1993 NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM

JOHN F. KENNEDY SPACE CENTER
UNIVERSITY OF CENTRAL FLORIDA

EXPANDING THE KATE TOOLBOX

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 6, 1993
CONTRACT NUMBER:	University of Central Florida NASA-NGT-60002 Supplement: 11

Acknowledgements

I would like to thank Peter Engrand of NASA, Charlie Goodrich, Chuck Pepe and all my INET colleagues for their cooperation and technical support this Summer. Thanks and best wishes also go to Li Yang as she continues this work. I would also like to extend both compliments and thanks to Dr. Hosler and Ms. Stiles for their professional management of the 1993 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. In order to bring KATE to the level of performance, functionality, and integratability needed for firing room applications, efforts are under way to implement KATE in the C++ programming language using an X-windows interface.

This report describes two programs which have been designed and added to the collection of tools which comprise the KATE toolbox. The first tool, called the schematic viewer, gives the KATE user the capability to view digitized schematic drawings in the KATE environment. The second tool, called the model editor, gives the KATE model builder a tool for creating and editing knowledge base files. 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 schematic viewer or the model editor.

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. In order to bring KATE to the level of performance, functionality, and integratability needed for firing room applications, efforts are currently under way to implement KATE in the C++ programming language using an X-windows interface. The version of KATE currently under development is called KATE-C; however, we will omit this distinction and refer to KATE generically.

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 programs which have been designed and added to the collection of tools comprising what is called the KATE toolbox. The first tool, called the schematic viewer, gives the KATE user the capability to view digitized schematic drawings in the KATE environment. This tool was designed to illustrate the potential for integrating real-life schematics into the operation of KATE. The second tool, called the model editor, gives the KATE model builder a tool for creating and editing knowledge base files. Without such a tool the KATE model builder has to create and edit knowledge bases outside of the KATE environment using a traditional text editor. The body of this report discusses design and implementation issues having to do with these two software tools which have been designed and prototyped this summer. It will be useful to anyone maintaining or extending either the schematic viewer or the model editor.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

I

AN INTRODUCTION TO MODEL-BASED REASONING

1.1 BASIC PRINCIPLES

The premise of model-based reasoning is extremely 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 must 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 carries out whatever actions are necessary 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 powerful capabilities. 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 relationship 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 can be issued 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 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 or student can create failure scenarios in the simulation environment to test the student's ability to respond to almost any failure in 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].

