The Design of DECmodel for Windows

by Stewart V. Hoover and Gary L. Kratkiewicz

ABSTRACT

The DECmodel for Windows software tool represents a significant advance in the development of business process models. The DECmodel tool allows rapid development of models and graphical representations of business processes by providing a laboratory environment for testing processes before propagating them into workflows. Such an approach can significantly reduce the risk associated with large investments in information technology. The DECmodel design incorporates knowledge-based, simulation, and graphical user interface technology on a PC platform based on the Microsoft Windows operating system. Unique to the design is the manner in which it separates the model of the business processes from the views or presentations of the model.

INTRODUCTION

Many approaches have been developed for understanding, specifying, testing, and validating business processes. In the late 1980s, Digital began to reengineer some of its most complex and mission-critical business processes. It soon became apparent that modeling methodologies and tools were needed to document, test, and validate the reengineered processes before they were implemented, as well as to provide a high-level specification for their design and implementation. Consequently, Digital decided to provide the business process engineer with tools similar to those used by architects, mechanical designers, and computer and software engineers.

The first implementation of Digital's dynamic business modeling technology, Symbolic Modeling, was developed at Digital's Artificial Intelligence Technology Center. The technology was embodied in an application called Symmod, which in 1991 ran only on a VAXstation system.[1] Symmod's knowledge base and simulation engine were implemented using the LISP programming language and the Knowledge Craft product, a frame-based knowledge representation package with modeling and simulation features.[2] Because models were written in LISP code, users had to be computer programmers as well as business consultants. The application contained a graphical presentation builder and viewer implemented in the C programming language that used a relational database for presentation storage. The user had to start the knowledge base component and the presentation component as separate processes. A primitive mailbox system was used for interprocess communication. To serve the needs of nontechnical business users and to achieve the necessary product quality, Symmod needed to be completely redesigned and rebuilt.

In early 1991, the Modeling and Visualization Group decided to build a product version of the Symmod application, which would be released as the DECmodel tool. The team drafted requirements, specifications, and an architecture. The DECmodel product was initially targeted at two platforms: VAXstation workstations running under the DECwindows operating system and personal computers (PCs) running under the Windows NT operating system. As users were interviewed and requirements were accumulated, it became clear, however, that by far the most important platform for DECmodel users was the PC platform based on the Windows operating system. Consequently, the DECmodel development effort shifted to this platform.

During 1991, the team enhanced the existing version of Symmod so that it would meet user needs until the release of the product version for PCs. The most significant enhancement was the development of an X Window System interface for building and editing models. A second important enhancement was a graphical shell program that transparently started up the knowledge base and presentation components for the user.

In March 1992, Digital officially announced Phase 0 (the strategy and requirements determination phase) of the DECmodel for Windows product.

DESIGN AND DEVELOPMENT GOALS

The DECmodel product design team had the following goals:

- Provide a modeling tool that maps directly to business processes

- Allow the modeling of both the static and the dynamic characteristics of the business process

- Allow multiple views of the business process model by separating the model from the presentation of the business process during simulation

- Allow the user to interact with the tool and to make decisions while the business process is being simulated in order to let the user "test-drive" the business process

- Provide a tool that is easy to use for business consultants and that requires no programming

Note that the designers intentionally omitted the following goals from the DECmodel design:

- Include resource constraints and queuing

- Allow the user to perform a statistical analysis of the behavior of the business process

By far the most important goal for the DECmodel design was the first one listed, an obvious mapping between elements of the model and business processes. The anticipated users of the DECmodel tool were business analysts and consultants, not system designers and software engineers. The designers felt that adding levels of abstractions to a modeling tool would make it less acceptable to the intended users. A notable corollary to providing an obvious mapping was modeling both the static and the dynamic characteristics of the business process.

To engage the user in interacting with the model and test-driving the business process required a graphical interface that was separate from the model. This "presentation" layer of the DECmodel tool provides a layout and graphical appearance that has the look and feel of the actual business process, hiding the irrelevant technical details of the model. The presentation enables the user to step through the business, watching information and material flows occur, and thus see where the dependencies and concurrencies exist.

