process control techniques is KATE's ability to include sensors themselves in the diagnostic process. A high-level overview of the monitoring process and its relation to diagnosis is illustrated in Figure 1-1.

In addition to their ability to monitor and diagnose complex systems, model-based reasoning systems such as KATE have the potential for several other very useful functions. If the computer has the ability to issue commands to the physical system, it should be possible to describe a desired state of the physical system and have the reasoning system determine what commands to issue to achieve that state. If the physical system has redundant pathways and components, as is frequently the case in NASA systems, the model-based system can often determine how to utilize redundant hardware in order to continue operation of the physical system after some component or components have failed. It is also possible to have such a model-based reasoning system construct an explanation of the steps taken to identify a failed component or to achieve a specific objective.

In addition to their operational use, model-based reasoning systems have great potential as training tools. An instructor can create failure scenarios in the simulation environment to test the student's ability to respond to failures of the actual hardware.

Another potential use for model-based reasoning systems is to determine the adequacy of the sensors in a complex system before it is built. For a more in-depth introduction to model-based reasoning see [1].

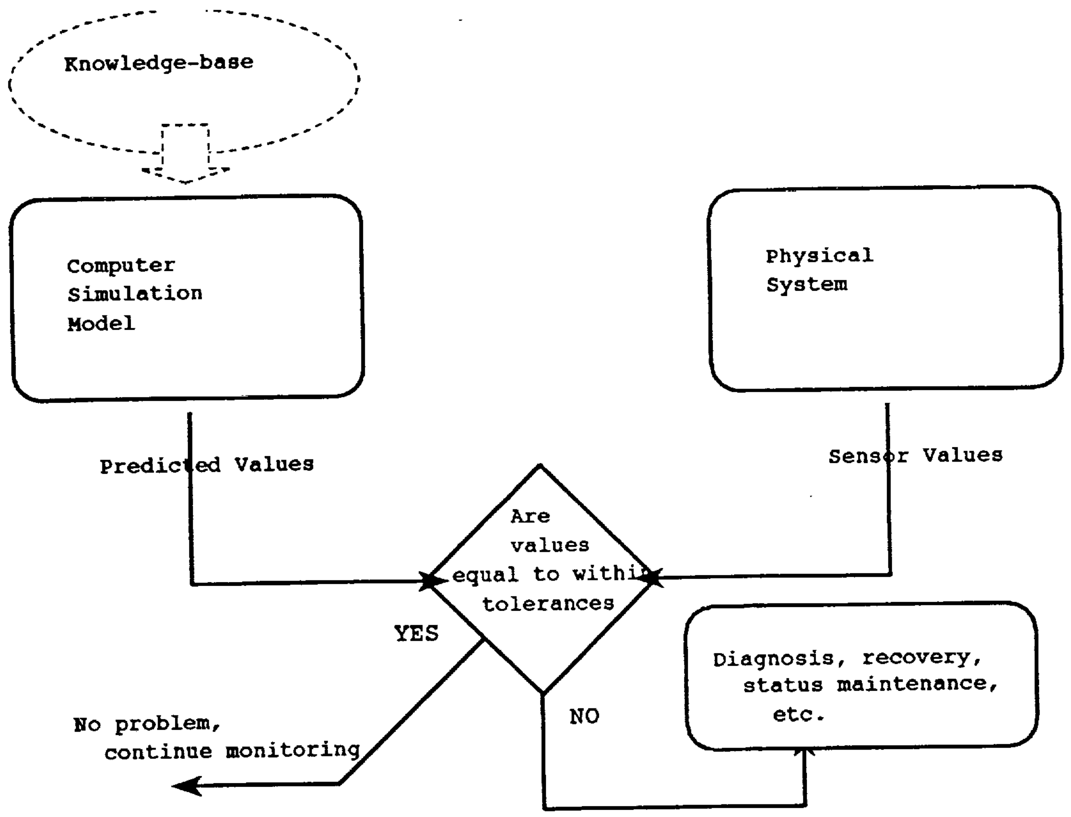
Knowledge-base
Computer Simulation Model
Physical System
Predicted Values
Sensor Values
Are values equal to within tolerances
YES
NO
No problem, continue monitoring
Diagnosis, recovery, status maintenance, etc.

