Model-Based Reasoning for System and Software Engineering The Knowledge From Pictures (KFP) Environment

Sidney Bailin, Frank Paterra, and Scott Henderson
CTA Incorporated*

Walt Truszkowski**
NASA/Goddard Space Flight Center

1. Introduction

This paper presents a discussion of current work in the area of graphical modeling and model-based reasoning being undertaken by the Automation Technology Section, Code 522.3, at Goddard. The work was initially motivated by the growing realization that the knowledge acquisition process was a major bottleneck in the generation of fault detection, isolation, and repair (FDIR) systems for application in automated Mission Operations. As with most research activities this work started out with a simple objective: to develop a proof-of-concept system demonstrating that a draft rule-base for a FDIR system could be automatically realized by reasoning from a graphical representation of the system to be monitored. This work was called Knowledge From Pictures (KFP) (Truszkowski *et. al.* 1992). As the work has successfully progressed the KFP tool has become an environment populated by a set of tools that support a more comprehensive approach to model-based reasoning. This paper continues by giving an overview of the graphical modeling objectives of the work, describing the three tools that now populate the KFP environment, briefly presenting a discussion of related work in the field, and by indicating future directions for the KFP environment.

2 Graphical Modeling as a Basis for Answering Questions: KFP Concept

By way of introducing the major concepts in the current KFP environment we describe an approach to modeling a system that allows one to perform the following functions:

- Verify the correctness of a system design
- Simulate the behavior of a system
- Monitor the behavior of a system

Each of these functions amounts to answering certain questions about the system being modeled. We therefore view verification, simulation, and monitoring as different forms of *querying a system model.* In its current state of development, our models can be used to answer the following types of questions:

1) Under what conditions will event **E** or state **S** occur? This can be asked at design time for verification, or at run-time for explaining an observed event **E** or state **S** (monitoring).

* Mailing address: CTA Incorporated, 6116 Executive Boulevard, Suite 800, Rockville, MD 20852. E-mail: sbailin@cta.com, fpaterra@cta.com, scott@cta.com.

** Mailing address: NASA/Goddard Space Flight Center, Code 522, Greenbelt Road, Greenbelt, MD 20771. E-mail: wtruszkowski.520@postman.gsfc.nasa.gov.

2) What will occur as a consequence of state S and/or event E? This can be asked at design time for verification, or during system testing (simulation).

3) Will state S_2 occur as a consequence of state S_1 and event E? This can be asked at design time for verification, during system test (simulation), or at run-time to explain the observation of state S_2 (monitoring).

2.1 The Graphical Language

The modeling language represents a system as a set of components that are connected together via input and output ports. Each component has a set of output ports, which transmit information or physical resources (e.g., heat, power) to other components; and a set of input ports, which receive such resources from the output ports of other components. In addition, each component has an internal state, which is represented through one or more stores or variables. A component may also contain sub-components, which in turn are connected with each other. Thus, there is no conceptual difference between a system and a component: a component is a system consisting of its sub-components, and any system may be used as a component within a larger system.

For example, Figure 1 shows a model of the temperature control subsystem of a spacecraft instrument. The purpose of this subsystem is to control the temperature of the lens. Heat provided to the lens influences the temperature sensor, which in turn sends a digital temperature signal to the temperature driver. The function of the temperature driver is to turn the heater and cooler on and off as needed. The heater and cooler in turn influence the lens by passing or reducing heat, respectively.

