intricately interlinked with their environment, and which must be used by domain-engineers, the model-based approach is an important and viable technique.

10 Acknowledgements

The Vanderbilt group was supported, in part, by Osaka Gas Information Research Institute Co. Ltd, Osaka, Japan. Their kind support is acknowledged. The IPCS is currently being commercialized by OGIS-RI.

References

[1] B. A. Abbott et al.: "Model-Based Approach for Software Synthesis", IEEE Software, May, 1993, pg. 42-52.

[2] Booch,G.: *Object-oriented Analysis and Design with Applications*, 2nd. ed., The Benjamin/Cummings Publishing Company, Inc, 1994.

[3] *G2 Reference Manual*, Gensym Corporation, 1990.

[4] Karsai,G: "A Configurable Visual Programming Environment", IEEE Computer, March 1995, pg. 36-44.

[5] Moore,M.,Nichols,J.: "Model-Based Synthesis of a Real-Time Image Processing System", in this Proceedings.

[6] Padalkar,S. et.al: "On-line Diagnostics Makes Manufacturing More Robust", Chemical Engineering, Mc-Graw Hill, March, 1995.

[7] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.

[8] Sztipanovits,J. et al: "MULTIGRAPH: An Architecture for Model-Integrated Computing", in this Proceedings.

[9] Stephanopoulos,G.: "MODEL.LA. A Modeling Language for Process Engineering, Parts I, II," Computers and Chemical Engineering, 14, 495-539, 1990.

[10] Dee, K. and Westerberg, A.: "CEPHDA: Chemical Engineering Process Hierarchical Design with ASCEND", EDRC 06-140-92, Engineering Design Research Center research report, Carnegie Mellon University, 1992.

[11] Wilkes,M.D. et al. (1993) "The Multigraph and structural adaptivity", *IEEE Transactions on Signal Processing*, pp 2695–2717, August 1993.

values of unmeasured process variables. The application utilizes about 10 process variables and predicts 4 other variables. The application was created in a week by one engineer.

A third application uses balance equations for diagnosing process sensors. Material and energy balance equations are executed using data received from redundant sensors, and because sensors might appear in multiple balance loops, one can diagnose their failures. Statistical analysis can detect other sensor failures, including "stuck" sensors. This application diagnoses 35+ sensors, and was created by two engineers in about 2 weeks.[6]

A complex application

A fourth application is being developed, which is using a very sophisticated external process simulator (running on a remote supercomputer) for dynamic modeling and control purposes. The application helps in predicting the behavior of a refining train in a polyester intermediates plant. The refining train consists of several distillation columns, and it is a highly interlinked structure, with non-linear components. The behavior of the system is modeled by about 8,000 nonlinear differential equations. The application predicts dynamic process behavior upto 100 hours into the future through numerically solving the equations.

The application works as follows:

1. Real-time plant data is acquired through the use of the external interface activity models.
2. This data is preprocessed and partially displayed.
3. Some of the data is sent to a remote supercomputer, where the simulation is running. The data provides initial conditions and parameter values for the differential equations. The simulation is configured from IPCS models. The activity model which interfaces the IPCS internal dataflows with the simulator, lets the user visually configure the connections between IPCS signals (i.e. process variables) and simulation variables.
4. The simulation results are "brought back" into the IPCS dataflow, and are displayed (after some post-processing) on an operator screen.
5. Eventually, simulation results will be used to specify setpoints for the controllers on the process.

These results indicate that the system offers a viable platform for problem solving. Chemical engineers with minimal training in software engineering were capable of building relatively complex applications in a short time.

7 Other domains

While IPCS has demonstrated the viability of the approach in the field of chemical engineering, the question remains how these approaches can be applied in other domains. The answer lies in the underlying technology that has been used to develop The Multigraph Architecture is a flexible toolset for creating domain-specific problem-solving environments; and its flexibility is achieved through the use of *generic* tools (e.g. configurable visual model builder, execution environment, etc.).