Designers believed that while simulating the business, the user should be able to interact with the model and thereby select and test more than one scenario. The DECmodel tool was intended to be a working scale model of the business, giving the user a sense of how the business process would work as different choices were made. The tool, by design, neither predicts congestion and throughput as a function of resource constraints nor provides information through statistical reports. The DECmodel product was designed to provide a slow, deliberate simulation of the business, not to compress weeks or years of activities into a few seconds, leaving behind only a statistical summary.

The team's development goals for the DECmodel product were to

- Provide a tool that runs on a popular hardware platform used by business consultants

- Achieve a short time-to-market, i.e., delivery within one year

- Utilize a widely accepted software base technology (for maintainability)

THE DECmodel WORLD VIEW

Every modeling and simulation tool is based on a predefined view of the world.[3] In the DECmodel world view, a business process is composed of aggregate centers capable of performing one or more tasks or work steps. Each aggregation is referred to as a process, and the tasks that can occur in a process are called activities. Processes communicate through the exchange of messages, which are sent by activities and received by another process or other processes or by the same process that contains the activity.[4]

This view differs significantly from the one taken by the typical workflow model in which work steps are directly linked. In the DECmodel model, an activity that sends a message to a process has no knowledge of what work steps will occur next. For example, when a customer (a process) sends an order (a message) to a supplier (another process), the customer does not know what work steps (activities) the supplier will initiate when it receives the order. It is invisible to the customer whether or not the supplier decides to change its work rules, for instance, by sending the order to a second source because materials are not available. Similarly, when the supplier's activities have been completed and the material that was ordered has been sent to the customer, the supplier has neither knowledge of nor dependencies on the work steps that the customer undertakes next. In contrast, in a workflow model each task is directly linked to another task. Changes in the supplier's way of doing business force changes in how the customer's tasks connect to the supplier's tasks. More succinctly, the DECmodel tool encapsulates the behaviors and work rules of each individual process in the larger business process. This difference between the process and workflow models is shown in Figure 1.

[Figure 1 (The Process Model versus the Workflow Model) is not available in ASCII format.]

Processes, Activities, and Messages

As described above, the DECmodel model represents a business process as a collection of smaller encapsulated processes. The behavior of each process is defined by the activities that it contains. The DECmodel tool provides three general types of activities: generating activities, processing activities, and terminating activities. Generating and terminating activities represent the boundaries of the model; processing activities represent the work steps in the business process.

An activity is characterized by (1) a receive rule, which defines the messages that the activity needs for initiation, (2) a duration, and (3) a send rule, which defines the messages that the activity sends out at the end of its duration. Generating activities have only send rules, and terminating activities have only receive rules.

Activities can send messages to processes only. The receiving process makes the message known to every activity that uses the message in its receive rule. Messages are universal to the model, and the same message type can be sent by activities in different processes.

Processes can have state knowledge (attributes) that can be assigned values as a side effect of an activity being completed. The activity can use a process attribute value to decide what messages to send out and where to send them. That is, processes have a state that can be altered to change the behavior of the model.

Like processes, messages can contain information, which is stored in their attributes. When a process receives a message and passes it on to an activity, information in the message can be used in both the receive rule and the send rule of the activity. Additionally, the information in a received message can be copied into the attributes of any message that an activity sends. In this way, the DECmodel tool supports information propagation.

The DECmodel representation of business borrows heavily from both the stochastic-timed Petri net (STPN) model and the object paradigm found in object-oriented design.[5,6]

The Stochastic-timed Petri Net Model versus the DECmodel Model.

An STPN model represents a system as a collection of places, transitions, arcs, and tokens. Places contain tokens and act as inputs to transitions. A transition results in the movement of a token to another place if an arc exists between the transition and the place. Before a transition can occur, a token must be present at each place that is connected to the transition by an arc. Associated with each transition is an exponentially distributed random variable that expresses the delay between the enabling of the transition and the firing of the transition.

The DECmodel model welds the STPN place, transition, and arc elements into a single object called an activity. The analogous elements of the STPN and DECmodel models are

STPN	DECmodel
Place	Activity receive rule
Transition	Activity duration
Token	Message
Arc	Activity send rule
--	Process

The DECmodel model goes beyond the STPN model by

1. Adding the process object between the activity send rules (arcs) and the activity receive rules (places). Each process can have multiple activity send rules. As the process object receives messages (tokens), it dispatches them to the appropriate activity receive rule

(place).

