KNOWLEDGE FROM PICTURES (KFP)

Walt Truszkowski
Code 522.3
NASA Goddard Space Flight Center
Greenbelt, MD 20771

Frank Paterra, Sidney Bailin
CTA Incorporated
6116 Executive Boulevard, Suite 800
Rockville, MD 20852

ABSTRACT

The old maxim goes: "A picture is worth a thousand words". The objective of the research reported in this paper is to demonstrate this idea as it relates to the knowledge acquisition process and the automated development of an expert system's rule base. A prototype tool, the Knowledge From Pictures (KFP) tool, has been developed which configures an expert system's rule base by an automated analysis of and reasoning about a "picture", i.e., a graphical representation of some target system to be supported by the diagnostic capabilities of the expert system under development. This rule base, when refined, could then be used by the expert system for target system monitoring and fault analysis in an operational setting.

Most people, when faced with the problem of understanding the behavior of a complicated system, resort to the use of some picture or graphical representation of the system as an aid in thinking about it. This depiction provides a means of helping the individual to visualize the behavior and dynamics of the system under study. An analysis of the picture, augmented with the individual's background information, allows the problem solver to codify knowledge about the system. This knowledge can, in turn, be used to develop computer programs to automatically monitor the system's performance. The approach taken in this research was to mimic this knowledge acquisition paradigm. A prototype tool was developed which provides the user: 1. a mechanism for graphically representing sample system-configurations appropriate for the domain, and 2. a linguistic device for annotating the graphical representation with the behaviors and mutual influences of the components depicted in the graphic. The KFP tool, reasoning from the graphical depiction along with user-supplied annotations of component behaviors and inter-component influences, generates a rule base that could be used in automating the fault detection, isolation, and repair of the system.

INTRODUCTION

This paper details the results of the Knowledge From Pictures (KFP) work. The continuing objective of this work is to develop a system that can build a knowledge base to perform Fault Detection, Isolation, and Recovery (FDIR) from an annotated graphical description. Specifically, the KFP tool should take a user defined graphical image of a system's components and interconnections, and drawing from domain specific libraries for component behavior, develop an expert system to perform FDIR processing of the system defined. The user defined graphical image is also intended to be used as a user interface or front end to the generated knowledge base.

As stated above, this work is motivated by the observation that pictures are often drawn to describe the operation or behavior of many systems and problems that humans address. For example, describing the three basic parts of an atom and how they interact is most easily done with a picture showing protons and neutrons tied together in the center and electrons orbiting around them. Other examples are discussed by Musen et al in (6) and Montalvo in (5).

This work draws on results by Navinchandra et al in (7) and Barker-Plummer and Bailin in (1). Navinchandra et al have exploited the idea that components of a system often cooperate by reacting to each others' actions. The action-reaction function can be thought of as a collection of influence paths where the action of one component changes the state of another. For example, if the system being defined consisted of a power supply connected to a light bulb, the power supply generating electricity would change the state of the light bulb. In the KFP tool we use the idea of influence paths and their flows to determine when a failure has occurred and to isolate failed components.

Barker-Plummer and Bailin describe a system designed to perform theorem proving from graphic descriptions of proofs. Their system, called GROVER, analyzes an image to generate a set of assertions and develop a proof strategy for solving the theorem described in the image. The proof strategy is represented as a set of lemmas that GROVER builds from the assertions in the image. These lemmas are then fed to a "conventional" theorem prover to provide the complete proof.

In KFP we use the concept of assertions, both derived from components in the image and from domain specific knowledge captured in component libraries, to determine when a component is in a state other than those in its definition. When this situation has been detected, a fault has occurred. This observation aids the KFP tool in isolating a fault and beginning the fault recovery process.

In the remainder of this paper we describe the current system and its state. At present a working prototype exists that can be used to generate FDIR knowledge bases. The generated knowledge bases have a text based user interface. The KFP tool demonstrates approximately a 10 to 1 expansion factor for lines of code generated vs. components and influence paths entered by the user.