Figure 1-1. Overview of Model-based Reasoning

1.2 KNOWLEDGE-BASES

In order for a model-based reasoning system to function, it must have information about the structure and operation of the physical system to be modeled. We call such information about the real world the system's knowledge-base. As characterized in [2] for the KATE system:

> The KATE knowledge base contains vital information about the physical system that KATE is controlling or monitoring. This information is the raw material used by KATE to construct a simulation model that mimics the system's structure and function. Objects in the model have a one-to-one correspondence with parameters, commands, sensors, and other components in the physical system. The knowledge-base is referenced by

KATE in the same way that schematic diagrams and operating specifications are used by system engineers.

1.2.1 THE KATE KNOWLEDGE-BASE. In order to lay the groundwork for topics discussed later in this report, we elaborate upon the organization of KATE's knowledge-base—a three-level hierarchical structure. Such hierarchical structures are typical of the organization of knowledge-bases used for model-based reasoning systems.

1.2.1.1 High level system knowledge. The so-called "top-level" of KATE's knowledge base represents information about very broad classes of system components. For the systems with which KATE is currently used, these classes are commands, measurements, components, pseudo objects, display function designators and so called synchronization objects. For operational efficiency, generic knowledge about the structure and function of these high level classes of objects is hard-coded into the C++ implementation of KATE. This means that changes to KATE's top-level knowledge of system component classes require possibly extensive modifications to the source code. Fortunately, such changes occur infrequently.

1.2.1.2 Middle level system knowledge. The so-called "mid-level" of KATE's knowledge-base represents information about specific types of system components. For example, this class contains knowledge about components such as pumps, relays, values, and tanks in addition to pseudo objects such as pressures and admittances. Each middle level class is an example of some top-level class previously defined and inherits properties from the top-level class. Again, for efficiency reasons, the mid-level of KATE's knowledge-base is represented in C++ header and source code files which are compiled into the corpus of KATE at compile time. However, modifications to the content of this level have no effect on the body of the KATE system, only the classes of components available for subsequent modeling. This level of the knowledge-base has a regular, predictable, organization and syntax which makes it easy to extend.

1.2.1.3 Low level system knowledge. The lowest level of KATE's knowledge-base is stored in what are referred to as "flatfiles". This is the information about the actual physical components in a system being modeled. Each object at this level is an instance of some mid-level class and inherits properties from that abstract class, which inherits knowledge from its parent class.

The flatfiles representing low-level knowledge are text files (ASCII files) with a well-defined keyword-based syntax. They are read by KATE at run-time in order to construct an internal representation of the physical system to be modeled.

1.2.1.4 Database files. The current practice of those building KATE knowledge bases is to work with a collection of what we will refer to as "database" or .db files. A single real-world system may be modeled using dozens or hundreds of database files. Under control of what we will refer to as a "project" or .kb file, collections of database files are compiled into a single "flatfile" representing a single KATE model. Database and project files are the files which the model builder directly constructs and edits using a text editor such as Emacs. They are discussed in more detail in Section III of this document.

II

THE FLOW SOLVER

2.1 PROBLEM DESCRIPTION

The first KATE enhancement developed this summer is known as the "flow solver". It adds to KATE the ability to model fluid flows in a network of pipes by specifying the connectivity, pipe admittances, external pressure values and static head pressures for the pipes composing the network. The internal pressures and flows in the pipes are calculated in such a way as to conserve flow at each interior node in the network while simultaneously meeting the boundary conditions imposed by the external pressures. Conservation of flow laws lead to a system of non-linear equations. This non-linearity makes the flow solver a challenging program to implement. The size of the network depends only upon the number of interior nodes in the network and can be of any size—however performance of the flow solver slows as the network grows.

Even though the flow solver code is expected to be used initially for modeling flow in the shuttle liquid oxygen (LOX) loading system, it applies to any flow system meeting the following fundamental requirements:

- o The fluid in the system is assumed to be incompressible.
- o The network is always full of liquid.
- o The interior unknown pressures to be solved for, the P_i, are pressures at the junction of exactly three pipe sections.
- o The fundamental flow law for a pipe with ends denoted k and l is
  $$FLOW = A_{k,l} * \sqrt[D]{P_k - P_l}$$
  where $A_{k,l}$ is the admittance of the pipe section from k to l, P_k and P_l are the pressures at the ends of the pipe and $\sqrt[D]{X} = sign(X) * \sqrt{|X|}$. $\sqrt[D]{X}$ is herein referred to as the directed square root of X.
- o There is conservation of flow at every junction. That is to say that the flow into each junction must be equal to the flow out of that junction.
- o The admittance for every pipe is known in advance.
- o The pressure at every "external" pipe end is known in advance.