2. Allowing more than one type of message (token) to exist.

3. Storing information in both the processes and the messages (tokens).

4. Using AND, OR, and message-matching receive rules in the activity receive rules (places).

5. Not restricting durations to being exponentially distributed random variables.

Like an STPN model, a DECmodel model does not explicitly have resources but can represent the availability of a resource by sending a message to a process when the resource is available.

Figure 2 shows the workflow system from Figure 1 as both an STPN model and a DECmodel model with the process receiving messages from the activities.

[Figure 2 (The Stochastic-timed Petri Net Model versus the DECmodel Process-activity Model) is not available in ASCII format.]

The DECmodel Model and Object-oriented Design. The elements of object-oriented design that the DECmodel model fully draws upon are encapsulation of information and the message-method paradigm. Information is encapsulated within DECmodel objects and is not available globally. However, an important difference exists between DECmodel systems and object-oriented systems. In DECmodel systems, a number of messages may by required to trigger a behavior; whereas, in classical object-oriented systems, each message triggers a method.

The DECmodel tool supports polymorphism, in that the same message can be sent to different processes, which can result in different behaviors. Developers investigated going beyond standard polymorphism by using one message to trigger different activities within the same process. The approach considered was to use process "filters" to examine the information in a message and then decide which activity or activities in the process should receive it. This feature was not completely developed because of time constraints and a less-than-clear mapping between the concept and the actual practices in most business. Further, using activity send rules that utilize the information contained in messages can provide a similar capability.

The DECmodel tool does not support inheritance, but the underlying technology of the product does support this feature. As in the case of nonstandard polymorphism, time-to-market pressures and the lack of clear evidence that the feature would be used in business processes drove the decision not to include inheritance support. Also, the DECmodel product does not currently support class types beyond the built-in classes of the process and the three activity types.

Process Hierarchies

To address the goal of having a strong mapping between the model and real business processes, the DECmodel model supports processes within processes. Processes can receive messages in two ways: hierarchical routing and peer-to-peer routing.

In a business process, a message sent to a high-level process should travel through the process hierarchy to the activity that is to act upon the message. For example, an activity in the sales process should be able to send a message to the manufacturing process and not be concerned that manufacturing contains several subprocesses. The knowledge of how to relay a message should be in the receiving process, not the sending process.

In business, however, much communication occurs on a peer-to-peer basis, with information seldom routed up and down the organization hierarchy. For example, the results of a marketing research activity go directly to the manufacturing planning function without traveling down through the various levels of the manufacturing organization. In a DECmodel model, as in most businesses, when an activity is completed, a message can be sent directly to any process in the business.

The DECmodel design feature that allows processes to receive messages and then pass them on to subprocesses and activities can result in multiple message receipts for a single send operation. That is, one activity can send a single message that is received by every activity in the model that includes the message in its receive rule. Modeling experts disagree about how well this phenomenon maps to real business processes. The DECmodel user can avoid this effect, if desired, by using uniquely named messages in the send rules of activities.

The Presentation

The first DECmodel design goal was supported by the modeling paradigm of processes, activities, and messages. The presentation aspect of the DECmodel tool supports the goals of a strong separation between the model and the graphical representation of the business process and the need to support user interaction and decisions during model simulation.

The presentation of the model is based on views that contain networked nodes. Each node in a view can represent zero or more processes in the model; however, no process can be represented by more than one node in a single view. This mapping between the processes in the model and the nodes in a view allows the user to develop and animate multiple views of the model simultaneously. For example, one view may show the model at its lowest level of detail, with each process in the model mapped to a single node. Another view may show a higher level of mapping, with multiple processes mapped to the same node. A third view may map processes based on attributes such as geographic location, the organizational chart, or technology. The construction of the views is left to the creativity of the analyst building the model.

During model simulation, the DECmodel tool uses animation to show the movement of messages from one process to another. The user can also view the messages and their attributes.

To accommodate user interaction, the DECmodel tool provides a menu send rule in the definition of an activity. If an activity uses the menu send rule, just before the activity fires, a menu appears that allows the user to make a choice that determines what messages are to be sent by the activity and which processes are to receive them. The user is unaware of the actual send rule; the choice made forces one of a set of send rules to be selected. The use of menus, animation of messages moving between processes, and user-controlled stepping through the simulation gives the user the feeling of test-driving the business process.