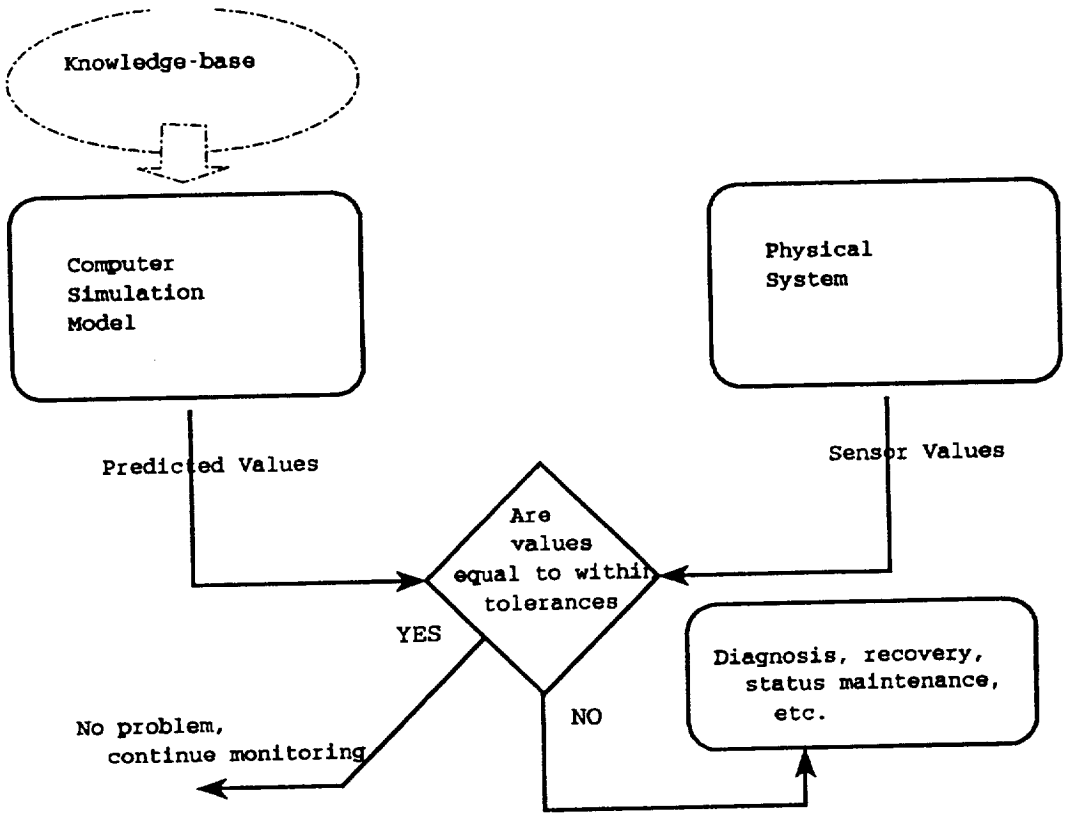

Figure 1-1. Overview of Model-based Reasoning

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 requires possible 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 is an example of some top-level class described above and inherits certain 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 add to.

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 textfiles (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.

II

THE SOFTWARE ENVIRONMENT

2.1 PROGRAMMING LANGUAGE AND GRAPHICS INTERFACE

Several implementations of the general principles underlying KATE have been carried out over the past decade under various titles, for example, the LOX [3] and ECS[2] systems. These systems were originally developed in LISP—the traditional language for rapid prototyping of AI systems—as a proof of concept.

In order to bring KATE to the level of performance and functionality needed for firing room applications, current efforts focus on simultaneously porting KATE to the C++ programming language while advancing its capabilities. C++ is an object-oriented language developed at Bell Laboratories. It is a derivative of the C programming language with features which encourage the writing of modular, reusable code. The work described in this report required the investigator to achieve proficiency in the C++ programming language.

For a software system as complex as KATE to be accepted and used by firing room personnel it must have an excellent user interface. At the same time, the KATE environment must be accessible from several different hardware platforms. In order to achieve both of these goals, the user interface for all KATE modules currently under development must adhere to the Motif/X-Windows graphics user interface standards. As with C++, it was necessary for the investigator to develop proficiency with these tools in order to carry out the work described herein.

2.2 FRAME UTILITIES

In an effort to standardize the user interface of KATE, user interface components are implemented as so-called "frame utilities". A frame utility interacts with the main body of KATE in a specific way and inherits graphic and functional properties from the KATE user interface environment. Working with frame utilities provides conveniences for the implementor but, at the same time, places constraints on the appearance and operation of the code being developed.

III

THE SCHEMATIC VIEWER FRAME UTILITY

3.1 REQUIREMENTS

The first KATE utility developed this summer is known as the Schematic Viewer Frame Utility (SVFU). The goal was to add to the set of tools available in the KATE environment a way for the KATE user to view schematic drawings. Specific requirements for this program were:

- The SVFU must operate as a frame utility.
- The SVFU should allow a KATE user to view digitized schematic drawings within the KATE environment.
- The SVFU should allow a KATE user to view system overview drawings created using a "painting" program.
- The SVFU should be useful for demonstrating the potential for accessing digitized information within the KATE environment.
- The SVFU should be designed and implemented in such a way that KATE model and hardware values can be superimposed on the drawings while KATE is executing.

