Managing business processes as an information resource

by F. Leymann
W. Altenhuber

The relevance of business processes as a major asset of an enterprise is more and more accepted: Business processes prescribe the way in which the resources of an enterprise are used, i.e., they describe how an enterprise will achieve its business goals. Organizations typically prescribe how business processes have to be performed, and they seek information technology that supports these processes. We describe a system that supports the two fundamental aspects of business process management, namely the modeling of processes and their execution. The meta-model of our system deals with models of business processes as weighted, colored, directed graphs of activities; execution is performed by navigation through the graphs according to a well-defined set of rules. The architecture consists of a distributed system with a client/server structure, and stores its data in an object-oriented database system.

Organizations typically prescribe how business processes have to be performed, especially those processes that represent complex routine work, that involve many persons (both concurrently and sequentially), and that are in general frequently performed. Examples of such processes are manifold: program development in the software business, credit allocation in the banking business, customer enrollment in the health insurance business, or expense allowance processing in the administration business.

The relevance of business processes as a major asset of an enterprise is being accepted more and more. Business processes prescribe the way in which the resources (e.g., data, capital, human beings) of an enterprise are used, i.e., they describe how an enterprise will achieve its business goals. The quality of the business processes will influence the quality of the performance of an enterprise. Thus, business processes themselves represent important information resources of an enterprise, and techniques or systems to manage and support business processes are always in demand.

The IBM program product called FlowMark* supports the management of business processes. Both fundamental aspects of process management, namely the modeling of processes (build time) and the execution of processes according to a process model (run time), are facilitated. FlowMark may be perceived especially as a repository for business processes. Within the IBM Information Warehouse* framework (of which some aspects are discussed elsewhere in this issue), FlowMark is positioned as the work flow management component.

Current approaches. Today, there is no generally accepted methodology for modeling business processes. Petri nets are traditionally used to describe and analyze concurrent systems.[1] Nevertheless, it has been recognized that Petri nets are

©**Copyright** 1994 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

not currently succinct and manageable enough to become useful in modeling business processes.[2] For this reason high-level Petri nets have been intensively studied during the last couple of years. In particular, predicate/transition nets[3] and colored Petri nets[4] have been applied in various application areas. But other methodologies have also been proposed and applied. It depends on the emphasis one puts on the usage of processes as to whether they are described as Petri nets, trigger systems, forms, case plans, or even collections of formulas of temporal logic, for example.

If one focuses on the pure data manipulation aspects of a process, process models are viewed as vehicles for ensuring database integrity. Guyot[5] shows that Petri nets are allowing database administrators to control and constrain the execution of activities that manipulate a database. Temporal logic has proved to be remarkably successful in describing parallel programs and in studying their properties;[6] the management of parallel components of a program has some similarities to managing transactions concurrently accessing databases. Thus, Lipeck and Saake[7,8] discuss how temporal logic is applied to describe valid sequences of database states and consistency-preserving transactions, which is in certain situations the major intent of a process model. Partial orders on event spaces are also considered to model consistency-preserving sequences of database actions.[9]

Process models may also be perceived as a means to extend and complement facilities known from conventional transaction processing monitors. A process model is viewed as the specification of the flow of control and the flow of data separate from the collection of routines performing the proper computations of an individual application. Applications are represented in Garcia-Molina and Salem[10] simply as sequences of related transactions ensuring either the successful execution of all transactions in the chain or its compensation. Predicate/transition nets are pursued in Wächter and Reuter[11] to model networks of "steps" (and "compensation steps") that represent the "scripts" of an application. A network of "activities," which are triggered by "events," which are sent along "arcs" connecting activities, is exploited in Hsu et al.[12] for that purpose. A very generic and abstract approach ("spheres of control") especially for managing flows, including their recovery, is presented in Davis.[13] Targeting in the modeling of long transactions, Kotz[9] proposes the use of event/trigger systems.

With the support of office work in mind the following has been suggested: To model office procedures, Behrmann-Poitiers and Edelmann[14] are proposing "case plans." In cases where the office system reveals specific object-oriented characteristics, life-cycle diagrams and composed activities are pursued.[15] If a process can be described as the processing of a form, a corresponding proposal is given by Tsichritzis.[16]

The commonality among processes in the areas of software development,[17–19] office work, and administration has been worked out by Chroust and Leymann.[20] A methodology applicable in these areas is described by Leymann.[21] It strives especially for a formalization of processes and their models in these problem areas that is even more succinct and "user-friendly" than in Genrich[3] and Jensen.[4] It encompasses, for example, the case plans of Behrmann-Poitiers and Edelmann[14] and in addition allows the definition of parameter-controlled work flows in their problem domain. In this paper the methodology of Leymann[21] is enhanced by introducing *PM-graphs* (Process Model graphs) in order to fulfill additional requirements posed by FlowMark.

Our presentation. First, we show that today process models are treated as information resources in a rudimentary manner; also, we sketch the potential embedding of process models into an IRDS (information resource dictionary system[22]). Next, we discuss the meta-model of FlowMark for processes. We provide and motivate a collection of constructs that have to be used in order to define the model of a process to FlowMark. Then, the architecture of FlowMark is sketched. It is a distributed system with a client/server system structure. All relevant data are stored in an object-oriented database system. The graphical end-user interfaces for defining and executing processes are sketched. Activities represented by executables complying to a certain invocation paradigm are invoked by FlowMark. In the final section, we give a mathematical formulation of the syntax and the semantics of the meta-model. Mathematically, a process model is represented as a special weighted, colored, directed graph of activities (called a PM-graph); the semantics of the meta-model are defined operationally, executing a process as an instance of a process model by navigating through the PM-graph according to a well-defined set of rules.

Process models as information resources

Along with the classical production factors of land, labor, and capital, enterprises are considering information more and more as an important resource, i.e., as one of their assets. This information includes data about all resources needed to reach the goal of the enterprise. It is generally accepted that this information should be represented in a formal manner as much as possible. The collection of actions needed to achieve this goal is referred to as "enterprise modeling."

Enterprise models. The conceptual base for enterprise modeling is sometimes called a *hypersemantic data model.*[23] Producing the model of a concrete enterprise by using a hypersemantic data model results in an *enterprise model*. Such an enterprise model consists of two components: the *data model* and the *knowledge model.*[23]

The data model describes the structure of all resources of the enterprise and is thus somewhat like the syntactical component of the enterprise model; in this sense, the data model describes *what* can be used by the enterprise to reach its goal. Today, enterprises are building data models based on *semantic data models* (for example, the entity/relationship model; for an overview of different semantic data models see Peckham and Maryanski[24]).