ARCHITECTURE AND DEVELOPMENT PROCESS

The overall DECmodel architecture, shown in Figure 3, contains two layers. The inner layer of the architecture is the internal engine, which provides the means for representing, storing, and executing (simulating) models. The internal engine is designed using ROCK, a frame-based, object-oriented knowledge representation system, and AMP, a modeling and simulation frame-class library implemented in ROCK.[7] The outer layer of the architecture is the user interface, which provides the means for all user interaction with the DECmodel model and has two major components: the model builder and the presentation builder. The user interface is designed as a set of classes specialized from the Microsoft Foundation Classes. Interaction between the two layers is achieved with an internal application programming interface (API).

[Figure 3 (DECmodel Architecture) is not available in ASCII format.]

This architecture was chosen for both technical and pragmatic organizational reasons. The partitioning into two layers allowed the internal engine to be built using state-of-the-art knowledge representation technology and the user interface to be built using state-of-the-art graphical user interface technology. The disadvantages in this separation were the necessity of designing an internal API and the need to duplicate some data (nominally stored in the knowledge base) in the user interface.

The partitioning mapped well to the human resources available in the DECmodel team. The DECmodel engineers had strong skills in developing LISP, knowledge-based, and X Window System applications but little experience in developing C++, ROCK, or Microsoft Windows applications. With the architectural separation, one team was able to focus on the internal engine using C++ and ROCK and, therefore, did not have to learn much about Windows programming. The other team was able to focus on the user interface using C++ and Windows programming tools and did not have to learn anything about ROCK. The engineering team felt that the efficient use of human resources in the development process overcame the technical disadvantages of the approach.

DECmodel development proceeded with the two teams. Since the bulk of their development work was completed first, the members of the knowledge base team also worked on the user interface team toward the end of the development process.

DESIGN AND IMPLEMENTATION

This section describes the design of the two DECmodel layers: the internal engine and the user interface.

Internal Engine

The internal engine represents models of dynamic business processes in a knowledge base and executes these models using discrete event simulation. This layer provides a set of services for interacting with the knowledge base. These services are accessed through the DECmodel tool's internal API. The internal engine contains the DECmodel knowledge base, simulation engine, and means of persistent storage. Using the DECmodel methodology to represent and execute business process models, the internal engine

- o Represents the structure, attributes, and behavior descriptions of the business processes in a knowledge base. (This representation is the model.)

- o Represents the structure, attributes, and behavior descriptions of the animated visualization of the model in a knowledge base. (This representation is the presentation.)

- o Represents the connections between the model and the presentation in a knowledge base. (This representation is the model-presentation mapping.)

- o Represents the dynamic behavior of the business processes by allowing for discrete event simulation of the knowledge base.

Knowledge Base. The DECmodel knowledge base contains the frame-based, object-oriented representation of the model, the presentation, and the connections between them. It also maintains the model relations, attributes, and methods. The knowledge base contains both classes and instances. The classes specify DECmodel objects; sets of instances make up specific models and presentations. In addition to containing all the information about model and presentation behavior and structure, the knowledge base contains all the graphical information used by the model builder and the presentation builder. This information is updated in real time.

Knowledge Representation Technology. The DECmodel knowledge base and simulation engine are implemented in ROCK, a frame-based, object-oriented knowledge representation system written in the C++ programming language. ROCK implements the IMKA knowledge representation technology and is used as a set of API functions in a C++ programming environment.

ROCK provides useful features such as frames, multiple inheritance of data and methods, user-defined relationships, and contexts. The basic unit of knowledge in ROCK is a frame, which represents an object or a concept. A frame is a collection of slots that contain the attribute, relationship, and procedural information about the object or the concept. Attribute slots store values, relation slots store user-defined links between frames, and message slots store methods (functions) that are executed when the frame receives the appropriate message from the application program. Class frames represent object types or categories. Instance frames represent particular members of a class. ROCK requires frame classes to be organized in a class hierarchy. Attribute slots and message slots can inherit values and methods from classes at a higher level in the hierarchy. This mechanism can be used to define default values for frame classes. Both frame classes and frame instances are objects, and both can be dynamically created, operated on, and deleted during run time. With respect to the C++ language, all frames appear to have the same data type. Slots are objects, whose behavior is defined independent of the frames.