In Figure 2-1 we illustrate the general form of the systems our flow solver can solve. Even though the connection topology is very simple, many real-world piping systems can be represented in this form. In the figure, the P_i represent known pressures whereas the $P_{?j}$ represent unknown pressures to be solved for by the flow solver. The $A_{i,j}$ represent known admittance values. In order to simplify our diagrams and the discussion we do not include references to static head pressures induced by height differences between pipe ends; however our computer programs incorporate head pressure data in a straightforward way. Although we refer primarily to solving for unknown pressures in this figure and in subsequent discussion, it should be obvious that solving for the flow in a pipe of known admittance is trivial (see equation above) once the two end pressures are known.

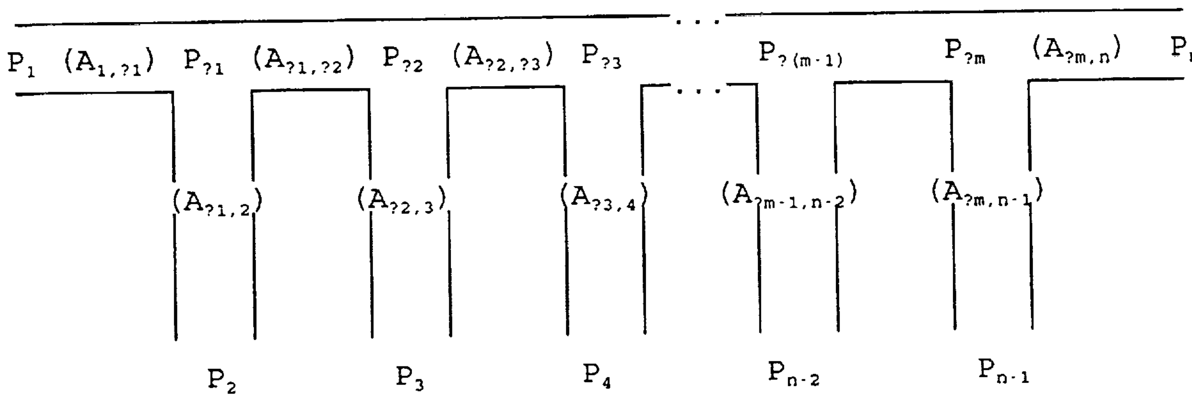

Figure 2-1. Abstract Flow Network

In Figure 2-2 we show an example network with seven known pressures and five unknown pressures. The pressure and admittance values shown were arbitrarily chosen but are illustrative of the kinds of values that might be encountered in a real pipe system.

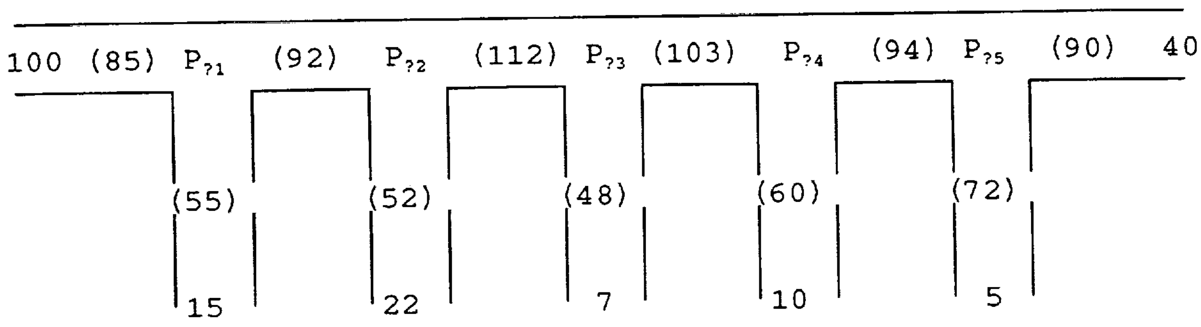

Figure 2-2. Example Flow Network

For a system with m unknown pressures out of a total of n pressures altogether, finding the m unknown pressures leads to solving a system of the form