3.2 ACHIEVEMENTS

A frame utility was developed which met the requirements listed above. Figure 2-1 shows the appearance of the SVFU within the KATE user interface. In this example a schematic drawing was digitized and loaded into the SVFU. Any image file in either the X-windows bitmap (.xbm) or X-windows pixmap (.xpm) format can be loaded and viewed. Using the Portable Pixmap library from MIT, almost every graphics and image file format can be converted to a bitmap or pixmap which can then be viewed in KATE. The SVFU allows the user to load multiple image files and to then switch among them quickly and easily. The utility has been demonstrated to potential KATE users and has been well received. There has been discussion of the possibility of having digitized schematic drawings stored on compact disc for loading into the SVFU.

3.3 ENHANCEMENTS

Work has begun on superimposing KATE values on top of drawings in the SVFU. This should be a very useful tool for console operators using KATE.

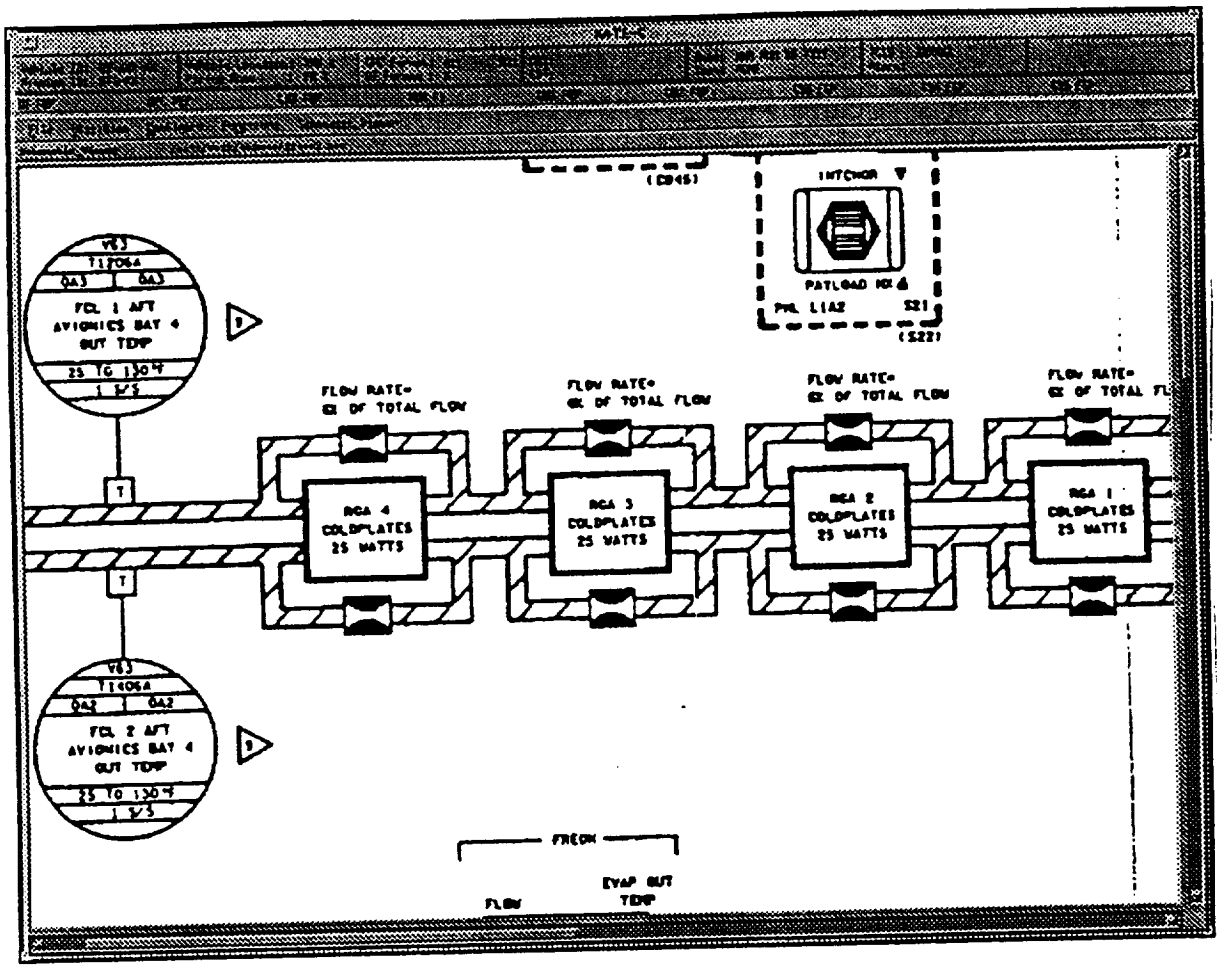

Figure 3-1. The Schematic Viewer within the Kate User Interface

Loading large files into the SVFU is time consuming. The files loaded into the SVFU need to be in a compressed format. This would greatly reduce the storage space required for storing large collections of digitized schematics and would improve the time to load a schematic into the system. This modification will require the SVFU to read image files stored in at least one widely used compression format. The ability to read various image file formats, compressed and uncompressed, would be ideal.

The initial version of the SVFU does not support the quick-loading feature expected of a frame utility. Several menu buttons for such things as "Help" options are not functional. The program has not been thoroughly checked for potential memory leaks. These are all issues which can be resolved quickly.