Portions of the knowledge base are built using AMP, a modeling and simulation frame-class library implemented in ROCK. AMP contains objects for representing process models that contain entity flow, for constructing and running discrete-event simulations, and for generating, collecting, and reducing statistical data.

The DECmodel frame classes are subclasses of ROCK and AMP classes and contain relations, attributes, and methods that describe the content and behavior of DECmodel objects. Some DECmodel frame classes are abstract classes used only as a basis for more specific subclasses; others are used for instantiation of DECmodel objects. The DECmodel tool contains three types of frame classes: model objects, presentation objects, and project objects. A specific DECmodel project is represented within the knowledge base as a set of model, presentation, and project instances. These instances are created in the knowledge base by loading a DECmodel modeling language (DML) file or through interaction with the model builder or the presentation builder.

Persistent Storage. The DML is a subset of the ROCK frame definition language and is used by the knowledge base for persistent storage. A DECmodel project is stored as ASCII text in three files that contain the model, presentation, and mapping objects. The language employs ROCK syntax but uses only the frame classes and slots defined in the DECmodel knowledge base.

The DECmodel tool utilizes the ROCK frame definition interpreter as the DML interpreter. Since the ROCK interpreter was not intended to be used for persistent storage, the DECmodel developers made several minor modifications to obtain the desired error handling capabilities. The DECmodel tool contains its own DML code generator.

Simulation Engine. The simulation engine executes a discrete event simulation of the model in the knowledge base. This simulation can be performed either interactively or in a batch mode. The simulation engine was designed to be so robust that a model can be simulated at any stage of its development, regardless of inconsistencies or undefined elements.

The simulation engine interacts with the presentation builder to control simulation, animation, and graphics. The user controls simulation through the presentation builder. The presentation builder calls simulation engine API functions to perform the requested actions, such as starting, stepping through, pausing, ending, and reinitializing the simulation.

Script Engine and Compiler. Scripts provide a means of specifying user-defined actions to customize model animation and to perform special presentation actions during simulation. The DECmodel tool contains a language for defining scripts, a script compiler, and a script engine for executing the scripts. Although the DECmodel team wanted to avoid requiring any programming in the tool, developers decided that a script language was the only way to implement these features in the available time frame.

The script language contains functions for

- o   Annotating, hiding, showing, flashing, moving, highlighting, and scaling presentation icons

- Playing sounds and sound loops
- Animating connections between nodes
- Showing, hiding, and clearing certain kinds of windows
- Starting other applications
- Temporarily stopping execution
- Loading a new project
- Starting and pausing the simulation
- Displaying files
- Displaying a list of DECmodel development team members

Analysis and Reporting Services. The knowledge base contains services that allow the user to analyze models and presentations in the knowledge base and to generate reports.

The consistency advisor checks models, presentations, and mappings for inconsistencies and potential problems at any point in the model development process. This check is analogous to the syntax check performed by a compiler. The consistency advisor check is the primary model-building debugging aid for users. Inconsistencies in the model do not prevent a model from being simulated.

The model description report lists the description, messages sent, and messages received for each activity and process. The model table report contains the basic model information in a table format for easy access by another application, database, or spreadsheet. The simulation summary report contains information on simulation performance.

Design and Implementation Decisions. The internal engine for the first DECmodel product release, DECmodel for Windows version 1.0, was implemented as a Windows dynamic link library (DLL) using the Windows version of ROCK version 1.0, the Windows version of AMP version 1.0, and Microsoft C/C++ version 7.0. For DECmodel for Windows version 1.1, developers ported the internal engine to Microsoft Visual C++ version 1.0.

Several options existed for implementing the DECmodel knowledge base. The knowledge base of the Symmod application, the precursor to the DECmodel product, was implemented in a LISP environment. The DECmodel engineering team wanted to move to a more standard programming environment and, therefore, focused on C++ and C++-based tools. However, a straight C++ implementation would have required the reimplementation of knowledge representation, simulation, and modeling technology available in other tools.