The knowledge model describes the use of resources and their connections; it is the semantical component of the enterprise model. In this sense, the knowledge model describes *how* the enterprise uses its resources in order to reach its goal. The knowledge model encompasses constraints, heuristics, and procedures. Constraints define the local and global consistency of the resources; if the resources are stored in a database, constraints fix the valid database states. Heuristics describe how to derive data. Procedures define events and correlated actions, set sequences of actions, and describe business processes. The model of a business process can be used to define, for example, valid sequences of state transitions in a database (intertransaction integrity[7,8,11]) as well as sequences of work steps, whereas executing a business process could be described as context-dependent.[17,20]

Documents as process models. Process models represent knowledge about an enterprise. Usually, this knowledge is found today in the format

The knowledge model describes the use of resources and their connections.

of textual processing instructions. The handling of each single business process according to these processing instructions then corresponds to a process, i.e., an instance of the process model as defined via the instructions.

Process models in the format of textual processing instructions are very inflexible. Enterprises cannot react with sufficient speed to changes in their environment. Changes in the processing instructions are communicated by distributing documents, thus, the time to activate these changes is dependent on when the corresponding documents arrive; in distributed organizations these documents will arrive at different points in time, resulting, in turn, in consistency problems. Support for handling individual processes is only enabled in a limited way via the textual processing instructions. Precise compliance with the instructions cannot be enforced directly.

Control programs as process models. Today, computer assistance for handling processes is achieved, for example, by programming the corresponding procedural instructions. Each single work step can be supported via "generic programs" (i.e., via tools or service routines), via special applications (i.e., via programs considering the individual needs of an enterprise), or simply via help texts (i.e., via electronically documented work instructions). A special control program determines the individual sequence of work steps dependent on the concrete context of the individual business process. In this sense, the control program represents the formalized process model, and an instance of the control program corresponds to a concrete, individual process. But such control programs also do not allow

A process model may be seen as a template for a class of similar business processes performed within an enterprise.

for highly dynamic reactions. Changing the knowledge embedded in the control program involves a great deal of effort (redesign, coding, compiling, etc.).

Separate representations for process models. For this reason, extracting the knowledge representing the process from the control program and forming a separate representation of this knowledge as its own syntactical unit is extremely desirable. As a consequence, these syntactical units—which now represent the process model—have the flexibility and comprehensibility to institute the required dynamics. A process interpreter receiving such a process model as input (along with other information) can instantiate the process model and determine the individual sequence of work steps, depending on the context of the instantiation of the process model.[20]

IRDS and process models. Process *models* describe (apparently) different functions such as the production of a part in a production line, the settlement of a damage case within an insurance company, the treatment of a form for making allowances for expenses, the procedure in developing a program, or valid sequences of transactions. In these situations, each individually executing process can be perceived as a separate *instance* of the process model.[20] The process model is the processing instruction for a concrete process to be executed (and is thus a processing model). Computer assistance thus means both the support for defining the process model (*modeling*) and the support for performing each individual process (*execution*).

Computer assistance for defining process models should be enabled via a language that provides constructs that can be embedded canonically in a dictionary. A dictionary concept that strives for integrating data models and process models is pursued and thus contains *all* information resources. It follows the International Organization for Standardization (ISO) conceptualization principle,[25] according to which as much knowledge about an application area as possible is moved from the programs to the dictionary of an enterprise. Such a language then encompasses a model for process models, i.e., a *meta-model* for processes; instances of the meta model are process models. In turn, the instances of the process models are the proper representations of process executions. Meta-model and process model, and process model and process, respectively, are building intension-, extension-pairs[26] that thus comply conceptually to the ISO Dictionary Architecture IRDS.[22] The integration of process models into the IRDS together with the already available data modeling capabilities then allows very flexible enterprise modeling based on a dictionary. For that purpose one has to describe our meta-model in terms of the fundamental modeling language of the IRDS.

FlowMark allows process models to be defined according to our meta-model described below. Each process model is an instance of the meta model. Process models are instantiated and executed by interpreting the instances of the types of the meta-model. The interpretation is performed by navigating through each individual instance (process) in accordance with the underlying process model.

The meta-model

A process model may be seen as a template for a class of similar business processes performed within an enterprise. It is a schema describing all possible variants of the (dynamic!) execution of a particular kind of business process. Each individual process is an instance of a process model, and it represents a concrete, specific execution of a variant prescribed by the process model.

The fundamental building block of the meta-model is the *activity*. An activity represents a business action that is a semantical unit at a certain phase of modeling effort. It might have a fine-structure, which is then represented in turn via a process model, or the details of it might not be of interest at all from a modeling perspective. It is important to note that these fine-structures allow both a bottom-up and a top-down approach to process modeling.

As an example, suppose a process model for credit allocation contains an activity called Solvency. For the modeler of credit allocation it is not of interest *how* Solvency is checked, but rather to make sure that this check *will* take place. The refinement of the activity Solvency as a process model again (if required) can be done (or may have already been done) by a different modeler.

In general, the work represented by an activity produces results. Within the meta-model the types of results of this work are associated with the activity as parameter types. Now, activities generally access types of results of other activities, or require information about the context of the current activity; such parameter types can also be associated with an activity. In general, an activity is associated with both *input parameter types* and *output parameter types* (in cases in which no misunderstanding will occur, the suffix "type" is omitted).

The collection of all input parameters of an activity is referred to as the *input container* of that activity, and the collection of all of its output parameters is referred to as the *output container*. Since process models may serve as fine-structures of activities, each process model itself is associated with both an input container and an output container; note that the input or output container of a process provides some sort of "global context" for all activities contained within this process. A concrete execution of an activity (also called an *activity instance*) is thus accessing the instances of the input parameter types from its input container and will produce instances of the output parameter types from its output container. Because of this, activities are considered to be mathematical maps.

In practice, only the "process-relevant" parameters of an activity are explicitly defined (i.e., externalized) rather than all parameters affected by an execution of the activity. For example, an activity generally modifies data that are not defined in its associated containers because these data are not of interest to other activities within the process; or an activity might obtain (additional) input from sources different from its input container (e.g., database reads).