To demonstrate the flexibility of these generic components here are some examples:

- The configurable visual model builder[4] has been used for building environments for high-performance parallel instrumentation[1], fault-modeling in aerospace systems, and real-time image processing[5]. The configuration is done through a special language and a program generator. Typically, a visual model builder for a new domain can be developed with a few man-day effort.
- The execution environment[11] has implementations for different platforms, for instance: multiprocessor networks built from high-speed DSP chips (used in the parallel instrumentation application), networked workstations (used in IPCS), and others.

In the case of IPCS, the generic tools constitute a considerable part of the system; the rest is specific to the chemical engineering domain. While there is flexibility and generality in MGA, to develop tools for a new domain still requires considerable expertise and the intricate knowledge of the tools. We are currently working on technologies to support this process.

8 Related work

The use of problem solving environments for chemical process industries is not a new idea: ASCEND[10] and MODEL.LA[9] are two examples for problems solving environments in an chemical engineering setting: the first is a high-quality design environment (with sophisticated solvers, languages, etc.), the second is a general modeling approach and framework for representing processes in chemical industries.

In IPCS we took a slightly different approach: in addition to modeling the processes, we provide facilities for representing (or "coding") the problem solution strategies as well. Very complex applications can be built using this approach, and through the use of the activity models, vastly different problem solving strategies can be integrated. For instance: real-time plant data can be acquired, and compared against ideal models (through the use of a simulator). Next, if failures are detected, a process diagnostics can be started which identifies the faulty equipment. This detection process may result in changing the setpoints of controllers, and/or changing the monitoring strategy used for observing the process.

9 Summary

IPCS is a problem solving environment that uses embedded models and model-based software synthesis to achieve its goals. This system was found useful by practicing plant engineers in solving their problems, using their own terminology and language. We believe, that for complex computer systems, that are

- *Types:* Models (if they satisfy certain requirements) may be converted to model types, which are library components. Types can be re-used in many model configurations, and they are stored in a single, shared copy.

The application of these model organization principles makes possible the creation of very complex models for large-scale plants.

5 Model interpretation

The task of the model interpretation process is to map the (plant and activity) models into executable code, which provides solution to problems. This process might be considered as *software synthesis*, since an executable system is generated from high-level specifications[1].

In IPCS the model interpretation process is performed in two steps.

1. At *system integration time* the model database is traversed and analyzed for procedural code that needs to be compiled. This happens by descending on the various model hierarchies and compiling the code fragments required by the individual models. This approach is used, for example, in compiling algorithmic activity codes, and in compiling the equations for process models. Finally, the system integrator tool links the application code with the run-time model interpreters and other support libraries and creates the executable.

2. At *run-time* the "real" model interpreters perform their task of traversing the hierarchies and creating the run-time objects.

The run-time model interpretation process is performed as follows. (Note that this is completely transparent to the end-user, and it is fast enough to be hardly noticeable.) There are various model interpreters for each model category. These interpreters are encapsulated in the form of `Builder` objects, that are created according to model objects in the model database. One model object may have multiple builder objects created from it. Builder objects are organized into hierarchies, and this hierarchy mirrors the model object hierarchy. Once the builder objects are created, their hierarchy is traversed, possibly in multiple passes, such that the objects create the run-time objects. Run-time objects constitute the final, executing system. Note that once run-time objects are built, the model and builder objects are no longer necessary and they can be discarded. Figure 3 shows the relationship between model, builder, and run-time objects.

The run-time objects are configured according to the models. This configuration happens on two levels: (1) models determine the behavior of *individual* objects (e.g. an algorithmic activity), and (2) models determine the *interactions* among the objects (the dataflows, data dependencies, etc.). Thus, the model interpreters configure the individual *objects* (by passing parameters to their constructors, setting their data

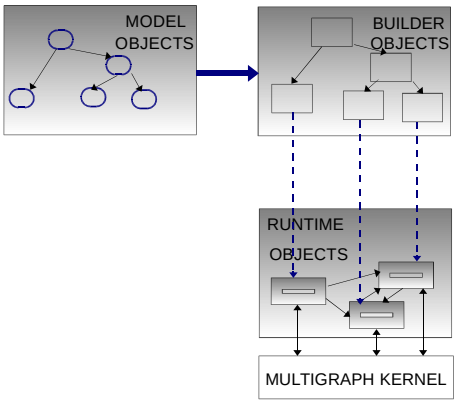

Figure 3: Objects related to model interpretation

members, etc.), but also the *network of computations* built from these objects.