SYSTEM DESCRIPTION

The original goal of the KFP tool was to be able to generate FDIR rules from graphical images and then use those same images as the user interface to the FDIR system. It was noted early in the system's design, however, that the image alone may not provide enough information about the system to support fault isolation or recovery activities. Given this, the goal was changed slightly to allow non-image information, i.e., system and component states, and component influence relationships, to be used in the FDIR system generation and operation. This information can be entered by the user when the system is being defined, or extracted from libraries that describe the behavior of known components.

The tool was designed to solve the FDIR problem as three subproblems, i.e., detection, isolation, and recovery. Each of the solutions generate knowledge base components that, when taken together, perform FDIR. This is the approach often taken by a human programmer developing an FDIR system, so it seems reasonable to use the same approach in an automated system.

As discussed in the Introduction, influences between components of a system are used to isolate a failed component. The fault is detected when an alarm condition occurs. An example of such a condition would be a temperature sensitive object operating outside of its design temperature range. Figure 1 shows a system in which such a fault may occur. In this figure there are five components that make up a subsystem. The lens component is temperature sensitive and will register an alarm when its sensor reads above or below defined thresholds. An alarm is specified as a collection of component states. In this example the only component involved in the alarm condition is the lens itself; however, in a more complex system one might also need to check other components, such as the quality of communication signals being received, before it is known that an alarm condition exists. In Figure 1, the temperature driver controls the temperature of the lens by turning on and off the heater and cooler as needed.

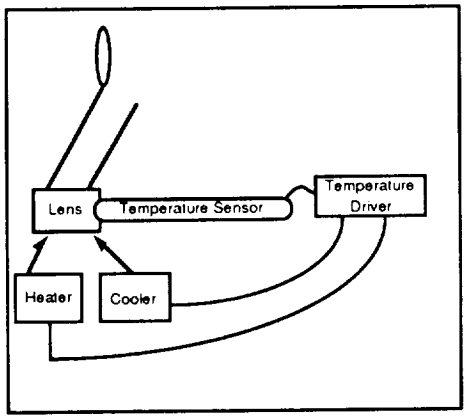

Figure 1: Example System

The cause of an alarm could be one of many failed components, and KFP uses the influence flows among components as well as their known behavior states to identify the component that has suffered a fault. The behavior states of a component describe how it will act when it experiences a particular set of influences. For example, in the system shown in Figure 1, the heater could have the behavior of producing heat as an output influence when it is experiencing the input influences of *power* and *heater on*. Behaviors are represented as component operation states. This implies that the same input influence combination may produce different output influences depending on the current state of the component. For example, when the heater component is off and it receives a *heater on* influence from the temperature driver, it begins to produce heat. If the heater is on already, however, and it receives a *heater on* influence, it does nothing in response. This allows an intelligent component to receive an input and remember the state that it has entered as a result, even after the influence has been removed.

In addition to states, objects can also have local variables. These variables can be manipulated with simple arithmetic operations and can be considered along with the current incoming influence flows to determine when a state change should occur.

Each alarm condition is represented by a CLIPS (C Language Integrated Production System - a NASA-developed inference engine) rule that uses facts about the state of the components contributing to an alarm to determine whether the condition exists. When an alarm is detected, a search begins for the faulted object causing the alarm. This search is performed by tracing back through the paths of influence that are input to the alarming object. The influence paths form a collection of chains of objects that either directly or indirectly influence the components contributing to the alarm. The tracing is performed via a collection of rules that examine the objects in each path. When these rules fire they use information about the current state of the object being examined, and the states of the objects that influence it, to determine whether it is behaving correctly. If the object being examined is not in the correct state then the fault has been isolated. If the object is in the correct state, the objects that influence it are examined next.

After a fault has been detected and isolated, the recovery phase begins. At present the recovery phase consists of notifying the operator and allowing him or her to take corrective action.