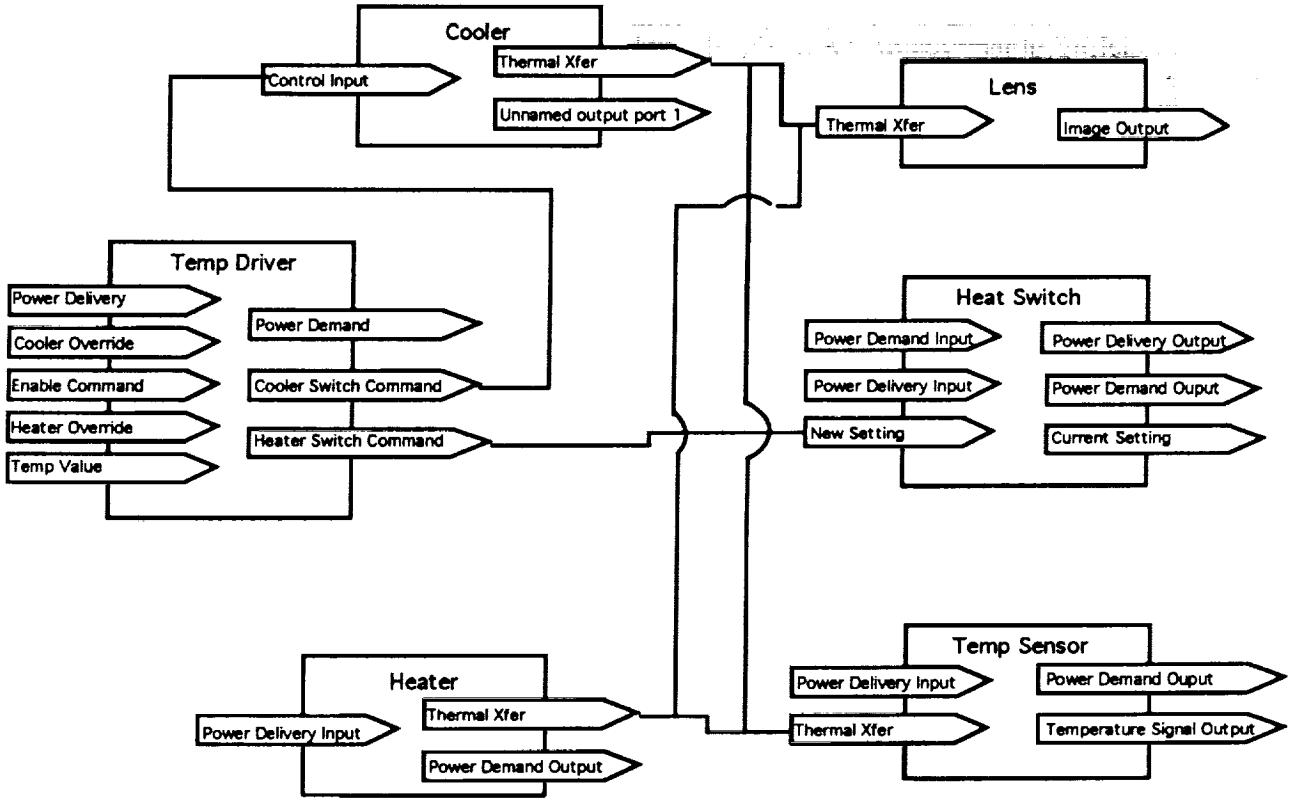

Figure 1: Temperature Control Subsystem Model

Each component in the model has a behavior description that specifies how its internal state and outputs change in accordance with changing input (and its previous state). To describe behaviors we use a tabular representation similar to that advocated by Parnas *et al* (1990). The tabular form allows us to accommodate both continuous functions and discontinuities. As Parnas observed, discontinuities are the major problem in specifying system behavior as a mathematical function—functional expressions in mathematics are typically continuous. At the other end of the continuous/discontinuous spectrum, a finite state machine is well-suited to describing a discrete set of behaviors, but is not suited to specifying new states and outputs as a continuous function of inputs and previous state.

The tabular representation is a blend of these two approaches. Each row in the table represents a nominalized, or abstracted, state, within which the system's behavior may be described as a continuous function of the current input and the specific (non-abstracted) state. Different rows in the table correspond to different abstracted states, in which the behavior is characterized by different functions. For example, the following table specifies the behavior of the temperature driver in the model shown in Figure 1:

Table 1: Behavior of Temperature Driver

Present State	Influence	Next State	Action
Idle	T > MaxTemp	CoolerOn	Set Cooler Signal
Idle	T < MinTemp	HeaterOn	Set Heater Signal
CoolerOn	T <= MaxTemp-Delta	Idle	Drop Cooler Signal
HeaterOn	T >= MinTemp+Delta	Idle	Drop Heater Signal

In this case the next-state and action functions are discrete-valued and are constant within each row of the table. The input port of the temperature driver is, of course, real-valued, but in the behavior table it is described in terms of three abstracted states:

T > MaxTemp, T < MinTemp, and (implicitly) MinTemp <= T <= MaxTemp

2.2 Querying the Models

Let us see how such models can be used to answer the types of questions posed above:

*Under what conditions will event **E** or state **S** occur?* To answer this we need to perform backward chaining through the state transitions and connections described in the model. We begin with the "fact" (whether hypothetical or observed) that event **E** or state **S** has occurred. This is treated as a goal in our backward chaining search. Typically **E** or **S** will describe the internal states and/or outputs of one or more components. We therefore look for state transitions in these components that would result in **E** or **S**. Each of these transitions will be predicated on a previously occurring internal state and input event. These previous states and input events therefore become subgoals. Input events of one component translate into output events of another component via the connections specified in the model. Similarly, the internal states that were pre-conditions of **E** or **S** become subgoals; they can be established by either assuming them as initial conditions, or tracing them back via still earlier transitions to previous states.