These computational networks are specified in terms of the Multigraph Computational Model (MCM). The MCM is a macro-dataflow model providing a unified system integration layer above heterogeneous computing environments including open system platforms, high performance, parallel/distributed computers and signal processors [1] [11]. The run-time support of the MCM is the Multigraph Kernel, which schedules the computations. The elementary computations are carefully defined reusable code components that are part of application specific run-time libraries. The MGK is implemented as an overlay above operating and communication systems. A unique capability of the MGK is its support of the dynamic reconfiguration of the executing system [11].

6 Generated applications

IPCS has been successfully used in various, practical applications, which solved actual problems. One application monitors 16, business–critical process variables, and continuously displays them. Some of the variables calculated, some are directly obtained from the plant data acquisition system. The system contains 10 activity models, and the entire application was created by one engineer, without any previous experience with IPCS, in a few days.

Another application uses an external, remote simulation process (running on a supercomputer), that utilizes process models and on-line data to predict the

Paradigm	Model	Aspects
Plant	Stream model	Process flow
	Process model	Process flow
		Finite states
		Equations
		Failure propagations
		Proc./equipm. associations
	Equipment model	Structure
		Finite states
		Faults
Activity	Algorithmic	Signal flow
	Timer	Signal flow
	Finite-state machine	Signal flow
	Operator interface	Signal flow
	External interface	Signal flow
	Simulation	Signal flow
	Compound	Signal flow

Table 1: Modeling Paradigms in IPCS

valve) or a complex assembly (like a distillation column). An equipment model represents a part of the plant's hardware.

The aspects of equipment models describe (1) how the equipments are hierarchically organized, (2) what are the discrete states for an equipment, and (3) what kind of faults are anticipated for the equipment.

Activity models describe computations which implement the desired monitoring, control, simulation, and diagnostics problem solving functions. They are hierarchically organized *dataflow diagrams*, each node representing a (simple or complex) activity performing some kind of data processing.

The various models in the *Activity* paradigm, are for specific, frequently used problem solving activities. An *algorithmic* activity is a piece of procedural code, that executes a user- (or library-) defined function. A *timer* activity represents a block which implements a delay function. A *finite-state machine* activity implements a state-machine, which is triggered by events, and performs state transitions. An *operator interface* activity provides a window for the operator, equipped with slide-bars, plotters, etc. An *external interface* activity connects the datastreams of the activity model with the plant's data acquisition system. The *simulation* activities provide ways for configuring a built-in simulator, and incorporating it into the datastreams of the problem solving activities. A *compound* activity contains other activities, making possible the organization of activity hierarchies.

Activities are typically model-based, meaning that they utilize engineering knowledge captured in the form of plant models. This utilization occurs with the help of *reference* objects, which may appear in activity models, and refer to objects in the plant models. For example, a simulation activity is configured by specifying what processes one wants to simulate. This is done by placing reference objects into simulation activity models, which objects point to the equations present in the *Equations* aspect of process models.

Essentially, activities represent two different things:

1. On the *conceptual level* they represent problem solving strategies. (For example, acquire data, filter data, calculate something, use data to run a simulation, compare the simulation results with plant data, interpret the results, etc.).

2. On the *physical level* they represent software configurations which solve the problem.

The central idea in IPCS is that the plant models and the activity models are integrated, such that the automatically generated application will perform the required problem solving activity, and the plant models involved will contribute to this process.

The variants of activity models indicate the variants of available problem solving strategies: computations (algorithmic), time-based (timer), discrete control (finite-state machine), data collection (external interface), operator communication (operator interface), simulation (simulation), etc. There are built-in activities which are not explicitly modeled, though their results can be incorporated into the dataflow networks of activities: plant state tracking (through the use of the finite state models of processes and equipments), and fault diagnostics (through the use of failure propagation models of processes). The fault diagnostics[6] detects incoming alarms, and using the propagation information embedded in the models, tries to come up with a plausible explanation for the current alarm pattern. The "output" of this activity (i.e. the identified fault cause) can be used as input to other activities.

There are many organizational principles used in the models to reduce model complexity. The principles include:

- *Hierarchy:* Models represent entities on various levels of abstraction.