IV

THE MODEL EDITOR

4.1 MOTIVATION

Constructing a KATE knowledge-base is a critical and time-consuming task. Starting with the pre-defined top-level KATE classes and a pre-defined 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 a flatfile for the physical components, commands and measurements in the physical system.
- o Specify the interconnections among the components in the flatfile.
- o Add to the pseudo objects to represent logical functions of groups of components in the system.

There are currently no tools to assist the KATE model builder. All work is done using a text editor and there is no way for the model builder to view the model under construction except as a collection of text files. This investigator undertook to design and implement sophisticated, graphically oriented tools to assist in the process of constructing KATE knowledge-bases.

4.2 DESIGN

With the benefit of preliminary design documents and discussions with INET engineers an initial design for a knowledge-base editing environment was developed. This design is illustrated in Figure 4-1. The principal feature is that of two distinct editing environments, one for the flatfile level of a model and another for the middle level components. Henceforth, we shall refer to those portions of an editing environment which have to do with editing the flatfile level of a model as the flatfile editor (FFE) and to those portions which pertain to editing the middle level components as the middle level editor (MLE). In Figure 4-1 the FFE obtains a description of the middle level of a knowledge base by reading a file containing enough information about each middle level component to allow a flatfile instance of a component class to be edited intelligently. The MLE is responsible for editing and updating not only this middle level description file but also the mid-level C++ header and code files.