Another modeling and simulation technology, the Modeling and Simulation System (MSS), had been developed for Digital's Artificial Intelligence Technology Center by the Carnegie Group, Inc. (CGI).[8] This graphical tool was designed at a lower level than Symmod. It used a modeling simulation language and was developed to implement the next version of Symmod. However, the MSS modeling paradigm was not compatible with that of the DECmodel tool.

IMKA had also been recently developed by CGI, funded by a consortium of companies, as a replacement for the Knowledge Craft product. IMKA's implementation, ROCK, lacked some of the class libraries included in Knowledge Craft for simulation and process modeling but ran significantly faster than Knowledge Craft. The engineering team decided to use ROCK to implement the knowledge base because of its knowledge representation power and its C++ compatibility. Digital contracted with CGI to port the class libraries to ROCK. The team, therefore, had a head start in designing and implementing the internal engine. The portability of ROCK was also an advantage; switching to the Windows platform from the DECwindows platform caused no disruption in development.

The original intent of the engineering team was to implement the DECmodel tool as a single executable file. The knowledge base contains much global data, however, and restrictions on the number of data segments required developers to implement the internal engine as a DLL. This encapsulation of the internal engine allows it to be used in other applications and enables easy porting to other platforms. The DECmodel team developed a set of internal API functions and structures to allow interactions between the DLL-based internal engine and the executable-based user interface.

The Symmod application had a modeling language based on LISP for persistent storage of models and used a relational database for persistent storage of presentations. Consideration was given to developing a modeling language specific to the DECmodel tool. Instead, the engineering team decided to use the ROCK frame definition language, since it was already completely defined and debugged and had an interpreter. The team used this language for persistent storage of both models and presentations to allow easy sharing of projects between users and to simplify debugging by users and DECmodel developers.

The knowledge base team was responsible for implementing the internal API between the user interface and the knowledge base. This interface was specified in detail early in the project. The team kept the specification up-to-date throughout the project. It prepared 19 revisions and produced a final document of more than 200 pages. This specification kept interface problems to a minimum, thus defusing a potential source of major technical problems.

The team specified the objects in great detail early in the project. It also held several internal and external design reviews. These measures reduced the number of potential design problems and thus yielded a higher-quality product and a faster implementation.

User Interface

The user interface provides the means for all user interaction with the DECmodel tool. It has two major components: the model builder and the presentation builder.

The user interface is designed as a set of classes specialized from the Microsoft Foundation Classes. Most of these special DECmodel user interface classes correspond to frame classes in the knowledge base; the remainder are necessary for implementing the user interface. The three main types of user interface classes -- windows, graphic objects, and dialog boxes -- are used by both the model builder and the presentation builder.

Window Classes. The user interface contains several types of window classes: graphics windows, text windows, and a frame window.

The graphics window classes are all derived from the generic DECmodel graphics window class. Graphics windows contain graphic objects, such as boxes or lines. Users act upon these windows through menu commands or through the Windows messages generated by the mouse and mouse buttons. The graphics windows are the model window, the view windows, and the palettes. Menu commands specific to each graphics window are handled by message handlers within the window class.

The text window classes are derived from the generic DECmodel text window class. Text windows are generally read-only and display various types of textual information, such as descriptions, the text of files, and clock information. As in the case of graphics windows, menu commands specific to each text window are handled by message handlers within the window class.

The one frame window class, i.e., the top window class, is derived from the CMDIFrameWnd Microsoft Foundation Class and serves as the frame window for the application. The menu commands not specific to a particular window are handled by default message handlers within this window.

Graphics Classes. Graphics window classes use graphic objects to build models and presentations. These classes implement the processes, activities, nodes, connections, and annotations displayed in the Model Editing Window and in the views.

Dialog Box Classes. The DECmodel tool contains a large number of dialog boxes derived from the CModalDialog Microsoft Foundation Class. The tool uses these dialog boxes to define the information and relationships contained in the DECmodel objects.

Menus. The DECmodel tool uses a set of menus individualized to match the capabilities of the window currently in use. When a user starts the DECmodel application, the tool presents a reduced menu that allows the user to start a new project or to load an existing one. Once a project is in memory, the menu changes as the user switches between the Model Editing Window, the views, and the other windows. Menu commands activate message handler functions within the window classes.