*What will occur as a consequence of state **S** and/or event **E**?* Answering this requires that we perform forward chaining through the state transitions and connections described in the model. We being with the "fact" that the system is in state **S** and/or that event **E** has just occurred, and we proceed to execute the state transitions that occur as a result (as specified

in the behavior description of the model). Values that occur at output ports must be propagated over to whatever input ports they are connected to, and subsequent state transitions must then be carried out. Forward chaining therefore amounts to "executing" the model.

Will state S_2 occur as a consequence of state S_1 and event E? This type of question can be addressed by either forward or backward chaining, and in both cases there is a possibility of inconclusive results. We can execute the model starting with the occurrence of event **E** in state S_1, and check for the system entering into state S_2. If there is feedback in the model—i.e., there is a cyclical connection between some components C_1 -> C_2 ... -> C_k -> C_1—it may not be possible to limit the execution time within which S_2 must occur. We can, alternatively, treat S_2 as a goal in backward chaining, as in the first type of query—but with the additional constraint that the initial conditions arrived at must be consistent with S_1 and **E**. Here too, if there is feedback in the system, there is a possibility of infinite regression. We can artificially limit such searches by placing a bound on the number of transitions executed and the number of connections traversed. If the execution of transitions and the flow of resources over connections are viewed as taking time, rather than being instantaneous, then such a bound corresponds to a time limit within which S_2 is required to occur.

3 Tools for Querying the Models - the KFP Environment

In the KFP environment we have developed tools to provide answers to each of the types of questions posed above. The *Formal Interconnection Analysis Tool* (FIAT) is intended to be used for verifying designs. It performs backward chaining to answer questions (1) and (3) at design time. *The Multiple Aspect Simulation Tool* (MAST) executes the models. MAST can be viewed as a tool for simulating a model or, depending on the context, for implementing the model as a software system. The Diagnostics Inferred from Graphics (DIG) tool generates rules that backward chain through the model at run-time. DIG is intended to be used for system monitoring. In this section we describe the user interface through which the models are specified, and then show how FIAT, MAST, and DIG process the models for their respective purposes.

3.1 The User Interface

Figure 2 shows the main selection panel of the environment. Operations for managing the model library are provided within the Load and Save Libraries menu.. The Edit Components menu brings up the editor with the selected component displayed, or with a work space for creating of a new model.

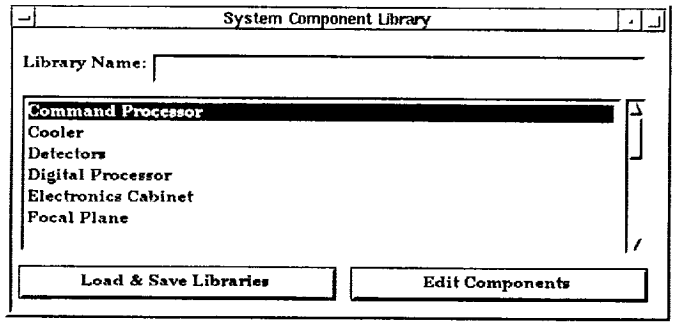

Figure 2: KFP Main Selection Panel

Figure 3 shows the model editor with an example system defined. The components of the system, shown as boxes, are *CmdProcessor*, *DigitalProcessor*, *Heat Relay*, *Heater*, *Cooler*, *PowerSupply*, and *Sensor*. Each component has ports, shown as large arrows. Connections between ports are displayed as arcs. As shown in the figure, a menu containing commands for editing and querying the model pops up in response to a middle-button mouse click in any "white space" area of the diagram:

