MODEL-BASED PROGRAMMING TOOLS FOR INTEGRATED MONITORING, SIMULATION, DIAGNOSIS AND CONTROL

Gabor Karsai, Sumir Padalkar, Hubertus Franke *, Janos Sztipanovits
Department of Electrical Engineering
P.O. Box 1824 Station B
Vanderbilt University
Nashville, TN 37235

Abstract

This paper presents a software technique which is useful for building the monitoring, simulation, diagnosis and control software for complex systems, like the ones found in aerospace systems. The technique is based on the extensive use of models, which include description of the equipment as well as the software configuration itself. The system has been successfully used in various practical applications, and it is being evaluated by various sites.

Introduction

The development of complex automation systems, like the ones in aerospace systems, is a formidable task. People implementing these systems must have training both in the methods of the domain engineering, as well as in software engineering. The two fields are not separable, because almost all of the monitoring and control systems are now computer-based which means "implemented in software".

We argue that these software systems cannot and should not be developed independently of each other. If they are, simply because of interfacing problems, solutions are prone to serious problems. On the other hand, both fields, i.e. the domain-engineering and the software-engineering, are large enough on their own, that we cannot expect from a person proficient in one of them to grasp the other.

In this paper we propose a technology, called the Multigraph Architecture (MGA), which provides ways for building complex, hardware-close software systems in such a manner, that domain engineers are capable of building complex applications on their own. It is based on our multi-year research effort in model-based programming methods and environments. The technology has been useful in many of our research projects [4, 5, 6], as well as in various practical, real-life applications [3].

Backgrounds

When complex, real-time software systems are developed using the traditional methods of software engineering, skilled software engineers are given specifications of the "program". Then these requirements are turned into designs, the designs implemented, tested, and then integrated with the hardware. The problem with this approach is that software is mostly developed completely independently of the hardware, and more than once, they are not up to date, leading to problems at system integration time, or, even at run-time. Many of the design specifications and design decisions related to the hardware do not find their ways into the software design. The actual hardware, the "plant", is more or less independent of the software, and the software has very few knowledge of it.

There is a software methodology which is suitable to help in this situation: it is called model-based programming. In many of our research projects we have successfully used, and there are other references in the literature indicating the viability of the approach. The central idea of model-based programming is the use of various models in the software development. As opposed to the traditional analysis/design/implementation sequence, in model-based programming we start with models, and end up with an executable system, which was created from the models using a process called model interpretation. The conceptual flow of model-based program development is shown on Fig 1.

The models are various, very high-level descriptions of the system. Here the term "system" includes the "plant" as well as the software. This inclusion, as it

*Dr H. Franke is currently with IBM T.J. Watson Research Center

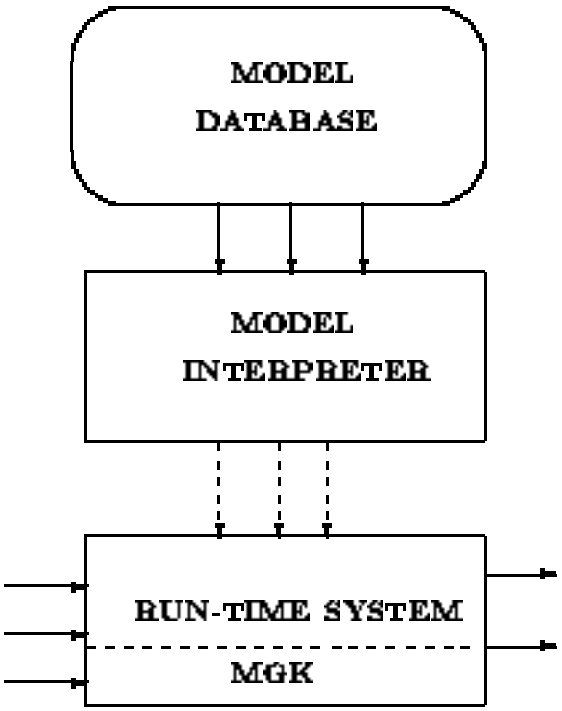

Figure 1: Model-based Program Development

turns out is a very crucial one: this is used for ensuring the consistency between that hardware and and its associated software. Once models are created and entered into a database, typically by domain engineers, they can be augmented to implement various software components. This latter process happens also through the use of models.

When model specification is finished, the rest of the system generation can be made sufficiently automatic: a model interpreter traverses the model database, creates and executable system from pre-fabricated software objects by properly configuring them into an coherent whole.