$$A_{1,1} * \sqrt[D]{P_1 - P_1} + A_{1,2} * \sqrt[D]{P_1 - P_2} + \ldots + A_{1,n} * \sqrt[D]{P_1 - P_n} = 0$$
$$A_{2,1} * \sqrt[D]{P_2 - P_1} + A_{2,2} * \sqrt[D]{P_2 - P_2} + \ldots + A_{2,n} * \sqrt[D]{P_2 - P_n} = 0$$
$$\vdots$$
$$A_{m,1} * \sqrt[D]{P_m - P_1} + A_{m,2} * \sqrt[D]{P_m - P_2} + \ldots + A_{m,n} * \sqrt[D]{P_m - P_n} = 0$$

for unknown pressures P_1, P_2, ..., P_m. In the type of pipe systems we are restricting ourselves to there will only be three non-zero entries in each row but in more general systems, with less restricted interconnections, more non-zero terms would occur. Nonetheless, the problem of determining the unknown pressures in a pipe network reduces to solving systems of non-linear equations of this form, systems which have no closed-form algebraic solution.

2.2 NEWTON'S METHOD

The most commonly used method to solve systems of non-linear equations is the multi-dimensional variant of the well-known Newton's method for finding roots of equations. We briefly review the familiar one-dimensional Newton's method and then describe the higher dimension variant. For an in-depth discussion of these methods, see [3].

Assume that we have a known function f(x) for which we want to find x values for which f(x) = 0. Within some neighborhood of x, f can be expanded in a Taylor's series

$$f(x+\delta) = f(x) + f'(x)\,\delta + \frac{f''(x)}{2}\delta + \ldots$$

By neglecting the δX^2 and higher terms we obtain a simple equation for the corrections δX that move f(x) closer to 0, namely the well known recurrence relation

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

which is so widely used for iteratively finding roots of non-linear functions.

The multi-dimensional variant of Newton's Method is based on the same principle, the primary difference being the calculation of the derivative of the function. Assume we are given a system of n functional relations f_i in variables x_1, x_2,...,x_n, denoted $f_i(x_1,x_2,\ldots,x_n) = 0$, for i=1,2,...,n. Let X denote the vector of unknown x_i values. Then in some neighborhood of X, each f_i can be expanded in a Taylor's series

$$f_i(X+\delta X)=f_i(X)+\sum_{j=1}^{N}\frac{\partial f_i}{\partial x_j}\partial x_j+O(\delta X^2)$$

By neglecting the δX^2 terms and higher we obtain a set of linear equations for the corrections δX that move all the function values closer to 0 simultaneously, namely

$$\sum_{j=1}^{n} A_{i,j}\,\delta x_j = B_i$$

where

$$A_{i,j} \equiv \frac{\partial f_i}{\partial x_j},\quad B_i \equiv -f_i$$

so the problem reduces to one of solving a linear system $A(\delta x) = B$ and updating X iteratively until convergence. The matrix A, known as the Jacobian, is a matrix of partial derivatives which plays a role analogous to that of $f'(x)$ in the one-dimensional case.

A number of observations are in order at this point. First of all, solving a system of non-linear equations by this method is an iterative process. At each step, a linear system must be solved in order to determine the changes to be made to the elements of the vector X. Fortunately, for the class of network flow problems we want to solve, it is possible to arrange the equations in such a way that the system to be solved at each step is tridiagonal. Solving a tridiagonal system of linear equations can be done very efficiently. The process only grows in time linearly with the number of unknowns. Secondly, Newton's method converges very rapidly and predictably as long as the initial X vector values are within a "reasonable" neighborhood of the solution vector. If the initial X vector does not meet this condition then convergence is unreliable.

At this point in our work we had some difficulty finding a reliable heuristic for obtaining good starting values to use for Newton's method. Fortunately a fast, reliable, method of getting starting values for the X vector was found and that is the subject of the next section.