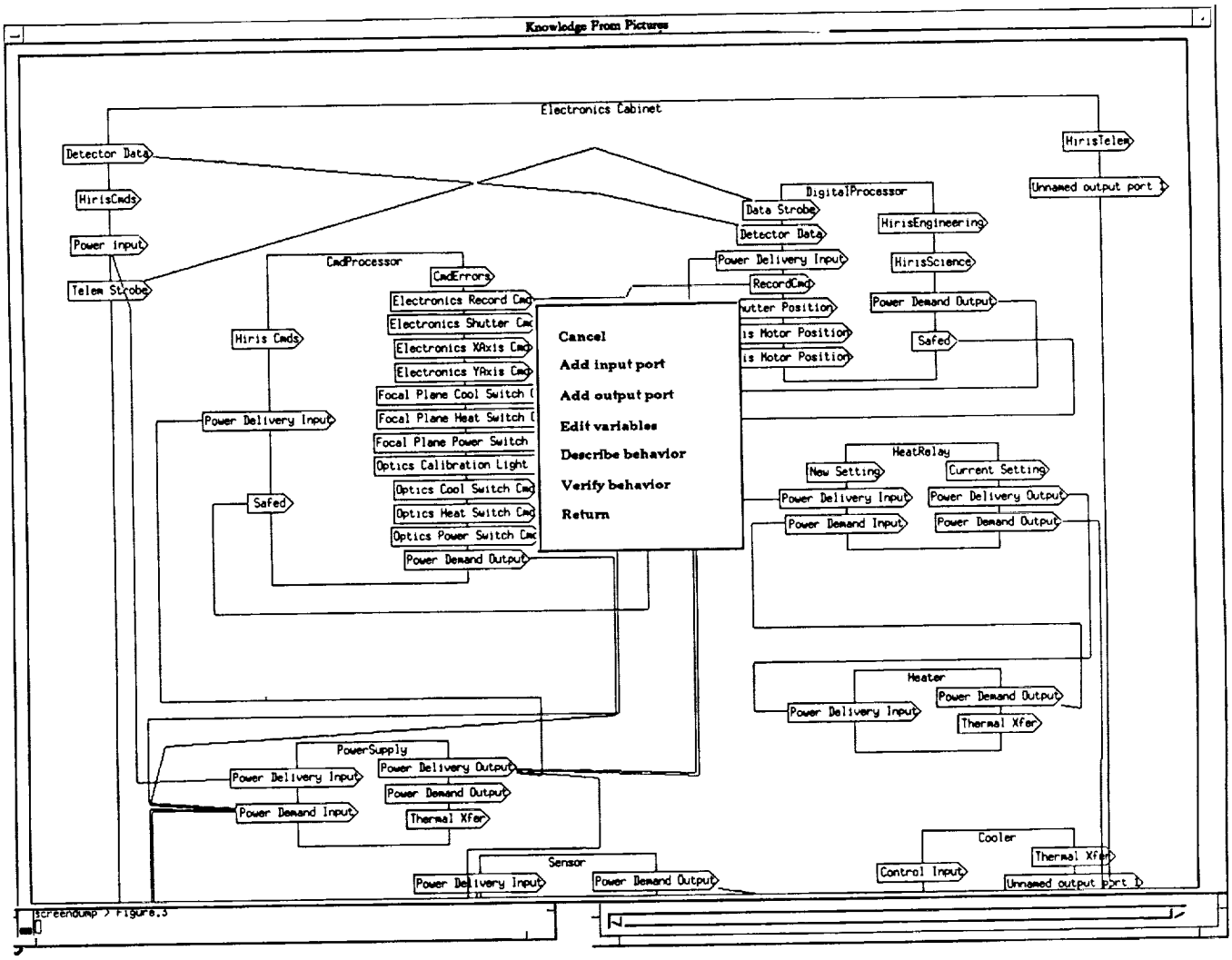

Figure 3: KFP Model Editor

The **Describe Behavior** selection is used to add, modify, or remove behavior states and transitions of the currently displayed system. As shown in Figure 4, the **Mappings** panel displays the state transitions that are currently defined. The **Transition** panel enables the user to modify an existing transition description, or to create a new one.

The *Starting State* is the state that the object is in before the transition occurs. The *Trigger* is the variable assignment or input influence that causes a state transition to occur. The *Ending State* is the state to which the object transitions. The **Add** buttons under *Starting State* and *Ending State* are used to add conjunctive conditions to these states' definitions. When the **OK** button is pressed, the behavior definition is added to the object's description, and is available to the specific tools that are used to query the model.

After adding all the components, behaviors, connections, and ports that are needed to define a system, the analyst selects the appropriate menu option to invoke one of the specific tools described below. For example, **Verify Behavior** (in Figure 3) invokes the Formal Interconnection Analysis Tool.