COMPONENTS OF THE KFP SYSTEM

There are three main components to the KFP system as shown in Figure 2, object libraries, the system analyst, and the KFP software. Object libraries contain behavior descriptions and associated influence information for known objects. The system analyst extracts objects from the libraries or defines new to describe the complete system to be monitored. The system analyst is responsible for identifying the influence paths among objects and indicating what influence types flow across those paths. Additionally, the system analyst describes the conditions that should cause alarms. The KFP software analyzes the information provided by the system analyst and generates an FDIR expert system. The remainder of this section describes the user interface and operation of the KFP software.

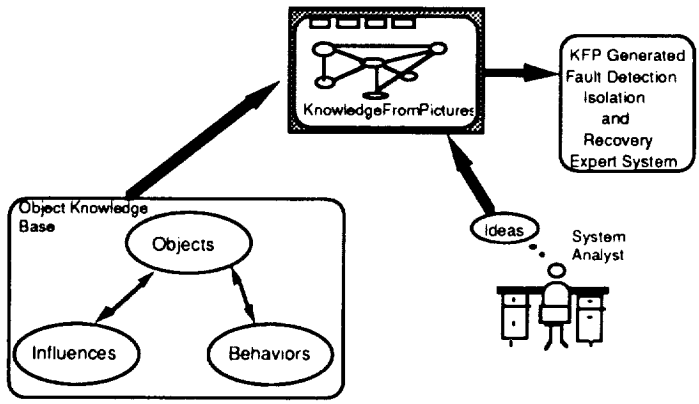

Figure 2: Components of the KFP System

Figure 3 shows the KFP main screen with an example system defined. The components of the system, shown as boxes, are *Temp Sensor*, *Lens*, *Temp Driver*, *Heater*, and *Cooler*. These components are connected via influence paths, shown as lines in the figure. The top part pf the screen contains a collection of pull down menus. The **Files** menu is used to load and save system descriptions, clear the current work space, generate FDIR systems, and quit the KFP system.

The **Objects** menu is used to add and remove objects from the system being designed. The analyst can select a known object from a library, define a new object, or remove an object from the work space. When defining a new object, the analyst only needs to provide a name initially; the object's behavior states and ports can be defined later with their respective menus.

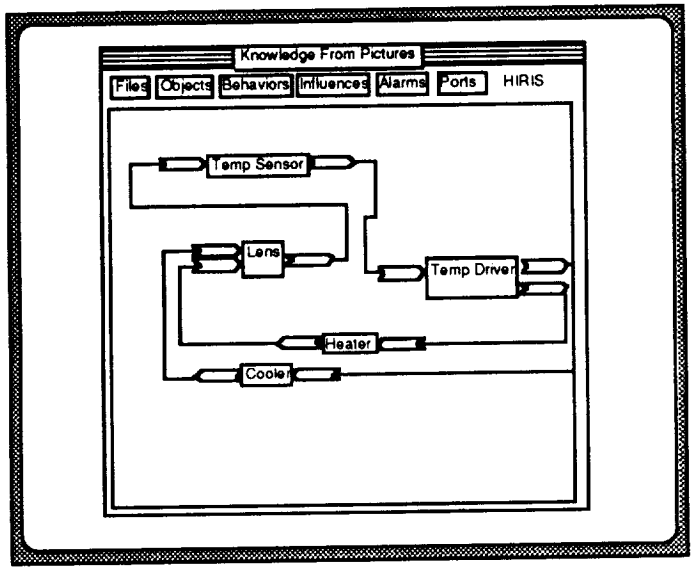

Figure 3: KFP Main Display

The **Behavior** menu is used to add, modify, or remove behavior states of objects in the system. When the analyst selects the *add behaviors* option, the menu shown in Figure 4 is presented The four menus in the figure are used to define the state transitions for an object. The *Present State* is the state that the object is in before the transition occurs. The *Cause* is the variable assignment or input influence that causes a state transition to occur. The *New State* is the state to which the object transitions, and the *Result* is the variable assignment or influence type that the object will exhibit after the transition.

Under each of these menus there is a button labeled New that is used to define a new state, influence type. Under the Cause and Result menus there is a second button used to define any variable assignment used as part of the transition. When the Exit button is pressed the behavior definition is added to the object's description, and is available to KFP when generating the FDIR expert system.