As a result, it is pragmatic to recommend capturing an activity as a *relation* between its input container and its output container (for example, because additional input as mentioned before might result in nondeterministic behavior of the activity with respect to its input container). In fact, choosing whether activities are "maps" instead of "relations" is not crucial to our meta-model, and the model could be easily made to accommodate a choice. Nevertheless, for simplicity we treat activities as maps because we consider this treatment to be more suited to the perception of process modelers.

In general, the activities of a process may not be executed in an arbitrary manner. Some activities are necessary for a process to start, some activities might only be run when others are finished, and so on. In other words, the activities of a process form a network with arcs that point from a given activity to its successor activities. Since a process model has to reflect all possible valid executions of a specific business procedure, each activity within a process model must be connected to all of its potential follow-on activities. A process model may be perceived as a directed graph having nodes that are the activities of the process and having edges that connect an activity with its potential successors. Since an edge represents the potential control flow from one activity to another, it is also referred to as a *control connector.*

As an example, suppose the credit allocation process model contains the activities Solvency, Reject, Accept, BranchManagerApproval, and Notify. The potential follow-on activities of Solvency are Reject, Accept, and BranchManager-Approval. BranchManagerApproval has the potential successors Reject and Accept. The activity Notify is the successor of both Reject and Accept.

When a process model is executed (or *instantiated*) it depends on the concrete situation in which the process is run as to what subset of the set of all potential follow-on activities of a particular activity is really executed once this particular activity is terminated successfully. A "concrete situation" is captured by the collection of all values actually associated with the parameter types of the various containers of the model.

The dynamics of subsetting potential follow-on activities are added to our meta-model by allowing Boolean functions to be associated with each edge that connects two activities; these Boolean functions are called *flow conditions* or *transition conditions*. Each of the Boolean functions has an input container associated with it, its parameter types stemming from the output containers of the predecessor activities or from the input container of the process model itself. Potential follow-on activities of a successfully terminated activity are considered for execution in the concrete situation only when they are reachable from the terminated activity via an edge having a flow condition that returns "true" based on the actual parameter values in its container. By adding Boolean functions to the edges of the directed graph the perception of a process model within our meta-model is that of a weighted, directed graph.

Activities are in general long running, and it must be permissible to interrupt them. Thus, when an activity terminates, it has not necessarily performed its task successfully. But only successfully terminated activities are relevant when determining follow-on activities. To capture this situation, our meta-model permits assigning a Boolean function to each node in the graph which represents the *exit condition* or *end condition* of the activity. The Boolean function representing the exit condition of an activity again has an input container associated with it that has parameter types stemming from the containers of the activities or from the container of the process model itself. An activity terminates "successfully" if its associated exit condition returns "true" based on the actual parameter values in its container. By adding Boolean functions to the nodes of the weighted, directed graph, the perception of a process model within our meta-model is that of a weighted, colored, directed graph.

In general, activities within a process will be executed in parallel. (This occurs when processing allowances for expenses, for example. After the activity CheckBill, both the payroll department as well as the bookkeeping department can work on the bill for expenses in parallel.)

Parallel execution is enabled by a process model via an activity having outgoing edges weighted by Boolean functions that can be "true" concurrently. After having terminated successfully, such an activity will function as a "fork." The activities along the different branches can, in general, be worked on in parallel.

An activity having more than one incoming edge can function as a "join." For that purpose our meta-model associates a Boolean expression with each activity in the Boolean functions that weight the incoming edges of the respective activity; this expression is called a *synchronization expression*. If the synchronization expression is the conjunction of all the Boolean functions weighting the incoming edges, the corresponding activity can only be executed if the Boolean functions *have been evaluated* and the Boolean expression in the returned values is "true." An activity of the latter kind thus works as a "join."

If the Boolean function of an edge is evaluated as "false," the endpoint of that edge might never become executable. The Boolean functions of the edges leaving a node that can never be executed will never become evaluated functions. In the case where the endpoint of an edge having a Boolean function that will never be evaluated is a "join node," the corresponding process will never terminate.

In order to avoid this situation, the "forward transitive closure until joins" of the endpoint of an edge with a "false" Boolean function is computed. This means that all directed paths originating at the subject node are traversed until a join node is reached. All Boolean functions of edges in this closure are considered as "evaluated" with a return value of "false" (dead path elimination). In case all of the incoming edges of a join node reached during dead path elimination are already evaluated, its synchronization expression is computed, and if it results in "false," dead path elimination is performed for that join node. This is because the join node is never executed, thus leaving the process in doubt unless dead path elimination is not performed.

When allowing parallel execution of activities, we must be sure that an executing activity does not generate results that will be produced by another executing activity or that were required as input by another executing activity (the *Bernstein Criterion*[27]). Although there are different well-known techniques (e.g., "locking"[28]) to ensure the fulfillment of this criterion on a per parameter value base, we pursue a more restrictive approach: The instances of the parameter types of an activity's input and output container are treated as the local context of each particular activity.

As a consequence, shared instances have to be defined explicitly. It is done when providing the process model by connecting a particular parameter type of the output container of an activity with a particular parameter type of the input container of another activity via a directed edge (*data connector*). Data connectors are only allowed between containers having activities that can be reached along a directed path; this ensures that an activity does not expect data as input when the data could not be produced by a preceding activity.

It is also allowed to define a data connector pointing from the input container of the process to the input container of an activity as well as to define a data connector pointing from the output container of an activity to the output container of the process. Thus input can be passed to an instance of a process model once it is started and output can be passed from the process instance once it terminates.

The implementation

Currently, the FlowMark product is available to implement most of the meta-model described above; we will point out the few constructs that are not implemented in their generality. Built for the Operating System/2* (OS/2*) 2.1 environment, FlowMark is—with the exception of the animation part written in Prolog—completely written in C++ using object-oriented components and technologies (like its underlying object-oriented database system) that only recently have become available.

System structure. At the highest level, FlowMark essentially consists of two parts: build time and run time. The syntactical aspects of the meta-model are implemented in the build-time part, and its operational semantics are covered by the run-time part. Each of these parts is itself split into a client component and a server component.

The functionality provided by build time (Figure 1) comprises the blocks Animation, Process Definition, Staff Definition, Program Registration, and Data Structure Definition. The build-time part allows a process modeler to define and maintain all the information necessary for a FlowMark process to be executable. As the process is being defined, the animation facility lets the modeler examine the behavior of the model. After a process model is completed by its definer, it is translated into a startable process also called a process template.