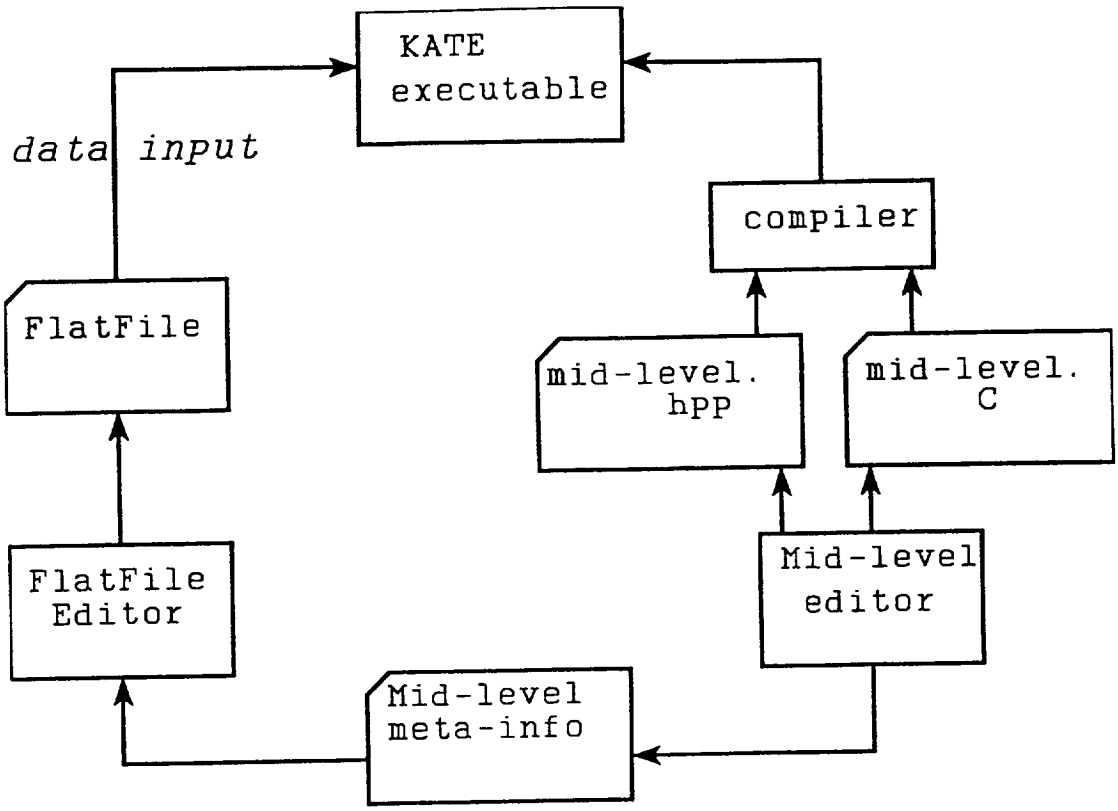

Figure 4-1. Preliminary Design for a Model Editing Environment

Upon further study it was decided that it might be possible to eliminate the additional file of meta-level descriptions of the middle level components by gleaning the required information directly from the mid-level header (.hpp) files. This revised design, illustrated in Figure 4-2, was pursued for the remainder of the project.

4.3 REQUIREMENTS

4.3.1 THE FLATFILE EDITOR. With the assistance of existing design documents and with an overall design plan in hand, specific requirements for the FFE portion of the model editor were developed. They are:

- The flatfile editor should be a frame utility. In the short term this requirement is not critical. However, there are two compelling reasons for having the flatfile editor tightly integrated into the KATE user interface. First, the simulation capability of KATE could be used to assist in the verification of a model while still under development. This could greatly enhance the ability of the model builder to create a thorough, accurate model