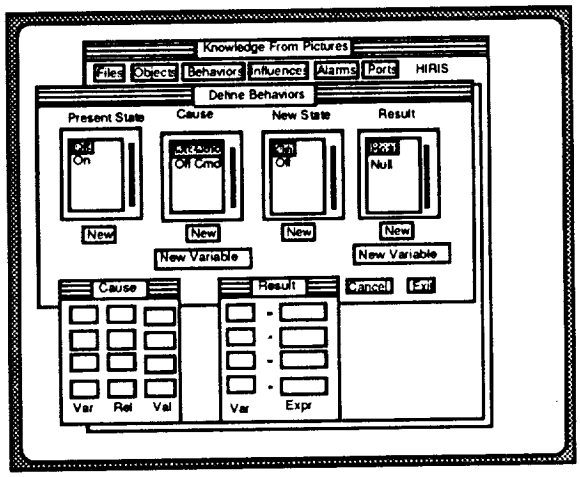

Figure 4: Behavior Definition

If the edit option is selected, the analyst is presented with a list of existing behaviors which can be selectively deleted. At present there is no way to modify an existing behavior other than to remove it and add a new behavior with the modifications.

The two small panel displays will be shown when the analyst selects the New Variable button for either the Cause or Result menus.

The **Influences** menu is used to define and remove influence paths among component ports. When an analyst selects *add influences* from this menu, the originator and receiver object ports for the influence flow can be selected by mouse clicks. The data type of the flow is determined by the connected ports. If the types of the ports differ, the user is notified and the flow is removed.

The influence path and type are then available to be used when generating the FDIR expert system. When an analyst selects *remove* from the **Influences** menu, a list of existing influences paths is presented allowing selective deletion by the analyst.

The next menu available on the main display is the **Alarms** menu. This menu allows the analyst to define alarm conditions for objects in the system. The analyst selects *add* from the **Alarms** menu and selects the object to which the alarm is attached. The display shown in Figure 5 is then presented, and the analyst can select the objects, states, and relationships that contribute to the alarm condition.