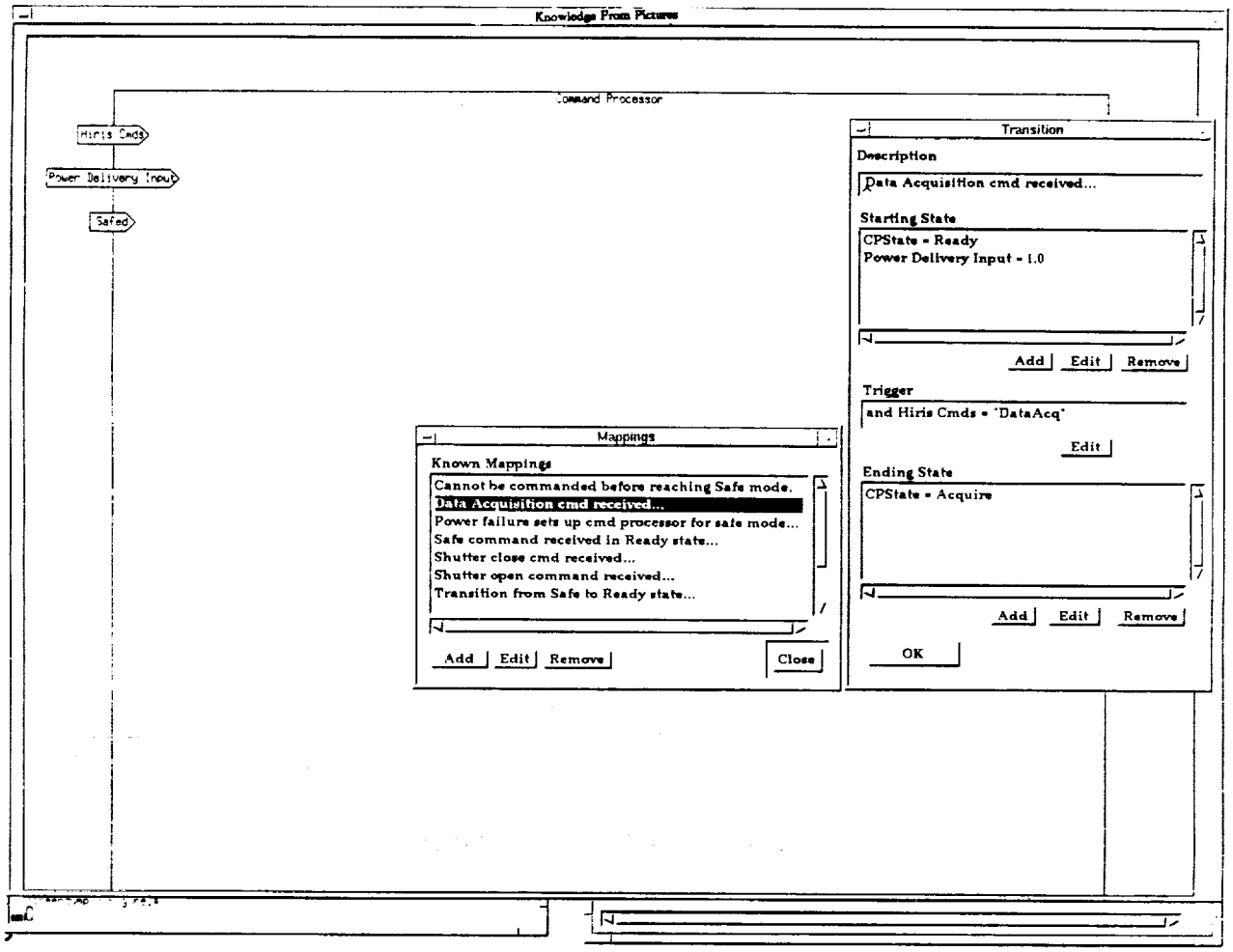

Figure 4: Component Behavior Definition

3.2 The Formal Interconnection Analysis Tool

FIAT uses a planning algorithm, implemented in Prolog, to chart a path from a (partially) specified initial state to a specified end state. Steps in the path are either the transfer of a value along a connection between ports, or the execution of a transition within a component. The planning algorithm works backwards from the specified end-state until it arrives at conditions that are specified in the initial state, or are consistent with the specified initial state.

FIAT is invoked by stating a goal to the planner. The planner then determines how to arrive at a situation in which this goal is true. A typical goal is of the form

<time-tag>: <goal-condition>

indicating that at the time designated by <time-tag>, the condition <goal-condition> is true. The time tag can be a numeric expression, a symbolic expression (e.g., containing a variable *t*), or one of the keywords START, END. A typical goal condition is **A.B.C.D** = *V*, indicating that the variable/port **D** within the component **A.B.C** has the value *V*.

Backchaining algorithm. The planner responds to a goal in one of the following ways, which are listed in order of priority:

- Finds a way to show that the goal is established. For example, a goal of the form START: <goal-condition> is established if <goal-conditrion> is implied by the user-specified initial conditions. A goal of the form T: <goal-condition> is established if the planner can show that <goal-condition> is implied by the user-specified initial conditions *and* that it is not affected by the user-specified initial event.
- Tries backchaining to create one or more subgoals. Backchaining takes one of two forms, either through a connection or through a state transition, depending on whether the goal refers to an input port, a state variable, or an output port.
- Adopts the goal as an additional assumption of the plan. The goal condition must be *consistent with* (though not necessarily implied by) the user-specified initial conditions.