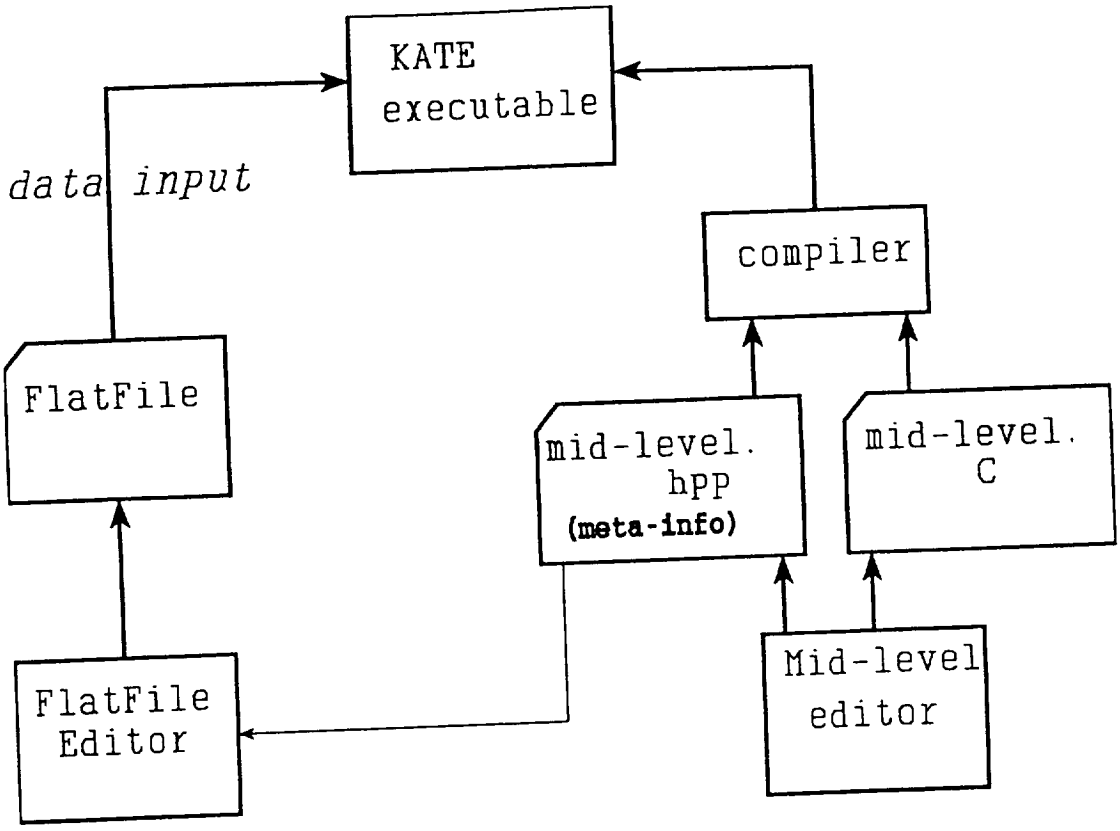

Figure 4-2. Revised Design for the Model Editor

without repeatedly having to reload and test the model. On the other hand, a tightly integrated environment could be used to make small corrections to a model already loaded into KATE.

- The flatfile editor should react transparently and correctly to changes in the middle level of the knowledge base. Practically speaking, this means that the connection shown in Figure 4-2 between the middle level header file (mid-level.hpp) and the flatfile editor should be the only connection needed between knowledge about the middle level of a knowledge base and the flatfile editor.
- The flatfile editor should be easy to modify when the top level of the knowledge base changes. Although any good software should be easy to modify we are making a slightly more specific requirement here.

4.3.2 THE MIDDLE LEVEL EDITOR. The only specific requirement for the middle level editor is that it should edit both the C++ header and C++ source code files composing the middle level of a knowledge base. Given the software organization of KATE in C++ it is not necessary for this editor to be a tightly integrated part of the

KATE environment. Any change to the middle level of the knowledge base requires the entire KATE system to be recompiled and linked. At this time no further work has been done on this portion of the editing environment. The remainder of this report deals exclusively with the flatfile editor.

4.4 IMPLEMENTATION OF THE FLATFILE EDITOR

A significant portion of the flatfile editor has been implemented. The following discussion will describe the major components of that program at a level which would be useful to someone endeavoring to continue development of this software.

4.4.1 OVERVIEW. At the highest level, the FFE works as a frame utility. The principal C++ class used is called the FFEFrameUtilityClass. Instances of this class manage three principal subdivisions of the screen along with a list of structures related to the individual files loaded into the editor at any point in time.

The FFE's portion of the screen is subdivided into a menu bar, a list area and a drawing area. The menu bar is used to select editing commands and options for the editor. The list area is used to list the names of the objects in the flatfile currently being edited. Several options have been proposed for restricting and organizing the content of this list window. The drawing area can currently only be used to display a tree-like representation of the flatfile being edited. This drawing area conceivably could be used to display an icon-based representation of the flatfile being edited.

The FFEFrameUtilityClass also maintains a list called the FlatFileList. Each item on this list is itself an instance of a C++ class containing information specific to a single flatfile. Along with obvious entries such as the name of the flatfile, each instance contains several entries including a list of the objects in the flatfile and a flag indicating whether or not any editing changes have been made since the file was saved to disk.

Two other important classes of objects are the TopLevelClass and the MidLevelClass objects. The TopLevelClass represents useful information about the editing of top-level KATE classes. The MidLevelClass represents similar information for each mid-level component class. For example, the MidLevelClass instance for a Pump would contain, among many other items, the names of the input values expected for a Pump. The information for the MidLevelClass is gathered almost exclusively at run time by scanning through a mid-level header file. Interestingly enough, the C code to carry out this scanning is generated by the UNIX tool *lex*. If the syntax of the middle level KATE classes changes, the code which scans the mid-level header file can be modified by making simple changes to the *lex* file describing the structure of header files.