Appearance of the User Interface. Figure 4 shows a small but typical DECmodel model. The figure displays each process and its member activities. Note that each of the three activity types is denoted by a different icon. Lines indicate the potential flow of messages. Figure 5 shows the DECmodel presentation for the model that appears in Figure 4. The presentation contains both a view and the supporting windows, e.g., the simulation clock and the description windows.

[Figure 4 (Typical DECmodel Model) is not available in ASCII format.]

[Figure 5 (Typical DECmodel Presentation) is not available in ASCII format.]

Design and Implementation Decisions. The team implemented the user interface for DECmodel for Windows version 1.0 using Microsoft C/C++ version 7.0 and Microsoft Foundation Classes version 1.0. For DECmodel for Windows version 1.1, developers ported the user interface to Microsoft Visual C++ version 1.0 and Microsoft Foundation Classes version 1.5.

As stated at the beginning of the paper, the DECmodel product was initially targeted at both VAXstation workstations running under the DECwindows operating system and PCs running under the Windows NT operating system. Consequently, when developers decided to focus solely on the PC platform running under the standard Windows operating system, the user interface development effort was disrupted. Engineers had done a significant amount of design work toward achieving a DECwindows implementation.

The DECmodel engineering team considered other class libraries and user interface implementation packages (such as XVT), but most were deficient in Windows features or in the look and feel. Since the Windows operating system was the only platform for the foreseeable future, the engineering team felt that using Microsoft Foundation Classes was the best choice. However, they made this decision after they had performed a significant amount of development work with one of the tools. Much of the work had to be redone, which contributed to the schedule delay.

During the design and development of the DECmodel product, the team debated how graphical to make the user interface, that is, to what extent dialog boxes should be used. Although the goal was to make the user interface as graphical as possible, the tight schedule forced the team to postpone plans for graphical editors in favor of dialog boxes, which were faster to implement. For example, the team had initially planned to implement an Activity Editing Window and had partially developed it. This window was to provide a complete view of an activity and allow graphical editing of its information. Schedule constraints required the team to abandon this plan and to develop a set of dialog boxes that were not as easy to use but were faster to implement.

The user interface design was not specified or committed to storyboards in any detail at the beginning of the project, partially to save time after the disruptions in the development work. This decision led to more lost time later in the project, though, because user interface features were designed quickly and sometimes incompatibly, and consequently required reworking. In addition, the resulting user interface was not as easy to use as it could have been if better planned.

External review of the user interface design was not performed until late in the project. The review yielded some ideas that would have resulted in a more usable product; however, there was not enough time left in the schedule to implement them.

DELIVERY

A discussion of the released product and the team's success in achieving the design and development goals follows.

Release

Digital released version 1.0 of the DECmodel for Windows product in November 1993 and version 1.1 in April 1994. Version 1.0 contained the basic capabilities for building models and presentations of business processes; version 1.1 added a set of minor enhancements and bug fixes. Because of its small, focused market and the large cost savings that can result from its use, the DECmodel tool was introduced as a low-volume, high-priced product. The product includes the software, example models, documentation, and a week of hands-on training. The DECmodel tool is an integral part of Digital Consulting's reengineering practice.

Success of Design Choices

The separation of the model from the presentation is the single most important element of the product's success. This feature, along with animation, distinguishes the DECmodel tool from its competition. Some users have even requested the capability of building the presentation first and then generating the corresponding model. Such capability would require considerable investigation.

The paradigm of process-activity encapsulation is difficult for some users to become accustomed to. Many still prefer to build a model using a workflow approach, which the DECmodel tool can support, rather than by defining each process and its behavior independently.

The exclusion of resource constraints has limited the application of the DECmodel tool to system design, thus preventing its use in modeling system performance. Although the capability was originally not a product goal, many users would like a future version of the DECmodel product to provide this feature.

To perform special user-defined actions during the simulation, a script language was included in the DECmodel tool. This design feature violated the goal of requiring no programming, and some users found scripts hard to use. However, many users have requested that a future DECmodel version provide more script functions and extend the script language to be more like the BASIC programming language.

Also, to enhance the use of the DECmodel tool in the design of business processes, a future version should support classes to make generic processes available as building blocks of a business process.