Synchronizing and checking consistency of subplans. FIAT processes a list of subgoals by achieving each subgoal independently. This is not sufficient in general, since the subplans may interact. Moreover, the subplans may be of different length, requiring that they be synchronized with each other. In a general planning context, checking the consistency of an arbitrary set of subplans can be computationally intensive, since one must consider arbitrary interleavings of the individual steps of the plans. In our domain, however, it is not necessary to consider arbitrary interleavings. Instead, we synchronize in one of two ways:

- *Without time.* In this approach, all time tags are of the form START or END. All transitions and propagations of values along connections are assumed to occur instantaneously. This approach can only be used for models without feedback. In such models, given any two components $\mathbf{C_1}$ and $\mathbf{C_2}$, either $\mathbf{C_1}$ is "upstream" from $\mathbf{C_2}$ or *vice versa*. FIAT can therefore assume that no changes occur to a component **C** until all components upstream from **C** have been processed. When viewed recursively, this implies that all upstream components have stabilized in their resulting states by the time **C** undergoes a transition. Thus, each component undergoes at most one state transition, from its initial state (START) to the END state, which results from the influence of its upstream neighbors.

- *With time.* In this approach, every possible state transition and every connection in the model is annotated with a numeric or symbolic delay value, which indicates the length of time consumed by the transition or by propagation over the connection. FIAT uses this information to tag each step of a subplan with its time of occurrence in relation to the time of the ultimate goal.

Once the subplans are synchronized, FIAT can check their consistency by comparing, at each step in the plan, the values of the ports and state variables affected at that step. In general such comparisons are difficult because the values may be symbolic rather than numeric. For example: the value of a state variable **V** of a component **C** at time *t* may be specified, in one subplan, as a polynomial expression E_1 in the current values at the input ports $\mathbf{I_1}$ and $\mathbf{I_2}$ of **C**. In another subplan, the value of **V** at time *t* may be specified as another polynomial expression $\mathbf{E_2}$ in $\mathbf{I_1}$ and $\mathbf{I_2}$. To verify the consistency of these subplans, FIAT must establish the equality of $\mathbf{E_1}$ and $\mathbf{E_2}$. This is a theorem-proving problem and cannot be solved in general. Thus, depending on the complexity of the behavior specifications, the plan returned by FIAT may not be a conclusive proof that the goal state can be reached. Expanding FIAT's theorem-proving power in order to handle complex behavior specifications is an important goal of our research.

3.3 The Multiple Aspect Simulation Tool

The multiple aspect simulation tool (MAST) is used to "execute" the graphical models. We use the term "simulation" because typically, in our environment, the diagrams are used to model physical (electro-mechanical) systems. If, however, the model simply represented the components of a software system, then the resulting MAST code would be an implementation of that system.

MAST is based on a generalization of the connection management approach described in (Lee *et al* 1990). In that approach, communication between components is achieved through the operation of a *connection manager*, which is responsible for visiting each updated output port of each component and propagating its value to the necessary input ports (those to which the output is connected). In our generalization of this approach, each *type* of connection has its own connection manager. Currently MAST contains connection managers for the following types of connections:

- Power
- Thermal
- Digital
- Image

Each of these types of connections requires its own form of processing, e.g., the frequency with which values are updated—hence the use of separate connection managers (and the name "multiple aspect"). Another deviation from Lee *et al* is that the connection managers in MAST are global, i.e., they range over all components in the model. In the original approach, each subsystem, sub-subsystem, etc. has its own connection manager, which handles the connections between objects in that subsystem, sub-subsystem, etc. The use of global managers provides more flexibility in determining the order in which components should be visited.

The major benefit of using connection managers is that each component in the simulation remains independent of all other components. The components influence each other strictly through the flow of information over the connections defined in the graphical model, and these connections are implemented by means of connection managers. This simplifies the

construction of the simulator from the graphical model: all that needs to be done is to generate data descriptions of the components and their connections, and code implementing the behavior (i.e., the state transitions) of each individual component. The connection manager code, which is driven by the component descriptions, remains constant from one model to another and is simply linked in.

The entire simulation is driven by an executive, which is another fixed block of code that is linked together with the connection managers and component definitions.