Obviously, the capabilities of a model-based system are determined by the richness of the modeling concepts available in the system, i.e. provided as database schemata and understood by the interpreters. In the sections below we describe the modeling concepts what we found suitable for complex automation systems, then discuss the model database organization, the model editing techniques, the method of model interpretation and finally the run-time support.

Modeling concepts

Engineering systems always accomplish a practical mission, therefore there are always *functionalities* what they have to provide. When modeling the plant (i.e. the environment of the automation software), we assume that these functionalities are clearly defined, and we are capable of modeling them in a structured way. In addition to functionalities, the plant is running on some hardware, which should also be modelled. One might say, that for functionalities we model the requirements, for the hardware we model the actual physical realization. The third category of models should deal with the software: the modeling concepts here should deal with the functionalities expected from the software. In our system we use the following terminology: functional models are called *process models*, hardware models are called *equipment models*, and software models are called *activity models*.

Modeling concepts in themselves are not suitable for building complex systems: model organization techniques are also needed for handling complexity. In the MGA there are a set of organizational paradigms supported, including:

* part-whole hierarchy,
* multiple-aspect modeling,
* cross-references among models,
* conditionalized models

The part-whole paradigm supports the hierarchical decomposition of systems: all the basic model categories support it. Multiple-aspect modeling [2] is a technique which supports complexity management by letting the user look at models only from a certain aspect at a time. The information presented to the user is thus reduced. Aspects are not independent of each other, and consistency should be maintained. Cross-references among models can ensure consistency between the plant models and activity models: if the specification of the plant changes, that will automatically propagate to the software models. Conditionalized models are capable of describing models which are dependent on the state of a plant: essentially different models are used in different, discrete states.

Plant models, which include the process and equipment models provide a method for presenting what is known about a particular plant. Process models are used for modeling functionalities in the plant. They have the following aspects: (1) structural, (2) discrete states, (3) mathematical, (4) fault propagation, and (5) process/equipment association. In the *structural* aspect one can specify the hierarchical decomposition of a process, (i.e. how it is built from lower-level processes), the relevant process variables (quantities, which determine the overall state of the process), the relevant process parameters (quantities, which are design-time constants), and process interactions among processes through material, energy, and information flows. In the *discrete states* aspect one can enumerate the major regions of the state-space of the process

(e.g. halted, running, suspended, etc.), and the possible transition among them. In the *mathematical* aspect one describe the process using standard mathematical notations, including state-space models, algebraic and differential equations, and others. The mathematical models essentially prescribe *mathematical constraints* among the values of process parameters and process variables. The *fault propagation* aspect is used for modeling what discrepancies (faults) are expected in a process and how these propagate in the plant (i.e. how a fault introduces other faults). Note that the propagation graph is also a hierarchically organized structure. The *process/equipment association* aspect can be used for modeling what equipment is used by a particular process. In this aspect sophisticated relationships between the process and its equipment can be described, including relating process faults to equipment failures, and the dynamically changing assignment of processes to equipments. Equipment models have far fewer aspects: (1) structural, (2) discrete states, and (3) fault states. In the *structural* aspect similar information can be modeled about the equipment as in the case of processes, but equipment characteristics (equations, etc) can also be included here. The *discrete state* aspect is similar to the one for process models. The *fault states* aspect can be used to model equipment faults (i.e. concrete problems with the hardware).

Software is modelled in terms of *activity* models. Activities are anything what the final monitoring, diagnostics, simulation, and control software does. We found it very plausible to use hierarchically organized block diagrams for activity models, with primitive and compound activities. There is a predefined set of primitive activities, including algorithmic activities (subroutines in a procedural language), timer activities (timing generator blocks), simulation activities (simulation equations solvers), operator interface activities (operator screens), etc. The activities are organized into block diagrams which also describe the way activities are scheduled. As it will be presented below, a macro dataflow scheduler is used as run-time support.

Activities can be independent of plant models, but typically they should be based on information stored in the plant models. Activity models are typically *related* to plant models via the use of *reference objects*, which are essentially pointers to objects in plant models. There various kinds of reference objects, including:

- *process parameter references* for connecting software parameters to process design constants,
- *process variable references* for connecting software variables to process quantities,
- *process equation references* for configuring the simulation activities to use equations defined in the context of process models,
- *process fault references* for relating computational events (e.g. software alarms) to anticipated process discrepancies.