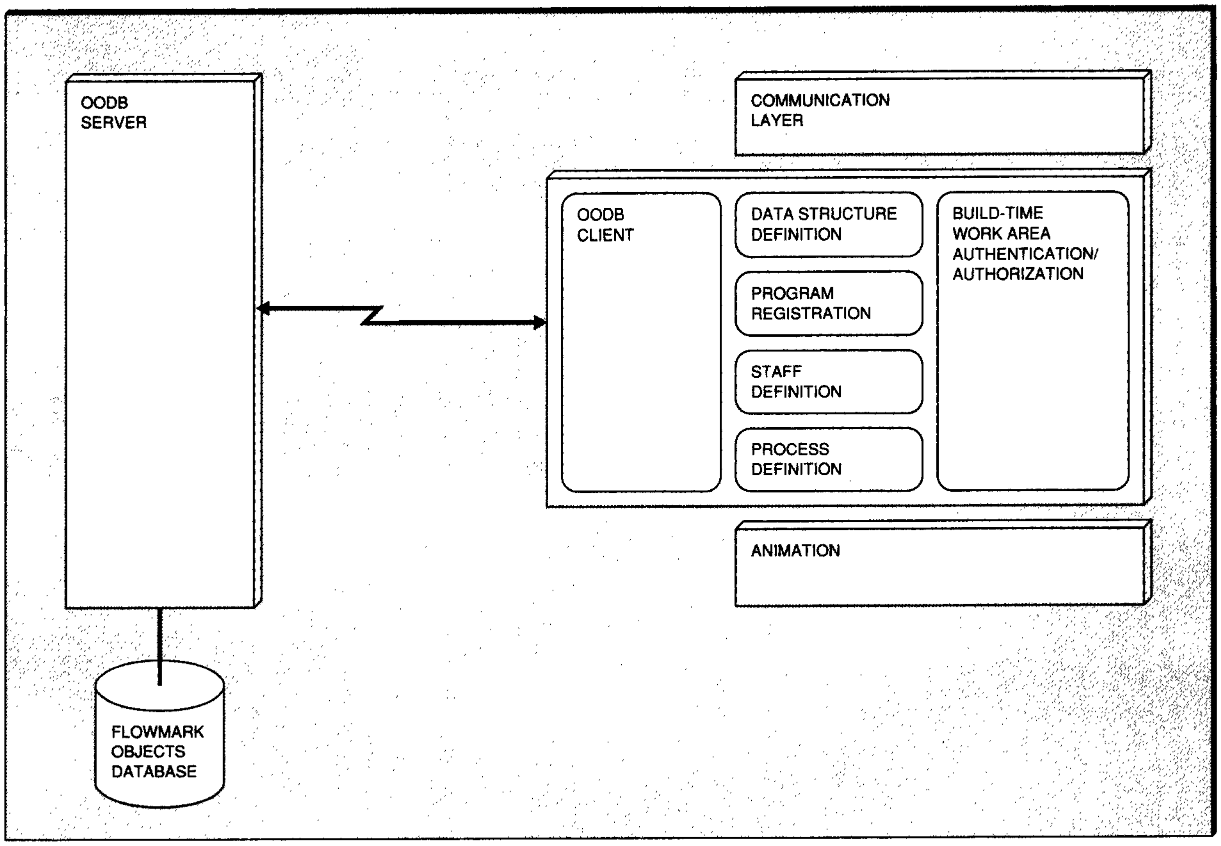
Build time exploits the client/server structure of the underlying database for seamless access to the definitional objects of FlowMark. The communication layer is used by the definition components to exchange notification messages for updating their views on data in case of modifications.

The run-time functions seen by the end user are Process Execution, Process List, Work List Handling, Local/Remote Program Execution, and Process API (application programming interface).

The client parts (build time and run time) provide the graphical end-user interface to all of the FlowMark functions. The server parts of FlowMark control and synchronize access to the FlowMark data and moreover synchronize all work maintained by FlowMark. Users can concurrently define processes (to be more precise: "describe process models") and store them in the database and at the same time execute an arbitrary number of ready-to-run processes. The number of processes and users is only constrained by the system resources available. The run-time server component of FlowMark is implemented as a hot pool. A hot pool in this context is a set of operating system processes each of which has the structure depicted in Figure 2. Each client request is dispatched to an idle server process; if all server processes are busy, the request waits to be served. The number of server processes can be configured by the user.

The work areas shown in Figure 2 are FlowMark-maintained OS/2 folders (for information on OS/2 see, e.g., References 29, 30, and 31) that organize all of the FlowMark information. The build-time work area contains a folder for each of its functional components. Thus a user finds all of the process models with which he or she is authorized to work in the process folder of his or her build-time work area. The run-time work area contains the work list folders of a user and a folder showing all startable processes, as well as the processes

Figure 1 FlowMark build-time system structure



the user has started or the running processes for which the user has administration rights. FlowMark requires the user to be identified at startup of either run time or build time for authorization purposes. Depending upon his or her authorization, a user can run either build time or run time (or both) concurrently.