- *Multiple aspects:* Models may contain many objects (icons, connections, etc.), but only those are shown simultaneously which are relevant in a given *aspect*. Aspects partition a complex model into manageable pieces. The modeling paradigm defines what aspects are available, what objects they contain, and how they interact with each other.

- *References:* Models can contain "pointers", i.e. references to other models. For example, activity models (i.e. computations) must often be tightly coupled to plant models (e.g. alarm limits may be dependent on equipment size), and this can be expressed by including a reference to an equipment parameter in the activity model for the alarm detection operation.

- *Conditionals:* Model components may be conditionalized based on the "activeness" of other components. For example, depending on what state a process is in (e.g. shutdown, startup, or running), different mathematical models describing the process must be used.

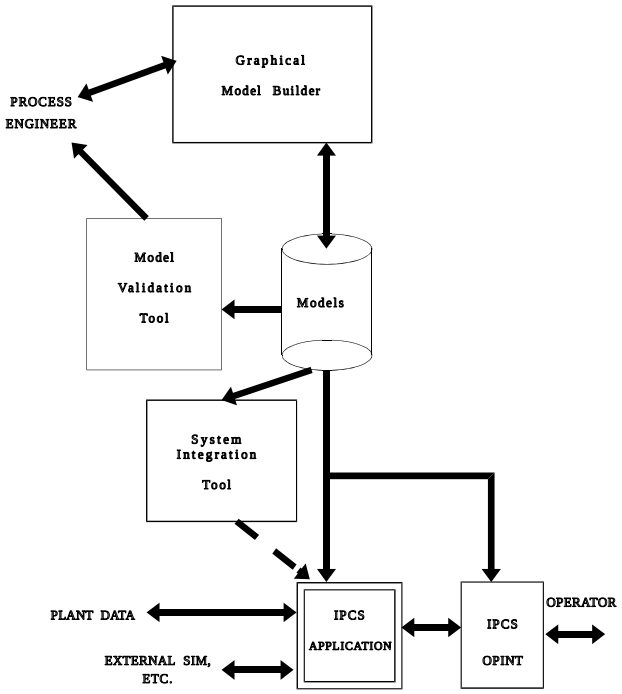

Figure 2: IPCS System Components

ated a *system integrator tool* builds an executable application code from the models, that, together with the model database, constitutes a particular, problem solving system (e.g. a custom simulation, etc.) This executable, when started, goes through *a model interpretation phase*, which creates various run-time objects as prescribed by the models, and starts "running" those under the control of an internal scheduler, the Multigraph Kernel (MGK)[11]. The application is using a generic *operator interface* package (configured according to the models) for run-time user interaction, and is interfaced with the plant's data acquisition system and possibly other software packages (e.g. process simulators). The individual components of the system and their interrelationships can be seen on Figure 2.

The individual components were designed with the following justifications in mind:

- The visual model builder had to be graphically oriented, because diagrams are the "natural" language for process engineers. It also had to support text manipulation, because that method is more obvious for entering, e.g. equations, than the graphical one.
- Model complexity and the sheer size of models necessitated the use of an object database. Other, e.g. relational databases lack the necessary level of support for storing complex models.
- The system had to be built that way, that it provided predefined concepts which are common in all process engineering applications (e.g. `Process variable`), so domain engineers could use it immediately.
- The abstract, high-level models had to be translated into actual problem solving code. The model interpretation process (together with the system integration) accomplished this task. Runtime libraries provide a set of "skeleton" problem solving activities (e.g. simulation), which are instantiated according to the models.
- Interfaces to the plant data and human operators had to be provided, because process engineering typically involves both. These interfaces had to be made as simple and usable as possible.

Naturally, the key to the usability of the system is determined by the modeling paradigm provided. We discuss that next.

4 Modeling paradigm

The modeling paradigm in a model-based system specifies what kind of models can be built, what they represent and how can they be structured. In IPCS there are two major modeling paradigms used: one is used for modeling the plant, the other one is used for modeling the problem solving activities. Each modeling paradigm defines a set of models what the user can instantiate. When the user creates a new model instance, he/she must specify its attributes, parts, and connections among the parts. Models may contain other models, thus hierarchical organization is supported. Each model can be looked at from a set of *aspects*, which define a partition over the model's parts and connections. This method also helps reducing (visual) model complexity. The paradigms, their models, and their aspects are summarized in Table 1.