Model Category	Aspects
Process Model	Structural
	Discrete States
	Mathematical
	Fault Propagation
	Process/Equipment Association
Equipment Model	Structural
	Discrete States
	Fault States
Activity Model	Structural

Table 1: Modeling concepts

Note that activity models are thus tightly coupled to plant models: whenever a plant model changes, the related activity models get changed as well. This feature ensures the integrity of models. Table 1 shows the major modeling concepts and categories.

Model database

The modeling concepts of MGA are complex enough that a sophisticated background storage system is needed. MGA models are essentially *objects* which are organized into complex networks. There are many ways for storing models, including:

1. Use Lisp text and a workstation's file system. This approach is simple at the first sight, but quickly breaks down if more sophisticated features are needed (e.g. versioning, change propagation, etc.)
2. Use an OODBMS. This methods is more flexible but typically requires a database package. In our experience, this is the method to follow as OODBMS are becoming more commonplace.

In MGA the objects of different model categories are stored in hierarchies, and it is possible to have more than one rootpoints in a hierarchy. A collection of models which are related to the same plant is called a project database. The models are stored as persistent objects, and they are accessible through a Common Model Interface (CMI). The CMI is part of all the tools and the run-time system of MGA, and provides a uniform access mechanism.

Model editing

The specification of complex models is supported by a graphical model editor. Because for many models the most natural way of expression is that of diagrams, we have used a visual representation formalism wherever it was suitable. Models are specified in terms

Figure 2: Typical editor window

of icons, connections, networks, and other graphical components. There are certain components, however, which are specified textually: e.g. equations have to be typed in using a certain syntax. It would be possible to model equations graphically, too, but the mathematical (textual) form simply seems more natural.

The models are built using graphical editing operations, e.g. "add", "connect", "move", etc. Browsers are available for selecting models from the database, and many icon-based operations for the editing process itself. The model builder has a direct-manipulation interface: the picture is *not* parsed as a visual language, changes are performed on the database as changes are made on the drawing. A typical editor window is shown on Fig 2.

Model interpretation

Once the model database is prepared, the model interpreter component takes over and the executable system is created. The model interpreter is a complex program which traverses the model database (starting from a specified root), checks all the models accessed, and creates run-time objects from them. Note that models are "abstract" specifications, and the model interpreter is the component which "instantiates" them into run-time components.

Note that the use of model interpretation attempts to combine the advantages of compilation and interpretation. Compilation, which is equivalent to fully automatic program generation, is very difficult to do, and not very flexible. Pure interpretation is very flexible, but very inefficient. Model interpretation takes the middle road: it assumes that a flexible run-time system is available which is configurable to a wide range of problems.

MGA allows the inclusion of user-defined software modules: the algorithmic activities can contain subroutines written in an arbitrary procedural language. Any external libraries can also be linked into the final software, while MGA provides the integration facility for these via the activity models. It is even possible to access MGA models from these user-defined subroutines.

The model interpreter is responsible for instantiating and configuring the run-time objects of the system, including:

* the overall run-time system for scheduling of the activities (this is discussed below),
* the simulation run-time system, which includes various equation solvers, etc.,
* the diagnostics run-time system, which performs real-time fault diagnostics on the plant,
* the external interface run-time system, which interfaces to the plant instrumentation,
* the operator interface run-time system, which provides facilities for operator access and interaction,
* monitoring and control run-time system, which implement the elementary monitoring and control functionalities.

All these sub-systems are configured based on the contents of the model database.

The integrative nature of the model interpretation makes possible to create very complex configurations. For example, it is very easy to create an activity network which collects data from the plant, runs a simulation in parallel with the plant to predict the plant's forthcoming behavior, captures alarms from the plant, diagnoses faults, and provides an on-line operator interface.

Run-time support

When the model interpreter creates the run-time objects, their configuration determines the functionalities provided by the system, i.e. how actually the monitoring, simulation, control, and diagnostics activities are performed. The algorithms attached to the objects are implemented in such a way that they are reusable in different configuration. There is an obvious need for a run-time support system which makes possible this dynamic configuration.

Many MGA models are based on diagrams, and the activity models are almost entirely configured through them. These block diagrams can naturally be considered as *dataflow diagrams*, where the blocks denote processing nodes, and the edges describe dataflows. It is also natural that the blocks are scheduled for execution, whenever data is available on their input datastreams, and produce data for further, downstream processing.

This implies that the blocks can be scheduled according to the dataflow principle.

These observations lead to the development of the Multigraph Kernel (MGK), the run-time support of the MGA [1]. It is a run-time kernel, which implements the support for graph building and graph execution. It is running on a wide range of architectures, including PCs, networked workstations, shared-memory multiprocessors, and distributed-memory multiprocessors.