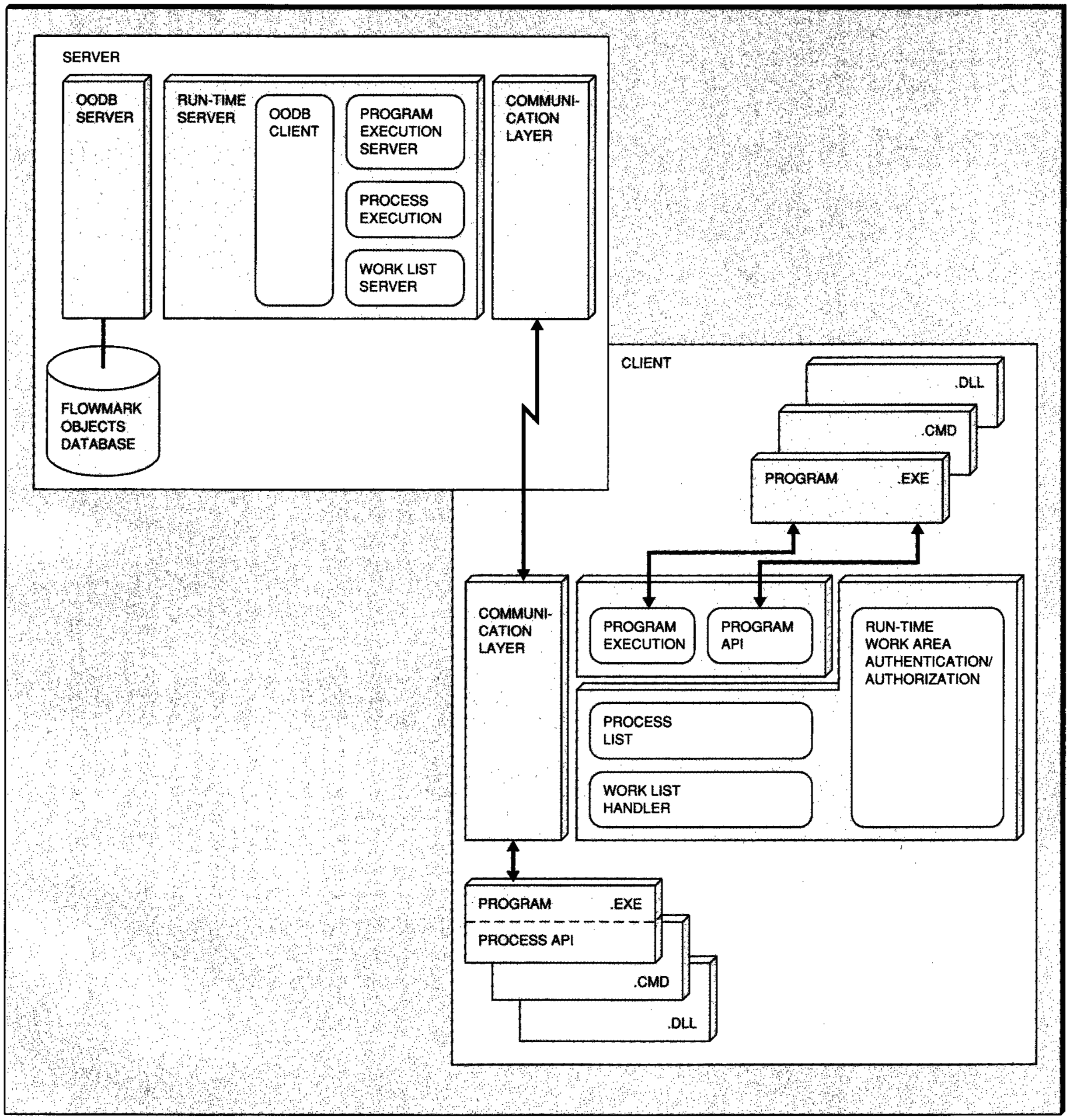
The run-time work area starts a separate process for local program execution at startup time. Local program execution takes care of program invocation as well as data passing between FlowMark and the invoked program. The process API is used by programs that want to start, stop, restart, suspend, and resume FlowMark processes. These calls are directly executed by the FlowMark run-time server.

The communication layer shields the FlowMark components from the specifics of the underlying transport protocols. It is implemented as a messaging service supporting synchronous and asynchronous message handling.

FlowMark has an import and export utility for maintenance and exchange of definitional information. It also alleviates the loading of existing enterprise information such as personnel data into the FlowMark database.

Process definition. Process definition is the most significant component as it lets a modeler graphically define a process model. According to the meta-model, a process model is presented as a weighted, colored, directed graph of activities. An activity is linked to other activities by control connectors or data connectors, i.e., both the control flow and the data flow are shown by edges of the directed graph.

Figure 2 FlowMark run-time system structure



The edges representing data connectors define the data connector map Δ of the meta-model as follows: A data connector edge is drawn between two activities A and B in the graphical representation of a process model if and only if $\Delta(A, B) \neq \emptyset$. Thus, a data connector edge pointing from A to B indicates that a map from the output container of A to the input container of B exists; this map (i.e., each member $d = (v_1, v_2) \in \Delta(A, B)$) is specified separately. Each container is characterized by one of the defined data structures. Since the data structures linked by a data connector may be distinct, the data connector also carries a mapping between those, if necessary. The mapping has to be done by the modeler who has to provide the necessary name and type matching between the corresponding data items to be passed (for identical data structures, FlowMark does the mapping automatically).

As previously described, a control connector pictures the sequence of the activities as well as the condition (transition condition) that has to be met for continuing navigation along the connector. The condition p weighting a control connector has to be specified as a Boolean expression evaluated at run time. A transition condition can simply be the constant Boolean value "true." This value is the default if the modeler does not specify it otherwise. For error or exception situations FlowMark offers the modeler a special control connector ("otherwise" connector) that is followed when no transition condition of the regular control flow connectors is fulfilled.

FlowMark supports three types of activities: program, block, and process. Each type has a different implementation. A *program activity* is implemented by a registered program. The program can either be a legacy application or a FlowMark specific application that makes use of the FlowMark programming interface to access the container data. The *block activity* is a construct that recursively allows nesting of activities to an arbitrary level. The block activity is only known within the process where it is created. It gives the modeler the opportunity to locally structure a process. It also provides a layering capability for refining the details of the implementation. Moreover, by using the editing functions of the process editor (clipboard functions), a block can easily be reused. A *process activity* is implemented by an already defined and existing process. The process that is referenced is dynamically started when the process activity is executed. The process activity ends when the referenced process finishes. Thus process models can be built and maintained in a very modular and incremental fashion. A program activity and process activity can either be manual or automatic. The execution of manual activities is initiated by a user, whereas automatic activities are immediately executed by FlowMark.

As previously described, multiple control connectors emanating from an activity (forking of control flow) easily expresses parallelism of activities within FlowMark. The semantics of multiple control connectors joining into one activity can be defined by the process modeler. At the moment, the two possible synchronization choices are: Execution can continue if at least one of the joining connectors evaluated to "true," or only if all of them have become "true." Thus, the synchronization expressions supported by FlowMark are

$$\phi_A = \left\{ \bigvee_{j=1}^{k} p_j, \bigwedge_{j=1}^{k} p_j \right\}$$

for each activity $A \in N$ with $\mathbb{P}_{\Rightarrow}(A) = \{p_1, \ldots, p_k\}$.

For each activity an end condition can be specified. The general meta-model allows end conditions to have an arbitrary input container; within FlowMark an end condition is a Boolean expression over the data items of the output container of its activity, i.e., the input container of the end condition p is always a subset of the output container of its activity A: $\iota(p) \subseteq o(A)$. Data connectors that target the output container of an end condition are thus not needed in this situation.