Process models are used for representing functionalities in the plant. They are hierarchically organized diagrams, each block in the hierarchy describing a material, energy or information transfer process. A process represents an identifiable sub-functionality of the plant.

The aspects of process models describe (1) the process flows (how processes interact through various streams, and how they are organized hierarchically), (2) what are the mathematical models that describe the process, (3) what are the finite, discrete states what the process can be in, (4) what failure modes are expected for a process and how they propagate in the process network, and (5) what equipments are used by the process.

Stream models represent the material, energy, or information flows, connecting processes.

Equipment models describe what the plant's hardware is and how it is constructed. These models are also hierarchical diagrams, each node representing a piece of hardware, be it a simple component (like a

code might be attached as well. The problem solving strategies must be explicitly coded in the form of rules. This method has some serious drawbacks: engineers should be trained to become knowledge engineers, the possibility of using frames and rules does not offer any support for how to use them, and they lack the predefined concepts prevailing in the application domain.

While there are many practical systems and environments which try to solve problems using the above approaches, one can anticipate that, as the system's complexity increases, their shortcomings will become more prevalent. This does *not* mean, however, that these approaches are not valuable: they have their own place and application.

A better approach might be to use the "model-based programming" technique for building such domain-specific problem solving environments. The approach itself is described in a different paper[1], here we merely summarize it from our standpoint, as follows:

- The central idea is to use *models*, which are abstract representations of concepts and entities relevant to the problem solving process. A model, technically speaking, is merely a data object which represents something meaningful. Examples for models will be given below.

- We use a *visual model-building environment* [4] for creating models of the plant, and the problem solving activity to be used. This environment has built-in concepts from the application domain (e.g. `Process`, `Process variable`, `Equipment`, `Failure mode`), and some concepts for typical problem solving activities (e.g. `Operator interface`, `Simulation`, `Event`, etc.). The problem solving activities are configured through a visual language, although procedural code might be embedded as well. We store these models in a model database.

- A system component, called the *model interpreter* is responsible for interpreting the models, configuring networks of active, "executable" objects which implement predefined (or user-specified) problem solving strategies. This process is automatic, and corresponds to the compilation step in traditional approaches. The result of this step is a network of active objects, which are interfaced to the data streams in a plant, and/or possibly other software components (e.g. process simulators), etc.

- A run-time support component schedules the computations (i.e. activates the objects) as data arrives and/or user interaction mandates it. This component acts as a "run-time system integrator" which couples the computational activities and essentially runs the system.

The steps of this model-based approach can be seen on Figure 1.

This approach has the following advantages: (1) domain-engineers can work with their *own* concepts and language, there is no need to learn a new language, (2) problem solving activities can utilize knowledge that was formalized and represented, and (3) the problem solving strategies themselves are formulated in a way which is close to the thinking of domain-engineers. While, admittedly, there are drawbacks in this approach with respect to efficiency, they are outweighed by the advantages gained.

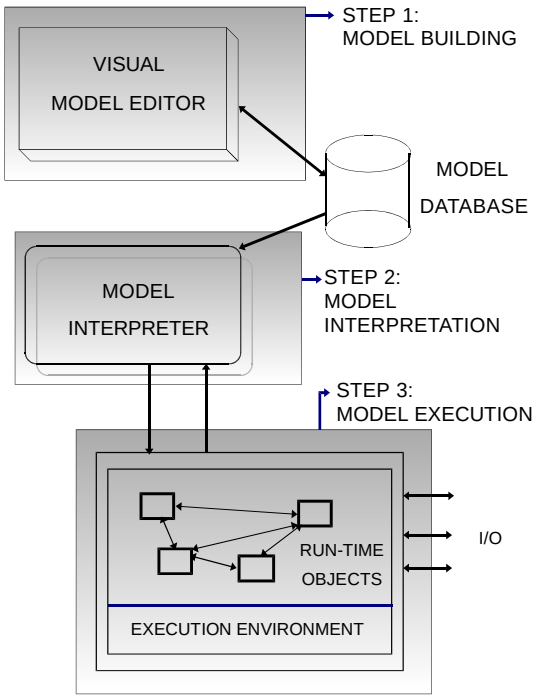

Figure 1: Model-based system development

A model-embedded problem solving environment has been implemented for chemical engineering applications, that system is described next.