3.4 The Diagnostics Inferred from Graphics Tool

DIG generates an expert system to monitor the system described by the graphical model. The expert system consists of a set of facts and rules in the C-Language Integrated Production System (CLIPS—see Giarrantano, 1991). The generated rules solve the fault monitoring problem as three subproblems: Detection, Isolation, and Recovery.

In the generated rules, connections 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 showed a system in which such a fault may occur. The lens component is temperature sensitive and will register an alarm when its sensor reads above or below defined thresholds. 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.

The cause of an alarm could be one of many failed components. DIG uses the connections between components as well as their known behavior states to identify the component that has suffered a fault. The values of each component's state variables are considered along with its current inputs to determine if it is operating according to its defined behavior. Both influence and behavior information are represented by CLIPS facts in the generated expert system.

Each alarm condition is represented by a CLIPS rule that uses facts about the state of the components contributing to the 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 two rules generated for each object. The first rule compares the object's current state and inputs to its behavior specification; if these do not match, then the fault is occurring in that object. If the fault is found, a fact is asserted to begin the recovery phase. If no fault is detected, the second rule fires and uses the connection information to identify the next object to be examined. The connection paths form a collection of chains of objects that either directly or indirectly influence the components contributing to the alarm.

After a fault has been detected and isolated, the recovery phase begins. At present the recovery phase consists solely of notifying the operator, who can then take corrective action.

We have recently developed a run-time user interface for the generated expert system, which uses an animated version of the graphical model to display system status to the user. The run-time user interface itself is independent of the monitored system, and works in conjunction with any rule-base generated by DIG (and the corresponding diagram). The animation works as follows: when an alarm occurs, the component to which the alarm is attached is highlighted in red. During the ensuing fault isolation process, components that

"check out" are highlighted in green; a component in which the fault has been isolated is identified by pointing to it with red arrows. The user can therefore follow the fault isolation process by observing the successive highlighting of components in the display. When the state causing the alarm changes to a satisfactory state, the highlights are cleared and the components are restored to their usual display mode.

4 Related Work

Our approach to model-based engineering is closely related to work on executable specifications in software engineering and to model-based diagnostics in artificial intelligence. In this section we briefly review these two research areas in so far as they bear on our work.

4.1 Executable Software Specifications

The trend towards ever higher levels of languages in software engineering has led to the use of diagrams as executable specifications. Numerous tools developed in the research community, and a small number that are commercially available, either interpret diagrams or generate exectuable code on the basis of an implied operational semantics for the diagrams. The syntax and semantics of the diagrams varies widely, from dataflow approaches to state-based representations (see, for example, Zave and Schell, 1986; Jensen, 1987; Wang, 1988; Pulli, 1989). In our work, both MAST and DIG act as code generators that are guided by graphical models.

Harel (1992) makes a point quite close to ours by suggesting that such tools are more than curiosities, or even productivity enhancers. They represent, rather, a significant shift in the level of abstraction at which engineers can, and should, think about software. Two open issues in this shift concern the degree to which diagrams can accurately represent the intended functions of a software system, and the performance levels that can be achieved with automatically generated code. The first issue—semantic richness—depends on the modeling approach used, including the way in which diagrams are interpreted operationally, the amount and kinds of text-based annotations permitted, and (importantly) the domain of applications for which the software is intended. For example, dataflow models are amenable to a wide range of operational interpretations (see, for example, Bewtra *et al*, 1992); the semantics implied by MAST are well suited to simulation systems, but may not be appropriate for systems in which messaging plays a more essential role than dataflow.

The second issue—performance—is one that Harel sees as being progressively addressed as more work is done in the area of executable specifications. The chief use of such tools today is for the execution of functional prototypes of a system; the production system can then be developed with adequate performance by means of more conventional methods. Harel sees this changing, however, as we become more skillful at generating code from high-level models.

4.2 Model-Based Diagnostics

Model-based reasoning has become an important alternative to the conventional fault-based approach to diagnostics which was first demonstrated in the MYCIN system (Hayes-Roth *et al*, 1983). The fault-based approach uses a symptom/explanation structure, typically encoded in rules, to offer possible diagnoses of an observed problem. The limitations of this approach are now well-known: faults must be explicitly encoded in the knowledge base in order to be recognized, there is no sound method of representing uncertainty, and the validity of the knowledge base is difficult to establish.

Model-based reasoning employs a more direct representation of the rules that govern a system's behavior (Struss, 1992). The model-based approach treats the knowledge base as a description of the structure and behavior of the system being analyzed. Valid behavior is then characterized in terms of the states of observable elements of the system, and the relations that must hold between these states. Invalid behavior consists of any violation of these constraints, rather than being characterized by some finite set of faults identified *a priori*.