Within FlowMark, the set $N' \subseteq N$ of start activities always consists of *all* activities that have no incoming control connector, i.e., $N' = \{A \in N \mid \forall B \in N \, \forall p \in P\colon (B, A, p) \notin E\}$. Activities without an outgoing control connector are called *end activities*; i.e., the set of all end activities is given by $\{A \in N \mid \forall B \in N \, \forall p \in P\colon (A, B, p) \notin E\}$. A process ends regularly as soon as at least one end activity has been carried out successfully, no further control flow path can be taken, and the end condition of the process is met.

Resource definition. In order to allow the execution of activities at run time, FlowMark facilitates the definition of the following kinds of resource information: assignment data (i.e., data about organizations, roles, or persons), program registration data, and data structure information. This information can be bound to the activities of a process model.

Tasks. The syntax and semantics of the meta-model deal with the *logic* of a process model, i.e., they describe the potential flows of control and data *between activities* within each process instance. From an enterprise point of view the flow of activities itself *between agents* is of similar importance, i.e., the *logistics* of a process model have to be specifiable. Within an enterprise different activities of a running process are usually executed by different persons (the executing "agents" or staff members). In general, one and the same activity (instance) within different process instances will be executed by different persons; thus, it must be possible to couple activities not only to concrete persons but to abstract resources that will execute the bound activity.

FlowMark allows each activity to be coupled with such a resource. The resulting pair models a *task*. A task represents a concrete run-time work request to a particular person to perform a specific activity. The resources are the key within FlowMark to distributing the activities to the right people in the sequence the process prescribes.

Although the concept of a task is not apparently visible in FlowMark, activities have task-related information used at run time attached to them. Each activity has an assigned resource who is responsible for carrying out the work; this assignment either becomes specific by associating a particular person with the activity or becomes abstract by associating a role or an organization with the activity. Also, each activity may have additional information for its expected average duration. Optionally, every activity may specify whether someone should be notified in case the activity is not completed within the given amount of time. This event is called escalation. Duration and escalation can be defined for a process, too.

Organization, role, person. How staff assignments should be bound to activities is defined at modeling time along with the definition of the process itself. The essential entities for modeling the personal resource structure are organization, role, and evidently, person. For the relation of those entities FlowMark provides a simple but powerful model.

An organization is a grouping of people within a given enterprise and can be hierarchically structured, thus reflecting the organization chart of the entire enterprise. A role is seen as a functional position within an organization or an enterprise and may have certain skills as a prerequisite. Skills again can be qualified in terms of grading levels. More than one person can play the same role within one organization or within different organizations. Organizations as well as roles have

At run time FlowMark resolves the relation between organization or role and person.

one manager, respectively, and one coordinator. All of the above information along with data about each FlowMark user is stored in the FlowMark database.

At run time FlowMark resolves the relation between organization or role and person. If several persons qualify for performing an activity, all of them receive the work request at their workstations. As soon as one person in a group takes the work request, FlowMark withdraws the request from all other workstations of the group.

For assignment purposes the modeler also can refer to the anonymous person who eventually starts the process, to the manager of the person who starts the process, to the process administrator, or to the manager of the process administrator. Thus it is, for instance, possible to have the first activity performed by the user who starts a process.

Programs. Program activities are carried out by running the associated application or tool. The application is the interface by which the user performs work on the given request represented by the program activity. However, applications that, so to speak, implement a program activity are not restricted to an interactive program but can also be unattended programs started automatically by FlowMark. The information describing the application, together with its input and output data structure, is kept in the program registration database of FlowMark. The input and output data structure of the program are identical to the input and output container structure of its activity. Programs can be executed locally on the workstation of the end user (which is the usual case for interactive tools like word processors or spreadsheet programs and the like) or on any remote computer that hosts a FlowMark program execution process.

Data structures. FlowMark obviously has the need to maintain and interpret container data of activities since these data may be used within transition conditions and end conditions. In order to fulfill this need FlowMark incorporates its own data structure definition facility. The supported elementary data types are string, long, and float, and arbitrary but fixed-size arrays of the elementary types. Existing data structures can be aggregated to build new user-defined data structures (nesting of structures). The nested data structures are not referenced by the parent structure but become part of it.

Animation. The definition of a medium-to-large process model calls for an iterative approach combining definition with verification. FlowMark includes an animation part that lets the modeler run a process model in animation mode. Animation enables the debugging of process models, the analysis of the impact of changes to a process model, and so on.

It is possible to select an arbitrary set of activities (not necessarily startup activities) as a starting point for animation. The modeler can step forward and backward through the process model watching its presumed behavior in terms of work assignments and navigation between activities. All work lists of concurrent assignees can be viewed simultaneously.

Animation does not require a process model to be completely defined; also program activities need not be coded for animation purposes. As animation navigates through the activities, the modeler is prompted for missing pieces of information. To alleviate the task of manually feeding data to the animation facility, all of the input data can be saved for reuse later.

Sample scenario. Figure 3 shows the FlowMark diagram of a credit allocation process modeled with FlowMark. Control flow connectors are shown as solid lines, and data connectors are shown as dashed lines. The two different connector types can be selectively hidden or shown. The same option is applicable to transition and end conditions. In the diagram only transition conditions are shown.