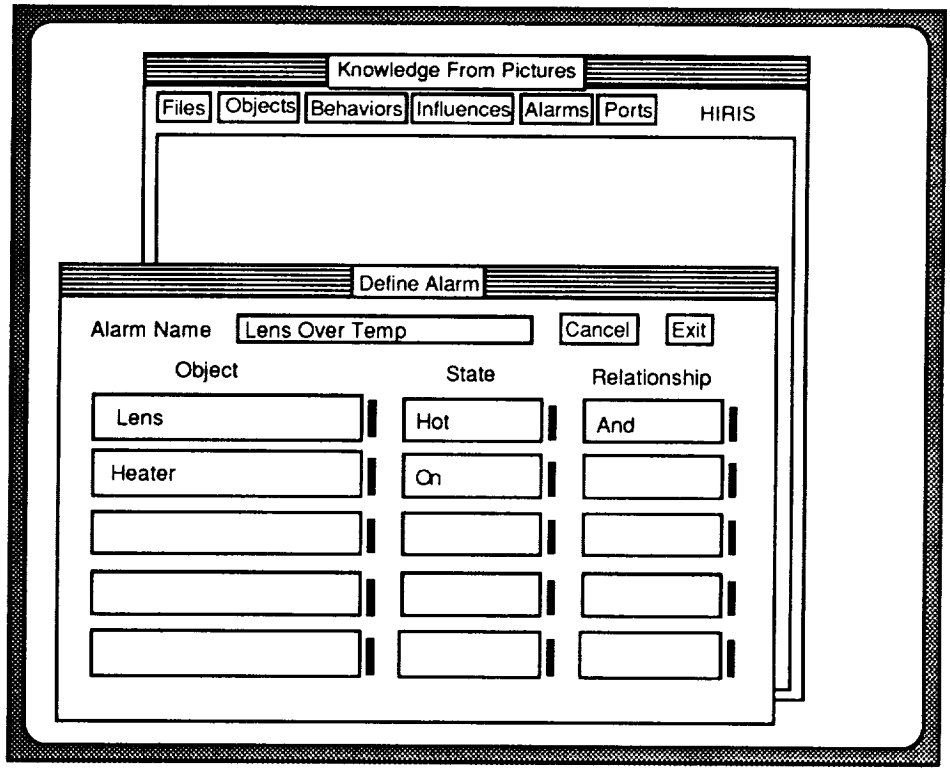

Figure 5: Alarm Definition

The analyst also specifies the name of the alarm at this time. All of this information is used when performing the FDIR task. As with the previous menus, there is an option to remove existing alarm definitions. When this option is selected, the analyst is presented with a list of the names of existing alarm conditions and can select the ones to be deleted.

The last menu available is the ports menu. This menu is used to add incoming or out going ports to an object. To add a port, the analyst first selects the port direction from the menu and then clicks on the object that is to have the

new port. After the object has been selected a menu like that shown in Figure 6 is display, and the analyst can select or define the data type of the port.

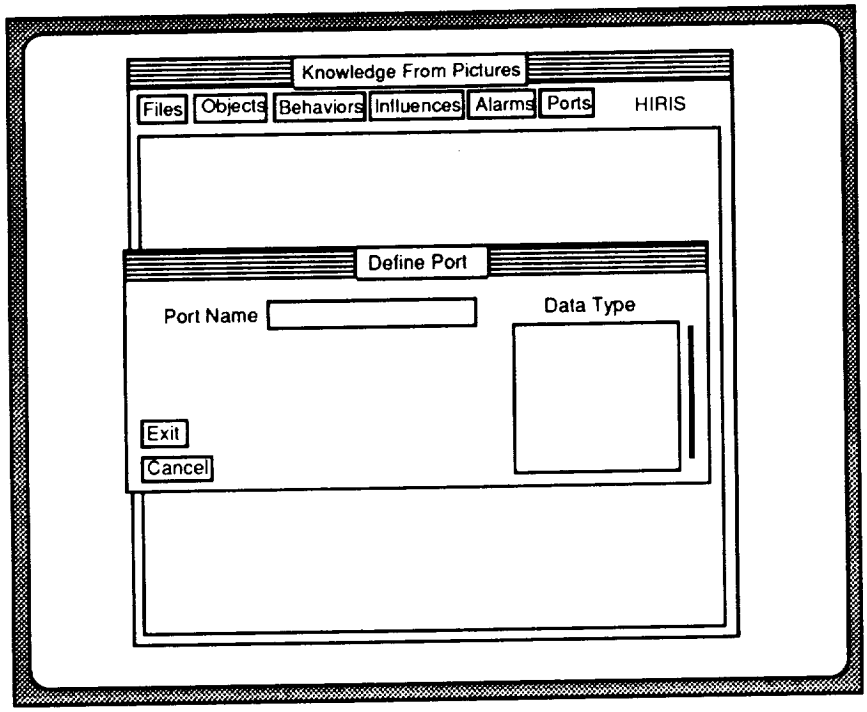

Figure 6: Port Defnition

After adding all the components, behaviors, influences, alarms, and ports that are needed to define a system, the analyst selects the *generate* option from the **Files** menu to generate an FDIR expert system.

SAMPLE OUTPUT RULES

This section shows a sample of the rules and facts generated by the KFP tool. Separate rules are generated for each of the tasks: detection, isolation, and recovery. Facts are generated to describe the influence paths and behavior states of the system. Additionally, rules are generated to update local variable values when state changes occur. Each of these except the recovery and variable update rules is shown in this section. The recovery rules simply print out advice extracted from a library once the fault has been isolated.

```
;;; Fault dection rules

(defrule fault-detection-lens-over-temp
;;; Rule to detect over-temp for lens
(declare (salience 100))
;;; get the current telemetry reading for this lens
(telemetry-status lens temp ?telemetry-level)
;;; is the alarm condition present?
(test (> telemetry-level 270)
==>
;;; notify the operator
(printout t "*** Fault detected in lens" crlf)
(printout t "*** over-temp" crlf)
(printout t "*** Attempting to isolate" crlf)
;;; get information about who could cause this alarm,
;;; and post facts that start the isolation process
(assert (fault-unresolved lens over-temp temp)))
(influenced-by lens ?by temp ?)
(assert (check by lens temp))
)
```