2.3 THE SGOS FLOW SOLVER

The Shuttle Ground Operations Simulator (SGOS) is a software system used for training, testing and validation at KSC. Beginning in the summer of 1979 an effort was made by Dr. Roy Jones and Dr. Richard Ingle of UCF to implement a flow solver in the SGOS environment [4]. Mathematically, the technique they implemented was that of using a piecewise linear approximation to the directed square root function in order to iteratively find an approximate solution to the system of equations introduced in section 2.1. In other words, each appearance of $\sqrt[D]{X}$ in the earlier system of equations is replaced by a linear function of the form $m^*X + b$ where the value of the slope m and the intercept b depend upon the actual value of X. We thus replace the original non-linear system by a linear system of the form

$$\begin{aligned}
A_{1,1} * f(P_1-P_1) + A_{1,2} * f(P_1-P_2) + \dots + A_{1,n} * f(P_1-P_n) &= 0 \\
A_{2,1} * f(P_2-P_1) + A_{2,2} * f(P_2-P_2) + \dots + A_{2,n} * f(P_2-P_n) &= 0 \\
&\vdots \\
A_{m,1} * f(P_m-P_1) + A_{m,2} * f(P_m-P_2) + \dots + A_{m,n} * f(P_m-P_n) &= 0
\end{aligned}$$

where f(x) is a linear function whose slope and intercept depend upon x. The algorithm is as follows:

1) Divide the X axis into a small number of intervals, for example, six intervals. Within each interval determine the appropriate slope and intercept value to approximate the directed square root within that interval.
2) Generate random or heuristically chosen starting values for the unknowns $x_1, x_2, \dots, x_n$.
3) Repeat the following for some fixed number of iterations or until all the x_i values stay within the same x-axis interval for two consecutive iterations:
   3a) Construct a linear system using the current x_i approximations as appropriate. The system will be of the form $A(\delta x)=B$ where the A and B matrices are derived from known coefficients in the system. The δx values represent changes to be applied to the unknown x_i's.
   3b) Solve the linear system for the δx values using whatever technique is available.
   3c) Update the x_i vector values.
4) Test the current solution values for $x_1, x_2, \dots, x_n$ by plugging these values into the system and determining the error. Each equation's left and right hand side values should be 0.
5) If the solution obtained is acceptable then terminate, otherwise go back to step 1 and subdivide the X axis into a greater number of intervals.

This algorithm rapidly obtains approximate solutions to the original system of non-linear equations. Acceptable solutions are usually obtained in a few dozen iterations using from six to ten sub-intervals of the X-axis.

For our purposes we implemented the above algorithm with several refinements and enhancements. For example, our starting X values are chosen to be within the known upper and lower bounds of the external pressures of the pipe network. We use a very efficient LU decomposition technique to solve the linear system as mentioned in step 3b. Our implementation also generates solutions with differing numbers of sub-intervals and selects the solution with the smallest total error in the system, an improvement over the SGOS version of the algorithm.

We use this algorithm in two ways. First of all, we use it to compute starting values to be utilized by the multi-dimensional Newton's method. Secondly, we retain the SGOS solution values in the unlikely event that Newton's method does not converge or will not work due to being given a singular matrix to solve. Such situations have not occurred in our testing of Newton's method to date but the availability of the SGOS solution values is, nonetheless, felt to be an important fall back.

2.4 IMPLEMENTATION OF THE FLOW SOLVER

The flow solver was implemented in approximately fifteen hundred lines of C++ code. The basic outline of the program is as follows:

1) Gather vectors containing the known pressures, known admittances, and optionally, static head pressures. These inputs can be obtained interactively from the keyboard or passed as parameters from the KATE system.
2) Generate starting values for the unknown pressures by averaging the external known pressures. The known external pressures give us useful upper and lower bounds for the unknown pressures.
3) Use our implementation of the piecewise linear approximation concept, as in SGOS, to get approximately correct starting values for the unknown pressures.
4) Use the multi-dimensional Newton's method approach to obtain a highly accurate solution for the unknown pressures.
5) In the unlikely event of a problem in step (4), return the approximate solutions obtained in step (3), otherwise return or report the best solution found.

Although there are many steps to this process, the program returns extremely good solutions, with errors less than 10^{-12}, very quickly, on the size of systems expected to be encountered in practice.

