N93-17500

Developing Satellite Ground Control Software through Graphical Models

S1-61
136875
P-5

Sidney Bailin, Scott Henderson, and Frank Paterra

CTA Incorporated
Rockville, MD

Walt Truszkowski

NASA/Goddard Space Flight Center
Greenbelt, MD

1 Introduction

The old maxim goes, "A picture is worth a thousand words"—ten thousand, if you believe Larkin and Simon (1987). Most people, when faced with the problem of understanding the behavior of a complicated system, resort to the use of some picture as an aid in thinking about the system. Barwise and Etchmendy (1991) make a strong case for the effectiveness of diagrams, over and above other representations, in certain problem solving situations.

This paper discusses a program of investigation into software development as graphical modeling. The goal of this work is a more efficient development and maintenance process for the ground-based software that controls unmanned scientific satellites launched by NASA. The main hypothesis of the program is that modeling of the spacecraft and its subsystems, and reasoning about such models, can—and should—form the key activities of 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 outline of this paper is as follows. We describe the application domain in the next section, and the graphical modeling technique in Section 3. Sections 4 and 5 describe the approach to generating diagnostic and simulator software from these models. In Section 6 we describe the work we are doing in automated reasoning about the diagrams. Finally, in Section 7, we summarize what we think are the prospects for this program, the key issues, and major risks and unknowns.

2 The Domain: the Intelligent Ground System

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. Development of command loads prior to spacecraft launch employs simulation to verify their proper operation. 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."

The importance of such models is a result of an object-oriented system architecture, which is one of the defining characteristics of the IGS. The object-oriented architecture describes the IGS as a model of its environment. This environment consists primarily of the spacecraft, its subsystems, and payload, and the users of the IGS (the Flight Operations Team, or FOT), who are divided into several distinct roles within the control center. The environment may also be viewed as including the communications systems through which the IGS and the satellite interact, and various other ground systems to which a control center is typically connected. Each of these environmental elements is represented as a distinct object in the IGS. This approach enables us to make the IGS "intelligent" by making each such object a knowledge-based system in its own right, with its own simulation capability, diagnostic capability, etc.

There are various consequences of this architecture for the operation of the IGS, including the need for a cooperative framework, and for an intelligent user interface. The object-oriented architecture defines the IGS as a collection of interacting knowledge-based systems. This interaction models the interaction found in the system's environment, but it must also include means for cooperative problem solving among the system's components. For example, the successful diagnosis of telemetry anomalies may require interaction between the diagnostics of several subsystems. Thus, a key requirement of the IGS is a set of problem-solving protocols through which multiple knowledge-based systems, in conjunction with the FOT, can converge towards a goal. A framework for such cooperation is described by Bailin *et al* (1989).

The need for an intelligent user interface follows from the fact that he IGS does not operate autonomously—the health and safety of the spacecraft precludes such an approach. The FOT are active players in the cooperative process just described. Thus, the IGS must model the FOT roles in a way that a) facilitates communication between the human and the machine, and b) enables the system to interpret human actions within the cooperative protocols.

2.1 Implications for Software Development

The IGS architecture makes everything a model. 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 IGS—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 following sections describe the progress we have made to date in demonstrating this idea. Similar ideas have been put forward in a recent article by Harel (1992).

The more advanced IGS functions—the cooperative framework and the intelligent user interface—go beyond our current view of what can be automatically generated from graphical models. The reason is simple: we do not yet have an adequate understanding of these functions. Our work on the IGS is attempting to make inroads into these areas, especially the cooperative framework, but this work is still exploratory. We expect that with the definition of cooperative protocols, code implementing such protocols would be provided as reusable library assets. It is conceivable, therefore, that they could form a part of the automated development framework towards which we are working.

3 The Graphical Models

The diagrams consist of objects described by behavioral annotations and connected to each other by influence paths. Each object has a set of state variables, some of which serve as input ports (receiving influences), some of which serve as output ports (creating influences), and some of which are internal to the object. In translating such a diagram into a simulator, the influence paths are implemented as data flows. The influences paths may, however, correspond to the transfer of physical attributes, e.g., heat transfer, in the modeled system itself. Thus, the graphical representation is somewhat different from the conventional notion of a dataflow diagram.

The object behaviors are described in terms of states and transitions, but the representation is more powerful than that of a finite state machine. Each internal and output state variable has a finite number of "transitions" associated with it, but each such transition is a mathematically specified function. Thus, the domain of each transition is a set of possible initial state values; the resulting state, and any corresponding outputs, are a function of the initial state. This function may be defined in a piecewise fashion: that is, the set of possible initial states may be partitioned into a finite number of subsets, and the transition may then be defined on each subset by an appropriate expression. This seems to be similar to the approach recently advocated by Parnas (1990), in which tables are used to specify the discontinuities often present in functions that software is required to compute.

Components are stored in a library, so that they may be reused in many applications. Components are typed, and intuitively fall into a class hierarchy, although the library system does not yet support inheritance. Components may contain subcomponents as well as the state variables discussed above. In such cases, the interconnection of the subcomponents via influences forms part of the parent component description. There are no "systems" *per se* in the library: everything is a component. A system can be stored in the library as a new component, in which case it is available for use as a component in a still larger system in the future.

4 Generating Diagnostic Rules: the Knowledge from Pictures System

The Knowledge from Pictures (KFP) tool builds a knowledge base to perform fault detection, isolation, and recovery from a diagram of the monitored system.