Figure 7: Sample Fault Detection Rules

Figure 7 shows a sample fault detection rule, in which the present telemetry for the temperature of a lens is obtained and checked against a known threshold. If the threshold is exceeded, facts are asserted to begin the search for a failed component.

```
;;; Fault Isolation rules

(defrule fault-isolation-heater-1
(declare (salience 90))
;;; is an alarm present
?notification = (fault-unresolved ?fault-object ?alarm-name ?inf-type)
;;; If so, should I be checked for causing it?
?check-me=(check heater fault-object inf-type)
;;; get the current state to see if heater is influencing mp now
(state heater ?cur-state)
;;; get the current telemetry so that behavior can be determined
(telemetry-flows heater p_temp ?telemetry-level)
;;; Get the behavior information for the heater
(behavior heater curstate ?cur-behavior)
(test (~= telemetry-level cur-behavior)
==>
;;; notify the operator
(printout t "*** Fault isolated in heater" crlf)
(printout 1 "*** Developing a recovery plan" crlf)
;;; removed the notice and the 'check-me' fact
(retract notification)
(retract check-me)
;;; post the fix-it fact
(assert(fix-it heater p_temp cur-state))
```

Figure 8: Sample Fault Isolation Rules

Figure 8 shows one of two rules generated to isolate a failed object. In this rule an object is check to determine if it is in the correct state for its input influences. If it is not, then the fault has been isolated to that object. If it is in the correct state, a second rule would fire that would assert the name of the next object to be examined.

Figure 9 shows a sample of the influence facts that are used to determine which objects are in an influence path. These facts are used by both the fault detection and the isolation rules.

```
;;; (influenced-by <from> <from's-state> <to> <influence-type>
(influenced-by sensor lens hot hot)
(influenced-by sensor lens cold cold)
(influenced-by driver sensor hot hot)
(influenced-by driver sensor cold cold)
(influenced-by heater driver on cold)
(influenced-by heater driver off hot)
(influenced-by cooler driver off cold)
(influenced-by cooler driver on hot)
(influenced-by lens heater hot on)
(influenced-by lens cooler cold on)
```

Figure 9: Sample Influence Facts

GRAPHICAL MODELLING AS A BASIS FOR SOFTWARE DEVELOPMENT

Simulation and diagnostics play a key role in a satellite control center. They support the two principal activities of the control center—commanding and monitoring the spacecraft. Commanding employs simulation to verify the

acceptability of commands, prior to their being uplinked to the spacecraft. Monitoring involves fault detection, isolation, and recovery when telemetry values received from the spacecraft fall outside of defined limits. In our work implementing a testbed for an advanced control center, which we call the Intelligent Ground System (IGS), we found that simulation and diagnosis activities tend to derive from the same set of knowledge, namely models of the spacecraft components. An integrated approach, in which diagnosis and simulation are both driven by the same run-time models, seems feasible to us; at this point, however, we are aiming at a less ambitious goal, which is the generation of distinct programs to support the respective functions from the same graphical model. We can view this as "design time integration" rather than "run time integration."

Our main hypothesis is that modeling of a spacecraft and its subsystems, and reasoning about such models, can—and should—form the key activities of ground system software development; and that by using such models as inputs, the generation of code to perform various functions (such as simulation and diagnostics of spacecraft components) can be automated. Moreover, we contend that automation can provide significant support for reasoning about the software system at the diagram level.

The software models the states, behaviors, and interactions of elements in its environment. Given this role for the software, it seems appropriate to look for a language in which such information can be made explicit. Graphical modeling of objects, their behaviors, and their interactions is an obvious choice for such a language; there is nothing new in our advocacy of diagrams to express such information. Our contention, which may be more questionable, is that the real complexity of the software lies in the interactions expressed by the graphical models, not in the implementation details of the eventual code.