4.4.2 OPERATION. When the FFE is loaded, a pane is obtained within the KATE runtime user interface. At this point the user has two choices—load an existing flatfile

for editing or create a new flatfile from scratch. If the user chooses to load an existing file, the file is read into a KATE knowledge-base structure and this structure is then used to create a list of the flatfile objects for editing. A list of the flatfile contents is shown in the list area and a tree diagram is automatically generated. The user can subsequently either select items for editing or add additional items to the flatfile under consideration. Whenever the user clicks on an item in either the list window or the tree window, an edit template is generated containing known information about the component. The user can then modify the item by typing in the template. Similarly, if the user wants to add items to a flatfile, he obtains a list of top-level classes and from a class obtains a list of all the known mid-level components for that class. Selecting one of these mid-level classes results in a blank template for the user to edit. In both editing situations, the editing templates popped up are constructed dynamically to correspond to the specific class of object being edited. The information to do this is derived from the TopLevelClass and MidLevelClass mentioned previously.

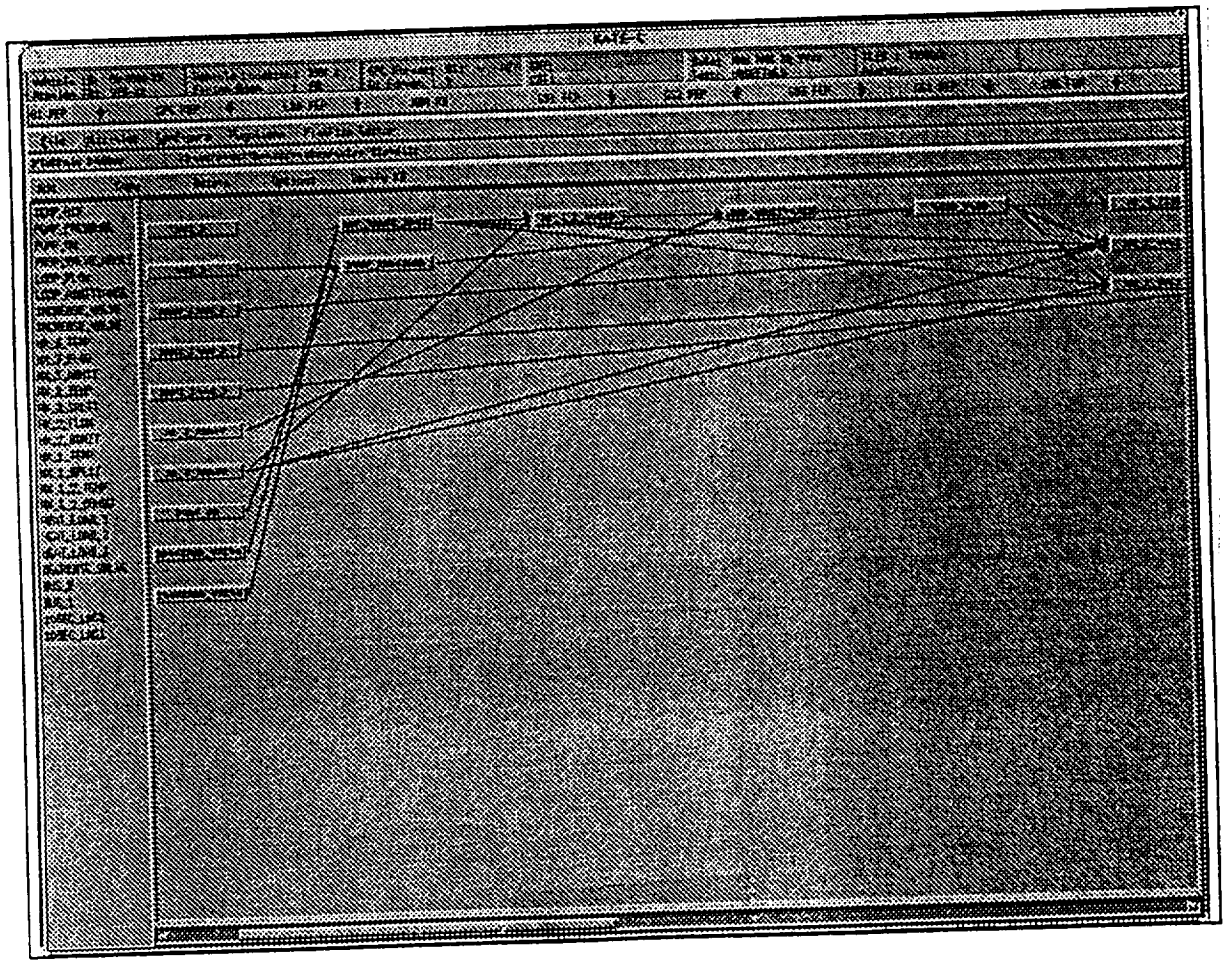

Figure 4-3. The ECLSS Flatfile in the Flatfile Editor

At any time during the editing process the user can elect to save the edited file, switch to editing a different file, or use other KATE utilities such as the Schematic Viewer Frame Utility. Figure 4-3 illustrates an editing session. The names of the objects in the flatfile are listed on the left and a tree drawing is on the right. The menu bar can be seen just above these areas.

4.4.3 STATUS. Much of the development effort this summer has been directed toward constructing a well-conceived foundation upon which the flatfile editor can be implemented. The major functions of the editor have been completed but many details remain. The automatic tree drawing capability of the FFE is in its infancy. There are probably a number of special cases in which the popup editing templates do not have exactly the correct editing fields displayed. It was not possible to incorporate the newly defined Synchronization Objects added to KATE this Summer. Also, the class of so-called TimeDependentObjects has not been dealt with. The Verify option on the menu bar is not implemented.

V

REVIEW AND RECOMMENDATIONS

5.1 CONCERNS