The generated knowledge base takes the the form of facts and rules in the C Language Integrated Production System (CLIPS), an expert system shell developed by NASA/Johnson Space Center. The diagram is also used as the basis for the user interface of the diagnostic system.

Assertions derived from the behavioral descriptions of the diagram's components are used to determine when a component is in a state other than those in its definition (for example, a temperature sensitive object operating outside of its design temperature range). When such a situation has been detected, a fault has occurred. Alarms are defined as collections of component states. In the generated knowledge base, each alarm condition is represented by a CLIPS rule.

The rules generated by KFP use the influence paths shown in the diagram to isolate failed components. When an alarm is detected, a search begins for the faulted object causing the alarm. The 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 known states of the object being examined, and the states of the objects that influence it, to determine whether the examined object is behaving correctly. If the object being examined is not in the correct state, then the fault has been isolated. If it *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 is represented by a template for recovery rules—one for each fault/object pair. The action part these rules must be filled in by the knowledge engineer.

In KFP, the diagram of the system being monitored is also intended to serve as the basis for the diagnostic system's user interface. The control center operator should see a display of the system as a graphical model, with the status of its components expressed through color coding or similar conventions. The current KFP tool does not do this, but the concept has been demonstrated by another prototype system, the Generic Spacecraft Analyst Assistant (GenSAA).[1] Our plan is to integrate KFP with the next version of GenSAA by the end of this year.

[1] The GenSAA project is directed by Peter Hughes of NASA/Goddard's Automation Technology Section (Code 522.3).

5 Generating Simulator Software: the Multi-Aspect Simulation Tool

Our generic simulation architecture is based on the connection manager approach described in the Software Engineering Institute's (SEI) recommendations for flight simulators (Lee, 1990). In this approach, the influences between objects are simulated as data flows, and the data flows are implemented by connection managers—objects whose specific role is to manage the connections between application objects. The benefit of this approach is that the application objects themselves remain ignorant of the context in which they are used, and thus can be reused in quite different contexts.

In the Multi-Aspect Simulation Tool (MAST) we have extended SEI approach by independently formalizing each aspect of a component's behavior, by integrating work on discrete event simulation done by Zeigler (1990), and by implementing the design using the object-oriented techniques of multiple inheritance and virtual base classes.

Simulations typically represent system behavior along several dimensions. In MAST these dimensions are rendered by the interactions of independent aspect managers. Each manager is concerned with different component attributes. A gravitational manager, for example, is concerned with a component's position and mass, but not with its shape or color. All components subject to a manager appear to that manager with the same form, regardless of their actual structure. The manager can therefore assess and manipulate the components in terms of this standard form, oblivious to interactions occurring within the component with other aspects of its behavior. For example the gravitational manager should be able to change a component's position oblivious to the fact that the change also modified the component's shape. This homomorphy is available in C++ through multiple inheritance and virtual methods.

MAST integrates both discrete event simulation and continuous simulation techniques. For continuous aspects of the simulation, the associated aspect manager schedules re-evaluations at regular intervals of simulated time. These intervals can be decreased during the simulation to enhance the fidelity of the behavior rendered for a particular passage, and then lengthened to speed the simulation through a passage where little is changing. For discrete aspects of the simulation, the associated aspect manager schedules re-evaluation at the time of the most imminent event known. When that simulated time is achieved, the aspect manager executes the associated event, propagates its effects, and then computes the next imminent event for scheduling. A central simulation manager decides how to advance the logical clock by perusing each manager's schedule. The clock is advanced to the most imminent re-evaluation time, and the managers who are scheduled for that time are executed.

Although not yet implemented, we view it as a straightforward task to generate the connection management code automatically from the graphical models, and plan to do so in the near future. Generating the specific algorithms of each aspect manager, using the associated behavior specifications from the graphical model, would be a far more difficult task, which we do not plan to tackle in the near future.

6 Reasoning about the Diagrams

We have been working for several years on an automated reasoning system that takes diagrams as input. The GROVER system attempts to interpret the diagram as a high-level description of a proof plan, and it attempts to carry out the plan using an underlying "conventional" theorem prover (Barker-Plummer and Bailin, 1992). 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 this (as yet unnamed) tool 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 the IGS 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.

7 Conclusions

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

We noted in Section 2 that the full IGS concept includes a lot more than a collection of simulation and diagnostic programs. We are not yet in a position to say whether these advanced capabilities can be accommodated in our application development framework. Even if they are not, however, the current framework raises the level of abstraction at which a significant amount of development for a control center is performed.

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 (Holzman, 1992; Lin and Liu, 1992)

References

Bailin, S., Moore, J., Hilberg, R., Murphy, E., and Baher, S., 1989. A logical model of cooperating rule-based systems. *Telematics and Informatics*, Vol. 6 Nos. 3/4, pp. 331-349.

Barker-Plummer, D. and Bailin, S. Proofs and pictures: proving the diamond lemma with the GROVER theorem proving system. *AAAI Symposium on Reasoning with Diagrammatic Representations*, March 1992.

Barwise, J. and Etchmendy, J., 1991. Visual information and valid reasoning. Preprint.

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

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

Larkin, S and Simon, H., 1987. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11, pp 65-100.

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

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

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

Zeigler, B., 1990. Object-oriented simulation with hierarchical, modular models. New York: Academic Press.