We contend that the structure of the implemented code, for at least certain functions of the ground system—specifically, simulation and diagnosis—is sufficiently well understood to permit us to generate it automatically, and therefore to allow us to redefine the development process as one of developing and reasoning about the graphical models. The previous sections described the progress we have made to date in demonstrating this idea. Similar ideas have been put forward in a recent article by Harel (2).

REASONING ABOUT DIAGRAMS

We have been working for several years on an automated reasoning system that takes diagrams as input. Recently we have begun to apply these ideas to the problem of reasoning about software. The graphical models that we discussed in the previous sections are interpreted by the Formal Interconnection Analysis Tool (FIAT) as plans for proving assertions about the software design.

The particular type of assertions processed by this tool grew out of an actual experience in debugging part of our ground system testbed. In testing a particular simulator program it was found that the behavior of the system was not as expected, but no errors could be found in any of the simulator components. The problem turned out to be one of missing connections between objects in the simulator. Since the simulator architecture keeps each object autonomous—completely ignorant of the objects to which it is connected in a given application—the absence of these connections did not result in any anomalous behavior on the part of any object, but the system itself was not behaving as expected.

Thus we decided to apply the planning concept to verifying statements of the form, "If event x occurs at object A then event y will occur at object B." The planner takes event y at B as a goal, and tries to construct a plan that starts from event x at A as an initial condition (typically, various other context conditions are specified as well). A goal is reduced to subgoals by traversing the connections specified in the diagram: if a goal state in an object D follows, according to D's behavior description and the connections specified in the diagram, from a certain state in object C, then this state in object C becomes a subgoal of the goal state. A failed plan, when presented to the developer, serves to identify missing connections that may have been overlooked in defining the system.

We have noticed a similarity in the logic of this planner and that of the KFP tool, which similarly traces back through the influence paths in the diagram in generating fault isolation rules. We have not studied this similarity in enough detail to decide whether the two tools can make use of a single "influence traverser" mechanism, but there seems to be some promise of this.

NEXT STEPS AND FUTURE IMPROVEMENTS

The tool is currently a working prototype that achieves most of the task goals. There are some issues that still need to be considered, and extensions that would add to the system's power. At present the system provides only a text based interface for the generated FDIR system. Providing a graphical user interface, by reusing the analyst defined picture, is the next function that we will address in this prototype. After this is completed, all of our original goals will have been met. The remainder of this section lists the enhancements that will be added to the existing system to make it more widely usable.

Currently there are very few objects in the object libraries. Populating this library would make the generation of new system description images simpler because predefined objects could be reused.

Once a system has been defined, it might be advantageous to use the complete system as a component of a larger system. For example, a power system FDIR expert system may be usable for more than one spacecraft. Such complete systems could be represented as *smart icons* on the work space, and could be used just as if they were simple components.

There may be some refinements to the heuristics used to analyze the graph and generate an expert system, as well as those employed in the generated expert system itself. We intend to explore this issue in order to increase the quality and performance of the generated code.

The user interface for the current prototype was developed in TAE and is easy to use. There are some refinements, however, that would present more information to the user and would result in faster operation of the tool. For example, currently to display all the influence paths between two objects, the tool must generate an additional window and menu. It would be faster to display this information in the work space when the user clicks on an influence path line. The enhancements that we currently envision can be implemented in TAE.

Finally, we are concerned about KFP's ability to handle very complex system pictures. The current expansion factor of objects, behaviors, and influence paths to generated code is approximately 10 to 1. We believe that the expansion is linear, but it still may be too large for very complex systems. We intend like to address this issue by working with additional, more complex examples.