The model-based approach also admits a theoretically sound representation of uncertainty. The field of causal modeling applies Bayesian probability to the causal relationships between aspects of a system's state (Lemmer and Kyburg, 1992). The distinction between this and the fault-based approach is subtle but important. The fault-based approach draws a direct relationship between sets of symptoms and possible diagnoses. Causal modeling relates partial observations of a system's state to possible extended descriptions of the system's state. The range of possible explanations is much wider than in the fault-based approach, and is less susceptible to the biases that can easily enter unnoticed into a symptom/explanation structure.

Although the DIG tool generates a knowledge base in the form of production rules, the form of reasoning performed by these rules is clearly model-based.[1] There is no explicitly defined fault set—only a description of admissible and inadmissible states of the system. The rules are used to isolate the problem on the basis of the known relationships between components.

The open-endedness of model-based diagnostics has an analog in our approach to model-based engineering. As described in Section 2, we view the process of engineering as a process of creating, querying, and modifying models. In this context, open-endedness means that the questions that can be answered are not limited to some pre-defined set. This is a significant departure from the common practice in software engineering of using "canned" methodologies, which in essence prescribe a certain set of questions to be answered about a system under development. The model-based approach to system and software engineering provides a basis for adapting a method to the needs and constraints of a given project. Adaptation is achieved by tailoring the questions that will be asked about the system models. Of course, changing the questions may require enhancing or otherwise changing the models. By placing the emphasis on querying models, however, our approach encourages a scientific mindset in developing systems, as opposed to a mechanical "cookbook" approach.

5. Future Directions

Currently the KFP environment is a stand-alone set of tools for model-based graphical reasoning. In the coming year this environment will be integrated into a version of the Generic Spacecraft Analyst Assistant (GenSAA) workbench. GenSAA is designed to support rapid development and application of real-time expert systems in the Mission Operations domain. This experimental integration of the two environemnts (KFP and GenSAA) will provide an opportunity to more fully evaluate the anticipated benefits that will be derived from embedding a model-based graphical reasoning capability in a workbench for the real-time development of expert systems.

---

[1] We make this observation because model-based reasoning is often contrasted with rule-based reasoning.

References

Giarrantano, J., 1991. CLIPS Reference Manual. NASA Johnson Space Center, Houston, TX.

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

Hayes-Roth, F., Waterman, D., and Lenat, D., eds., 1983. *Building Expert Systems.* Addison-Wesley Publishing Company.

Jensen, K., 1987. Computer tools for construction, modification, and analysis of Petri nets. *Advances in Petri Nets, Part II,* ed. W. Brauer, W. Reisig, and G. Rozenberg. Lecture Notes in Computer Science, Vol. 255, pages 4-19. Springer-Verlag, New York, NY.

Lee, K. *et. al.,* 1990. An OOD paradigm for flight simulators, 2nd edition. Technical Report of the Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

Lemmer, J. and Kyburg, H., 1992. *An Investigation of Independent Causality as a Basis for Uncertain Prediction and Inference.* Internal memorandum, CTA Incorporated, Rome, NY, November 16, 1992.

Montalvo, F., 1986. Diagram understanding: associating symbolic descriptions with images. *IEEE Computer Society Workshop on Visual Languages*, held in Dallas, TX, June 25-27, 1986. IEEE Computer Society Press, pages 4-11.

Musen, M., Fagan, L., Shortliffe, E., 1986. 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, pages 167-178.

Parnas, D., Asmis, G., and Madey, J., 1990. Assesment of safety-critical software. Technical Report 90-295, ISSN 0836-0227. Telecommunications Research Institute of Ontario. Queens University, Kingston, Ontario.

Pulli, P., 1989. Pattern-directed real-time execution of SA/RT specifications. *Proceedings of the Euromicro Workshop on Real Time*, June 1989. IEEE Computer Society Press, Los Alamitos, CA.

Struss, P., 1992. Knowledge-based diagnosis - an important challenge and touchstone for AI. *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 863-874. John Wiley and Sons, Chichester, England.

Truszkowski, W., Paterra, F., and Bailin, S., 1992. Knowledge from pictures. *Technology 2002 Conference*, December 1-3, 1992, Baltimore, MD. Proceedings to be published by NASA in 1993.

Wang, Y., 1988. A distributed specification model and its prototyping. *IEEE Transactions on Software Engineering,* Vol. 14, No. 8, pages 1090-1097. August 1988.

Zave, P. and Schell, W., 1986. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering,* Vol. 12, No. 2, pages 312-325. February 1986.