It is instructive to compare the results obtained from the SGOS algorithm with those obtained in step (4) above from Newton's method. In typical experiments, the SGOS values are plus or minus 1% of the value obtained by Newton's method. This statement causes one to ask whether or not the additional work of using Newton's method is necessary. In Figure 2-3 we tabulate the results of applying both the SGOS algorithm and the complete Newton's method algorithm above to the example flow system from Figure 2-2. The values have been rounded to two decimal places for ease of reporting. Though randomly selected, these results are typical of those found repeatedly when comparing the two algorithms. Note that the absolute differences between the SGOS solutions and the Newton's method solutions are typically in the third or fourth significant digit. However, when the pressure values are used to calculate flows, these differences become greatly magnified. The non-linear nature of the flow network and the interdependent nature of the overall network result in very small errors in unknown pressures translating to sizable errors in flow. Given that these calculations of unknown pressures may be repeated hundreds or thousands of times by KATE during the LOX loading period, one begins to see that even small flow errors could cause the flow model to differ considerably from the actual flow by the end of the simulation period.

Unknown Pressure	SGOS Solution	SGOS Flow Error	Newton's Method Solution	Newton's Method Flow Error
1	41.58	-2.08	41.69	$<10^{-13}$
2	25.93	0.54	25.96	$<10^{-13}$
3	20.48	-0.19	20.52	$<10^{-13}$
4	19.80	3.25	19.84	$<10^{-13}$
5	20.92	-6.47	21.05	$<10^{-13}$
	Total Abs Error	12.53		$<10^{-12}$

Figure 2-3. Comparison of Results

The flow solver code has been interfaced with KATE and testing will soon be underway using recorded LOX loading data from previous launches.

2.5 FUTURE WORK

The flow solver program meets all the design requirements originally set out. The only major enhancement that may be needed is the ability to work with pipe networks with a more general topology than the "straight-line" systems now targeted. As stated previously, the current design assumes that each interior unknown pressure is connected to exactly three other pressures and at least one of those is always a known exterior pressure. Solving more general classes of networks will require work primarily in the data interface and in implementing a technique for collapsing clusters of connected unknown pressures into a single unknown. The data interface may require some type of adjacency matrix or adjacency list to represent the interconnections of the network. Performance of the program will also degrade markedly because Newton's method will require solving general NxM matrices rather than the current tridiagonal systems.

III

THE EMACS KATE MODE

3.1 DESIGN SPECIFICATIONS

Constructing a KATE knowledge-base is a critically important and time-consuming task. Starting with the predefined top-level KATE classes and a predefined library of mid-level components, the model builder proceeds along the following lines:

- o Gather together schematics and engineering documents for the physical system to be modeled.
- o Study the target system to gain an understanding of its principal components and their interactions.
- o Determine whether or not the existing middle level component classes are adequate for the system to be modeled. If not, add new component classes. This requires at least some C++ code to be written.
- o Construct database (.db) files for the components, commands and measurements in the physical system. Real world systems may contain over one hundred such files, each containing dozens or hundreds of entries.
- o Specify the interconnections among the components in the database files.
- o Construct a project (.kb) file to be used to coordinate the processing of all the various files which go into making the end product, a "flatfile" which can be loaded into KATE.
- o Add pseudo objects to represent logical functions of groups of components in the system.
- o Run a "make" program to compile together all the various database files describing a model and create a single flatfile from them.

Until now there have been no software tools available to assist the KATE model builder. Most of the work is done using a text editor, frequently Emacs, and there is no way for the model builder to view the model under construction except as a collection of text files. We have designed and implemented a number of enhancements to the Emacs editor to make the job of the model builder easier.

The goals for this work were as follows:

- o Design and implement an improved editing environment for building KATE models. The more the system knows about the structure of knowledge bases, the better.

- No editing abilities should be forsaken in this environment. That is, the environment must have all the features of a sophisticated editor such as cut and paste, configurable pull-down menus, mousing abilities, undo, backup, and rollback.
- The editing environment should be portable across all the platforms that KATE runs on. That currently means Unix, Linux and Microsoft Windows.
- The editing environment must be a stand-alone program. It should not be necessary to have KATE installed or operating on the machine where editing is being done.
- The editor should be able to construct graphical diagrams showing the interconnections between various components in a complete or partially complete model.
- The editing environment must be able to use the organizational information present in project (.kb) files to assist the model builder in organizing complete models and generating flatfiles for them.
- The editing environment should not be built on top of a commercial editor product. When KATE is made available to the general public this editing environment should be freely distributable.