3 Overview of the system

IPCS (short for Intelligent Process Control System) is a model-embedded problem solving environment for chemical engineering applications. The system is built on the framework provided by the Multigraph Architecture (MGA), described in a separate paper[8].

The system's purpose is to provide an environment for problem solving that can be used by chemical engineers working in plants in their various activities. The system is model-based, meaning that the users interact with the system via different kinds of models. Models are created using a *visual model builder*[4], which supports model editing in the form of diagrams. Some model attributes (e.g. equations) are entered textually, for compactness. Models are stored in a model database, which is an object-oriented database package. Optional *model validation tools* can analyze and check the models, if needed. Once models are cre-

Model-embedded On-line Problem Solving Environment for Chemical Engineering

Gabor Karsai, Janos Sztipanovits,Hubertus Franke*,
Samir Padalkar†,
Department of Electrical and Computer Engineering
Vanderbilt University
Nashville, TN 37235
Frank DeCaria
Old Hickory Plant,DuPont DeNemours
Old Hickory, TN 37216

Abstract

The building of custom monitoring, control, simulation and diagnostics applications for complex chemical plants necessitates the integration of models into the problem solving process. This paper describes a system and its practical applications that supports this activity. It is based on the Multigraph Architecture, which is a generic framework for building these model-based systems. The paper discusses the modeling paradigms used, how the applications are generated, and some practical, existing applications.

1 Introduction

Today's chemical plants are enormously complex systems, whose monitoring and control is a highly non-trivial task. This is especially true if higher level monitoring and control functionalities are required: while low-level loops are relatively easy to handle with traditional techniques, higher-level functionalities (like optimization) require fusion of information from different sources, and a high degree of system integration with, for instance, process simulators, optimization packages, etc.

In many cases, these higher level functionalities *evolve* with the plant, and they are created and maintained not by software engineers, but *plant* engineers who like to use their own language and terminology in formulating and solving problems. Note that the creation of these higher level functionalities is part of a larger *problem solving* process, where plant engineers need to solve specific problems related to the plant's operation. (E.g. "What is the effect of a rate change on the efficiency of the operation?"). Note also that these problem solving activity is highly *context sensitive*, i.e. the developed "solution" (i.e. the software) must be executed in the context provided by the entire plant.

We can conclude that the kind of problem solving activity mentioned above has two important characteristics: (1) it should be performed by domain-engineers, and (2) its is highly context sensitive. These problems are not easy to handle within the framework of traditional software engineering.

In this paper we present an approach and a practical system which addresses these issues and offers solutions to these problems. First we review other approaches, next we give a brief overview of the system, after which we describe two main components: the modeling paradigms used and the system generation process. Finally, some practical applications are discussed.

2 Background

Creating domain-specific problem solving tools can be approached from many, different directions. One can identify at least three distinct methods: (1) object oriented software design, (2) application generators, and (3) expert system environments.

The approach using object-oriented design can be summarized as follows: The user (plant engineer) should learn the design techniques of object-oriented programming, preferably with nice notation (e.g. Booch[2] or OMT[7]), and use these techniques to create solutions to the problems. This approach results in well-structured, high-quality computer solutions, but at a very high cost. Domain-engineers should be trained in OOD techniques, and they have to learn how to translate their "language" into that of OOD.

Application generators (e.g. Matrix-X) offer excellent solutions to a part of the problem. For example, they support the block-diagram oriented modeling (and building) of control systems. However, they do not offer ways for capturing the engineer's knowledge or physical data (e.g. size) about the plant. They are typically very easy to use and appealing, but they seem to lack the rich semantics required for domain-specific systems.

Expert system environments (e.g. G2[3]) support the knowledge-based approach: knowledge is encoded in the form of frames and rules, although procedural

*Currently with IBM TJ Watson Research Center
†Currently with Quant Trading, Inc.