The modeling concepts of the activity models are naturally mapped into MGK computational graphs, and the execution of these graphs is scheduled by the kernel program. Naturally, there is price to be paid, because instead of direct function calls between modules, the MGK scheduler should take care of the activation. However, in our experience this overhead was always negligible.

A detailed example: Diagnostics

Real-time fault diagnostics is a component in MGA of great importance. Its goals is to capture alarms from the plant, and provide the operator with online information regarding fault source processes and faulty components. In this section we briefly describe how it is using the plant models.

In the *fault propagation* aspect of the *process models* one can model:

- what functional failure modes are anticipated in each process,
- how these failure modes propagate from one process to a neighboring one, and
- how these failure modes propagate *within the hierarchy*, ie. how "low-level" failure modes lead to higher level ones.

With each propagation one can associate *probabilistic* and *dynamic* information: the minimum and maximum propagation times, and propagation probability. A typical failure propagation graph is shown in Fig 3.

In the *fault states* aspect of the *equipment models* one can model the anticipated fault states of equipments, and additional information (e.g. severity, MTBF, etc.). The two model categories can be integrated in the *process/equipment association* aspect of the process models: process failure modes can be connected through causal links to equipment fault states, expressing the fact the functional failure mode (e.g. "level high") is caused by the physical fault state (e.g. "stuck valve").

The above two models describe what is known about the plant. In the activity models one can configure the *diagnostics activity* as follows. Process failure modes can be linked to alarms, which are active objects indicating the presence of failure modes. These alarms

Figure 3: Failure propagation graph

constitute the sensory manifestations of process failure modes. Equipment fault states can be linked to event objects which get activated whenever the diagnostics system has pinpointed the relevant equipment's fault state as the source of the failure. In a sense, the diagnostics activity works as just another activity block: its inputs are alarms and its outputs are events which indicate the failure source(s). Internally, the diagnostics system traverses the fault propagation graph backwards, and using the timing information associated with alarms, determines the primary failure mode of an alarm sequence [2]. This primary failure mode is then projected onto equipment hierarchy, and the relevant equipment fault state is presented as diagnostics result. An operator screen showing the diagnostics output can be seen on Fig 4.

There are two observations to make. (1) The fault propagation graphs can be used not only in backwards manner, but also in a forward manner and can be used to predict impending failure modes in the future. (2) Sometimes failure propagation time parameters are not constants, but variable according to the values of certain variables (e.g. flowrates). We have implemented this feature of dynamically changing timing parameters: activity blocks can be used to dynamically compute these propagation parameters.

Conclusions

We have presented an integrated system for monitoring,simulation, diagnostics and control which is based on the Multigraph Architecture. It is a model-based system: models are used to encode knowledge about the plant, as well as the software system. A technique for automatic system generation: model interpretation is used for generating complex software systems. We have learned that it is a viable approach for system building. The system has been successfully used in various configurations and applications.

Figure 4: Operator interface screen

Acknowledgment

The research described in this paper was supported in part by the Boeing Company, USAF, and Osaka Gas Company.

References

[1] Biegl, C.: "Design and Implementation of an Execution Environment for Knowledge-Based Systems", Ph.D. Thesis in Electrical Engineering, Vanderbilt University, 1988.

[2] Padalkar,S, Karsai, G, Biegl, C, Sztipanovits, J: "Real-time Fault Diagnostics Using Hierarchical Fault Propagation Models", *IEEE Expert*, pp. 73-85. June 1991.

[3] Karsai, G, Sztipanovits, J, Padalkar, S, Biegl, Ca, et. al: "Model Based Intelligent Process Control for Cogenerator Plants", Journal of Parallel and Distributed Systems, pp. 90-103, June 1992.

[4] Sztipanovits, J., Biegl, C., Karsai, G., Bourne, J., Mushlin, R., Harrison, C.:"Knowledge-Based Experiment Builder for Magnetic Resonance Imaging (MRI) Systems," *Proc. of the 3rd IEEE Conference on Artificial Intelligence Applications*, Orlando, FL, pp. 126-133, 1987.

[5] Sztipanovits, J., Karsai, G. et al.: "Intelligent Test Integration System," *Proc. of the Conference on Artificial Intelligence for Space Applications*, Huntsville, AL, pp. 177-185, 1986.

[6] Sztipanovits, J., Biegl, C., Karsai, G., "Graph Model-Based Approach to Representation, Interpretation and Execution of Real-Time Signal Processing Systems", *International Journal of Intelligent Systems*, pp. 269-290, Vol.3., No.3 1988.