For the most part the design illustrated in Figure 4-2 has proven to be a satisfactory approach to the implementation of the FFE. We believe this organization will prove the easiest and most flexible editing organization to maintain as KATE evolves. There are however some significant problems which should be addressed if the FFE is to have long-term viability.

First of all, the scanning of the middle level header file (mid-level.hpp) is done in a very *ad hoc* manner. The structure of several header files was examined and certain rules of thumb about their structure were adequate to implement a scanner using *lex*. However there is no reason that a model builder constructing a middle level header file would necessarily stick with the same format. Any significant change to the structure of the middle level header files will necessitate updates to the *lex* grammar file. For long-term viability of the FFE an actual grammar should be constructed for the header files and a genuine parser should be developed. The middle level knowledge base is also being broken up into multiple files. This will require some immediate minor restructuring of the FFE to accomodate this change.

The second major concern is that the current prototype FFE does not do a good job with knowledge base constructs outside of its range of knowledge. For example, if a flatfile containing Synchronization objects is read into the FFE and then written back to disk, those Synchronization objects will not be written back to disk because as far as the FFE is concerned, they never existed in its realm of knowledge. A similar statement applies to "comment" fields within a flatfile which is read, processed, and written back to disk. Currently we do not envision a simple solution to this problem since the FFE actually works solely with an internal representation of the flatfile and does not retain any other type of "image" of the file being edited.

5.2 OTHER REMARKS

A significant amount of work has been completed toward the design and implementation of a knowledge-base editing environment. The FFE integrates very well into the KATE user interface environment and will, when complete, be of significant benefit to KATE model builders. Except for the concerns mentioned above, we expect development to continue along the lines in which it is currently headed. Much of the remaining work should be in the nature of flushing out and completing modules already partially developed. With the exception of the icon drawing components, the FFE infrastructure is complete.

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] Delaune, C.I., Scarl, E.A., and Jamieson, J.R., "A Monitor and Diagnosis Program for the Shuttle Liquid Oxygen Loading Operation", *Proceedings of the First Annual Workshop on Robotics and Expert Systems*, Johnson Space Center, Houston, TX, June 1985.