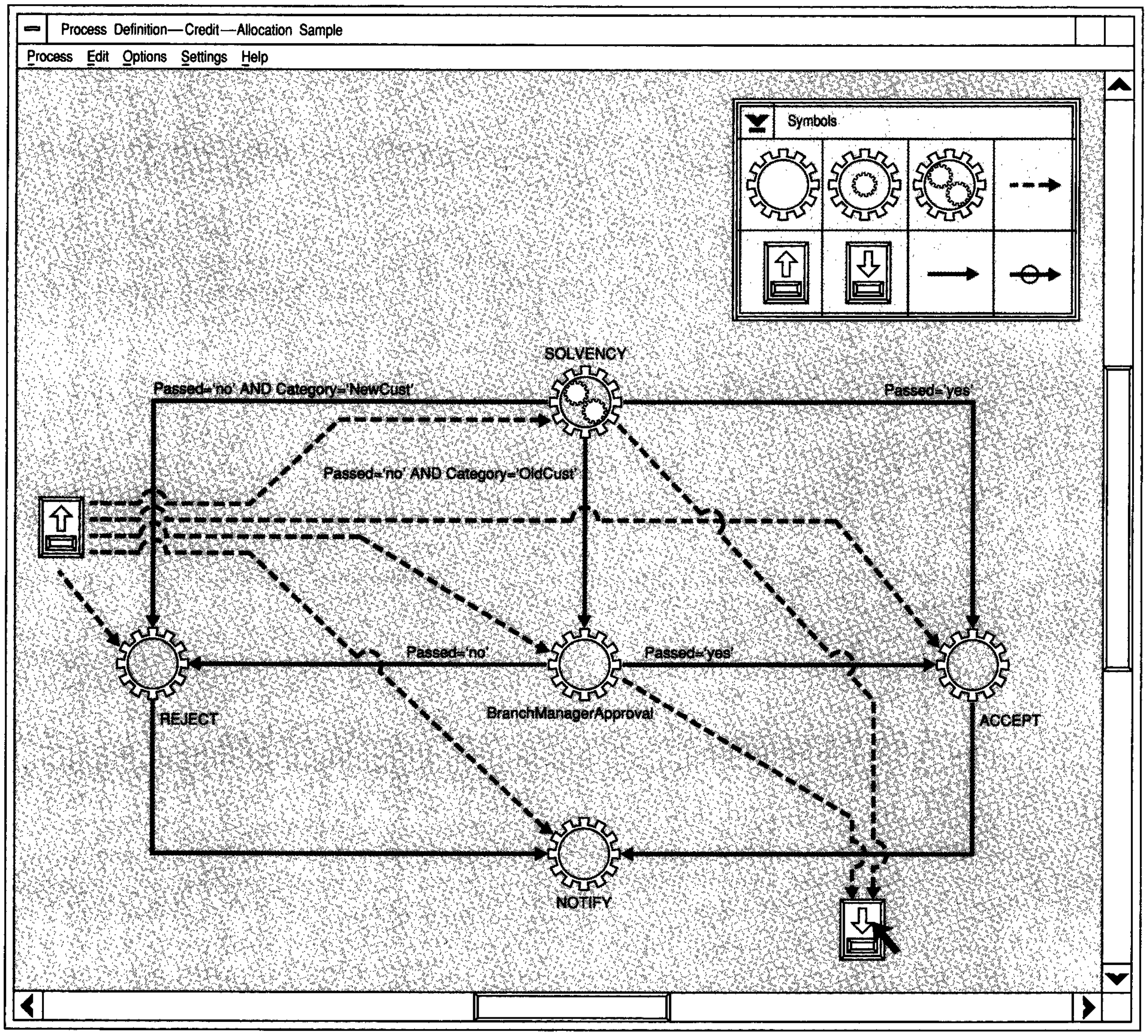
The scenario in the figure assumes that a loan clerk starts the process from the FlowMark runtime work area and initially is prompted for some customer data. These customer data are put into the process input container before the first activity can be started. For sake of simplicity the assumption in our process model is that the customer data just consist of a customer number (CustNo). The customer number is used as input in each activity. Therefore, data connectors are leading from the symbol representing the process input container to all other activities.

The first step of the process is to check the solvency of the customer by use of the process activity Solvency. Solvency receives the customer number as input and returns two information items in its output container. One item indicates whether the solvency check was passed or not and the other whether the customer is already known to the bank or not. The results of Solvency are stored in the process output container that is depicted by the corresponding data connector leading to the process output container symbol. Depending on the results of Solvency, three follow-up activities are possible: Accept, BranchManagerApproval, and Reject. BranchManagerApproval lets a branch manager override the Solvency results for known and trustworthy customers. Accept and Reject are automatic activities that handle the administrative aspects and database updates. Notify is the final step for producing printouts and notification letters.

Execution. As mentioned in the beginning of this section, the server part of FlowMark is the coordinator and synchronizer for work requests. The core part of the server process consists of the process execution component and the work list management component.

A process model becomes a startable process by translating it into the executable format. During translation the process model is checked for consistency and completeness; e.g., compliance with the syntax of the meta-model is verified. The startable process is an entity separate from the model and therefore is also unaffected by any changes to the model afterwards.

Figure 3 Credit allocation process


Process Definition—Credit—Allocation Sample
Process Edit Options Settings Help
Symbols
SOLVENCY
Passed='no' AND Category='NewCust'
Passed='yes'
Passed='no' AND Category='OldCust'
Passed='no'
Passed='yes'
REJECT
BranchManagerApproval
ACCEPT
NOTIFY

Process instances can either be started via the graphical end-user interface by clicking on the appropriate startable process icon or via the callable program interface (API). The algorithm for process execution in essence is encapsulated in each process instance itself and in its activity instances. When a new process instance is created, it is copied from the startable process blueprint.

The first step in running the process consists of finding its start activities N', distributing them according to their bound personal resources and allowing them to be executed (i.e., $\Sigma_0 = N'$). The activation condition of start activities (i.e., the value returned by their synchronization expressions) is "true" by definition. For all nonstart activities the activation condition is checked when all incoming control connectors either have been evaluated or have been marked by a recursive procedure called "dead path elimination."

As described above, the meta-model exploits three-valued logic when evaluating synchronization expressions, i.e., to check activation conditions. FlowMark requires that all transition conditions of incoming connectors be evaluated before the activation condition is checked, thus circumventing three-valued logic.

Dead paths within a process are subgraphs having activities that can no longer become startable because a previous transition condition within the subgraph evaluated to "false." Activities relying on input data from "dead activities" can be specified if the user is prompted for the missing input data, i.e., $\Lambda(i, A, v)$ is determined, or an exception is raised that leads to the termination of the process.

The steps generally involved in executing an activity are:

- Resolving the staff assignment
- Putting work requests on the work lists of the assigned persons
- Executing activity implementations
- Interacting with the work list handler client to handle manual (interactive) program invocations
- Checking the end condition

Resolving staff assignments and keeping track of the work requests are done by the work list manager. If not specified otherwise, role resolution is done dynamically within the context of the organization of the person who started the process. Such resolution allows for different physical assignments at run time based on the same model information. The assigned work requests stemming from the startable activities are stored in the FlowMark database. The state of the work flow is thus persistent, allowing for forward recovery in case of system failures (fault tolerance with respect to soft crashes). If there is manual activity and the assigned user is logged on to FlowMark and connected to the server, the work list manager immediately forwards the new request to the workstation of the user.

For automatic program activities, the program execution server takes care of invoking the tool without any further user interaction. Block activities and process activities are executed by starting their respective implementations. It is the underlying activity graph for the block, and it is the newly created instance of the referenced startable process for the process activity. In both cases the execution is synchronous in the sense that those activities complete after the underlying block or the referenced process has finished successfully.