3.2 IMPLEMENTATION

After considering several possible alternatives, including developing an editing environment from scratch, and considering the limited time available to the investigator, it was decided that most of the design goals would best be met by implementing an editing environment as an extension to the GNU-Emacs editor. Emacs is arguably the world's most sophisticated and powerful text-editing environment. It is implemented primarily in Emacs-LISP, a subset of Common LISP and is, in fact, as much a programming environment as a text editor. Recent versions of Emacs support menus, mouse operations, practically unlimited undo capabilities and can even take advantage of some of the features of the X-Windows system. Emacs runs on practically all Unix platforms and is now available for DOS and Microsoft Windows. And it is distributed at no cost to the end user.

In the Emacs terminology we elected to implement an Emacs "major-mode". We have programmed Emacs to automatically recognize when database (.db), project (.kb), or flatfile (.flatfile) files are loaded and make the transition into what we call "kate-mode". One of the primary features of this mode is that tables called tag tables and tables of input connections are automatically constructed as the database files and flatfiles are loaded. These tables form the basis for what might be thought of as a cross-referencing feature of the Emacs environment, called tagging. No matter what file the model builder is editing, he/she can, with only a few keystrokes, quickly find the file and section of code where an object is defined. This feature can be used not only for matching object names exactly but also for partial, substring, or apropos matches.

A second major feature of kate-mode is the ability to quickly generate simple hierarchical tree drawings representing the connections between an arbitrary object and its upstream and downstream neighbors. This feature works very rapidly because the requisite connection information is constructed once as the files in a project are loaded and then stored in a buffer for later access.

Another kate-mode feature is the ability to compile a collection of database files into a flatfile under control of a project file from within the editor. This process is currently done by a LISP program run on a Symbolics system, requiring several sequential steps to be carried out by the model builder on different platforms. Making this compilation process an integral part of the editor will in itself save a great deal of effort.

There are a number of other features of kate-mode which are expected to make it very useful. For example, the model builder need not be aware of any of the LISP programming being used to make his/her environment easier to use. Much of the operation of kate-mode occurs automatically and is totally invisible to the end user. The non-automatic features can be assigned to Control or Escape key sequences or can be utilized by typing in the name of a function which will then prompt the user for any necessary parameters. The system also attempts to save as much tag and connection information as it can between editor runs in order to save time when restarting or continuing an editing session.

3.3 FUTURE WORK

The current version of kate-mode is approximately fourteen hundred lines of Emacs-LISP code. Additional features can be added by anyone familiar with LISP programming. Incorporating new features can be done in a modular and straightforward way without a negative impact on existing features. We feel that the original design goals have, for the most part, been achieved. The display of the input connection relationships among objects is perhaps the weakest aspect of the current implementation. The information to create the diagrams has been extracted and can be accessed very quickly but it is just very difficult to display structural relationships using only character graphics. The information could be passed to a program running externally to Emacs which would display the connections graphically in a separate window. However, we have not had the time to pursue this idea. Ideally, the model builder should be able to manipulate the graphical representation, for example adding or modifying input connections, and have these actions reflected in the corresponding database files. This too could be done using Emacs as the "control center" to modify the underlying text files with actions performed in another process supporting a graphical interface, but we have not yet looked into the details.

IV

REVIEW

We have presented the design criteria and described the implementations of the two software projects undertaken this summer. In the case of the flow solver and the editor enhancements we feel that we have made very useful enhancements to both the development and application environments for KATE. We believe both tools will prove to be of great utility to future KATE users.

REFERENCES

[1] Steven L. Fulton and Charles O. Pepe, "An Introduction to Model-Based Reasoning", *AI Expert*, January 1990, pp. 48-55.

[2] Charles O. Pepe, et. al., *KATE - A Project Overview and Software Description,* Boeing Aerospace Report, Boeing Aerospace Operations, Mail Stop FA-78, Kennedy Space Center, Florida.

[3] Burden, R.L, Faires, J.D., and Reynolds, A.C., *Numerical Analysis,* Prindle, Weber, and Schmidt, Boston, MA, 1981.

[4] Ingle, R.M., *Modeling Fluid Networks Using the SGOS Flow Solver,* Lockheed Space Operations Technical Report, 35SM-FS01-06, August, 1984.