KFP AS A SOFTWARE ENGINEERING PARADIGM BASIS

We have made a start at what we hope will become an integrated graphical modeling and development system, in which software development becomes synonymous with defining and reasoning about graphical models. The prospects for such an integrated environment are based on a few empirically perceived similarities:

- Similarity between the information used to simulate a system and that used to diagnose faults
- Similarity between the logic used to reason about system behavior during development, and that used to diagnose faults during operation (backward chaining over influence paths)
- Similarity in the program structure of specific simulators and specific diagnostic systems, which has allowed us to define generic architectures for each of these applications

Within the scope of the current framework, there are perhaps two major open issues: 1) the impact of scale-up on the performance of the generated code, and 2) the feasibility of automated reasoning about additional aspects of the models.

The efficiency of the generated fault detection, isolation, and recovery rules for a large, complex system is an open issue. The examples we have worked with to date in KFP have been obtained from actual systems (either existing or being developed), but they are very small subsets of these systems. There is a solid basis of real-time scheduling theory (e.g., rate-monotonic scheduling) with which we can address scale-up performance issues for the generated simulator code, but we lack such a firm basis for a rule-based diagnostic system. The solution to this problem may be to evolve to a more thoroughly model-based approach to diagnosis, in which there is no production rule

interpreter at all. This would, in addition, permit a greater degree of integration between the diagnostic and the simulator code.

An open issue concerning reasoning about the models is whether automation can support reasoning about issues other than the pre-condition/post-condition behaviors currently addressed. One major area that we would like to investigate is support for reducing the state space of a set of interacting components. This problem arises in "reachability analysis," in which one tries to prove (or at least to convince oneself) that no unexpected states are entered. In the area of communications protocols, this has proven to be a difficult but necessary process that can be supported by a variety of heuristic techniques, some of which are automated (3,4)

POTENTIAL COMMERCIAL APPLICATIONS

Certainly the development of new software and knowledge engineering paradigms based on graphical reasoning would have great commercial value. But there are many other potential applications of the technology discussed in this paper. Consider the application of the KFP in the field of Computer Aided Design (CAD), a field strongly related to the KFP work. Intelligent CAD (ICAD) systems exist which are indispensable in the development of many products in various industries. However, at the Eurographics Workshop on Intelligent CAD Systems (held in April, 1988 at Koningshof Congress Centre, Veldhoven, The Netherlands) several important implementation issues were raised and discussed. These included: the definition of design objects, the overall object design process, linguistic issues associated with representing design objects and their inter-object interactions, the relationship between object-oriented and logical paradigms, and the need for increasing the intelligence of ICAD systems. The KFP project is addressing these issues. Though the current focus of the KFP work is on expert system rule-base development the relevance of the emerging KFP concepts and approaches to advanced ICAD systems are obvious.

Another longer-term application of the KFP concepts could be in the area of highly intelligent robots. The graphical reasoning techniques being investigated in the KFP system could prove to be extremely appropriate for advances in the evolution of robotic vision and reasoning capabilities.

REFERENCES

1. Barker-Plummer, D. and Bailin S. "Graphical Theorem Proving" To be submitted to the Journal of Automated Reasoning.

2. Harel, D., 1992. Biting the silver bullet: Toward a brighter future for system development. IEEE Computer, January 1992.

3. Holzman, G., 1992. Protocol design: redefining the state of the art. IEEE Software, January 1992.

4. Lin, F. and Liu M., 1992. Protocol validation for large-scale applications. IEEE Software, January 1992.

5. Montalvo, F. "Diagram Understanding: Associating Symbolic Descriptions with Images." IEEE Computer Society Workshop on Visual Languages, held in Dallas, TX on June 25-27, 1986. IEEE Computer Society Press, pp. 4-11.

6. Musen, M., Fagan, L., Shortliffe, E. "Graphical Specification of Procedural Knowledge for an Expert System." IEEE Computer Society Workshop on Visual Languages, held in Dallas, TX on June 25-27, 1986. IEEE Computer Society Press, pp. 167-178.

7. Navinchandra, D., Sycara, K., and Narasimhan, S., "A Transformational Approach to Case Base Synthesis", Journal of AI in Engineering Design and Manufacturing, Vol. 5, #2, 1991.