Development Successes and Lessons

The DECmodel engineering team successfully released a software product on the Microsoft Windows platform, the one most popular with business consultants. This achievement was significant because the group of engineers began the project with no PC experience. The team did not meet its one-year delivery goal, and the goal slipped to one and one-half years after the Phase 0 announcement. However, this time frame was still extremely short for developing a complex PC product from scratch.

The product retained the existing Symbolic Modeling paradigm (i.e., a process-activity-message model and a strong distinction between model and presentation) and exhibited performance an order of magnitude better than that of the Symmod product, which it replaced. The product utilized the most widely accepted modern programming technology base (C/C++), which simplified maintainability and reduced the need for special training of maintainers.

Splitting the development team into two subteams worked well. It distributed the amount of learning about new technologies required by the engineers and minimized the overall development time. Key factors in the success of this approach were the detailed object and internal API specifications that were kept up-to-date throughout development and thus provided a reliable interface between the two parts of the project.

After the product was released, the DECmodel team identified certain factors that could have made the team and the product even more successful. The entire engineering team would have benefited from Windows training at the onset of the project. The Windows design of the user interface should have been specified and committed to storyboard in much greater detail much earlier in the project. In addition, the team should have arranged for Windows experts to review the design. These changes in the engineering process would have helped the team produce a cleaner, easier-to-use, more maintainable user interface and would have reduced implementation time. The project schedule should have been created using a bottom-up rather than a top-down process. The initial one-year schedule was based on an unrealistic, management-imposed release date. When the engineering team revised the schedule and calculated a release date based on their detailed estimates, the team met the new date.

SUMMARY

Modeling and simulating business processes is an important part of business process reengineering. Digital developed the DECmodel tool specifically for this type of simulation. Although it borrows many ideas from other disciplines of modeling and simulation, as well as from object-oriented design, the DECmodel product is unique in the way it models business processes, separates the model from the presentation, and represents the model as frames in a knowledge base.

ACKNOWLEDGMENTS

The authors would like to acknowledge the following people who also contributed to the design of the DECmodel product: Ty Chaney, David Choi, Laurel Drummond, Peter Floss, Amal Kassatly, Mike Kiskiel, Kip Landingham, and Janet Rothstein.

REFERENCES

1. Symmod User's Guide (Maynard, MA: Digital Equipment Corporation, 1990).

2. Knowledge Craft Reference Manual (Pittsburgh, PA: Carnegie Group, 1988).

3. S. Hoover and R. Perry, Simulation, A Problem Solving Approach (Reading, MA: Addison-Wesley, 1989).

4. DECmodel for Windows: Modeler's Guide (Maynard, MA: Digital Equipment Corporation, 1994).

5. J. Peterson, Petri Net Theory and Modeling of Systems (Englewood Cliffs, NJ: Prentice-Hall, 1981).

6. G. Booch, Object Oriented Design (Redwood City, CA: Benjamin-Cummings, 1991).

7. ROCK Software Functional Specification, Version 2.0 (Pittsburgh, PA: Carnegie Group, 1991).

8. Modeling and Simulation System User's Guide (Pittsburgh, PA: Carnegie Group, 1991).

BIOGRAPHIES

Stewart V. Hoover Employed at Digital Equipment Corporation between 1984 and 1994, Stew Hoover is currently an independent consultant specializing in modeling and simulation. Before joining Digital, he was an associate professor of industrial engineering and information systems at Northeastern University. Stew contributed to the development of DECalc-PLUS, Statistical Process Control Software (SPCS), and the DECwindows version of Symmod. He has written many papers and articles on simulation and is coauthor of Simulation, A Problem-Solving Approach, published by Addison-Wesley in 1989.

Gary L. Kratkiewicz Gary Kratkiewicz is currently a scientist in the Intelligent Systems R&D Group at Bolt Beranek and Newman Inc. As a principal engineer in Digital's DECmodel engineering group from 1991 to 1994, Gary coordinated the architecture and high-level design specifications, and developed the knowledge base, script engine, API, and several user interface modules. Earlier at Digital, he developed an expert system for shipping and was project leader for a knowledge-based logistics system. Gary holds an S.B.M.E. from MIT and an M.S. in manufacturing systems engineering from Stanford University.

TRADEMARKS