Manual activities are performed when an end user issues the start request via the work list. On receipt of such a request the work list server passes it through to the program execution server that handles it.

Resolving staff assignments and keeping track of the work requests are done by the work list manager.

The end condition of an activity is checked when its implementation is finished and the output data are available in the output container. If the evaluation of the end condition $\varepsilon(A)$ of the activity A returns "true" (i.e., $\varepsilon(A)({}^{i}\iota(\varepsilon(A))) = 1$), the outgoing control flow connectors are examined. Connectors whose condition evaluates to "true" are pursued for further execution of their target activities. Finally, $\Sigma_i(A)$ is computed, resulting in the corresponding member Σ_i of the execution family. When no further control paths can be found, the process is finished. The completion status is reflected in the process folders of users who started the process and in the folder of the corresponding process administrator.

Process list. The process list gives a user access to all of the process templates that he or she is authorized to instantiate as well as to all process instances allowed to be seen. A process instance is seen by a user either when he or she instantiates it or when the user is the administrative owner of the process. The user interface allows filtering and sorting of process instances according to their status or to the actual data values within a process input container.

Work list handler. The end-user interface for a person working with FlowMark-generated work requests is called the work list handler. A user can maintain multiple work lists in parallel in order to arrange the work requests according to his or her liking. For this purpose FlowMark provides sorting and filtering capabilities for work requests. The work list handler gives the user all of the necessary functions for maintaining work lists, transferring work items to fellow workers, and starting processes. For each work item the user also can obtain a graphical view of the current status of the process from its origination.

All the information visualized by the work list handler is retrieved from the work list server when the user logs on to FlowMark. Thus a user can log on from any workstation and always obtain his or her work lists reflecting the actual situation; this especially results in fault tolerance with respect to hardware failures.

In order to work on a request appearing as an icon on a work list, the user simply double clicks on the icon. The underlying program is invoked, and the user interacts with the program until the task is finished. FlowMark detects when the program terminates and consequently passes control to the server for exit checking and further navigation of the process.

Depending on the level of authorization given, a user may have access to the work lists of other people and transfer work items between them. Such access can be used to channel all work requests to a supervisor who then distributes the work requests to the proper people.

Program execution. The program execution server finds the appropriate target host for executing a program by retrieving the corresponding program registration data. The target host for execution may be the workstation where an end user manually started an activity via his or her work list (local program invocation) or may be any other workstation that has a running FlowMark program execution process (remote program invocation). FlowMark supports invocation of several program types:

- OS/2 command files, executables, and dynamic link libraries
- Customer Information Control System (CICS*), Information Management System (IMS), or Time Sharing Option (TSO) applications via ASF (IBM Application Support Facility)

ASF uses an OS/2 stub program for communication with the CICS, IMS, or TSO environment. This makes the invocation of programs in these environments transparent for FlowMark.

As mentioned earlier, program execution runs in a separate OS/2 process. For each command file or executable a new OS/2 process is started. Resynchronization with FlowMark (also rendering the system return codes) occurs via an OS/2 queuing mechanism. For each dynamic link library a new thread is created within program execution for calling the appropriate entry point.

We distinguish between programs that are FlowMark aware and those that are not. FlowMark-aware programs exploit the FlowMark API that allows them to obtain data from the activity input container and put data into the activity output container. The data structure of the input and output container can also be queried by means of an API call, returning names and types of all the data elements down to the elementary fields.

So-called legacy programs that are not aware of FlowMark can obtain data from FlowMark via the command line. The desired data items of the input container used as command line parameters have to be specified as substitution variables at program registration time. FlowMark takes care of the proper substitution with the corresponding input container data at run time.

FlowMark currently offers C language bindings for its API and corresponding REXX language wrappings. REXX procedures often give a modeler a very versatile and quick way of implementing program activities.

Audit trail. One of the important functions of executing business processes is tracking the process in an auditable way. Tracking is done in FlowMark by means of an audit trail that records all events from starting a process, through working on activities or transferring activities to other coworkers, up to the successful completion or termination of a process. Various reports can be produced with the information provided in the audit trail. Process and activity characteristics such as overall process duration or maximum activity duration can be extracted from such reports. Analyzing the audit data should lead to identification of deficiencies in the process, and when they are consequently addressed, it leads to improved business processes.

Mathematical formulation of the meta-model

In this section we provide a mathematical formulation of the meta-model. Please note that a person modeling business processes based on FlowMark has no need to understand the mathematics presented here. But the mathematical foundation of the meta-model demonstrates the robustness of the concepts underlying our approach to the management of business processes.

Process models as weighted, colored, directed graphs. We introduce a special class of weighted, colored, directed graphs called *PM-graphs* (Process Model graphs). A PM-graph is a mathematical abstraction of a process model, i.e., it provides the syntactical elements of our meta-model.

Activities as maps. Let N denote the set of all activities that a particular process model consists of; it is important to note that we explicitly assume that each separate occurrence of an activity within a process model is uniquely identified. The set of all parameter types occurring somewhere in a process model is denoted as V; mainly, V consists of all input parameter types and output parameter types of each single activity (V also encompasses the input parameter types of all Boolean functions of a process model and the special parameter types like the maximum time for an activity to be elapsed before it has to be started, etc.). Accordingly, there is a map $\iota: N \to \wp(V)$, associating with each activity $A \in N$ the set of its input parameter types $\iota(A) \subseteq V$, and there is a map $o: N \to \wp(V)$ associating with each activity $A \in N$ the set of its output parameter types $o(A) \subseteq V$ ($\wp(M)$ denotes the power set of the set M). $\iota(A)$ and $o(A)$ represent the input container and the output container of A, respectively. Each parameter $v \in V$ has associated with it a set DOM(v) as its domain, i.e., DOM(v) is the set of values the parameter type v may take. Thus, an activity A can be perceived as a map

$$A: \underset{v \in \iota(A)}{\times} \mathrm{DOM}(v) \to \underset{v \in o(A)}{\times} \mathrm{DOM}(v)$$

Let A be the activity Increase_Salary. The input container of this activity consists of the parameter types Employee#, Salary, and Level, i.e., $\iota(A) =$ {Employee#, Salary, Level}. Since A produces a new salary, its output container consists of the parameter type Salary, i.e., $o(A) =$ {Salary}. Valid employee numbers are character strings of length 10, floating point numbers provide the values for salaries, and levels are measured via integer numbers, i.e., DOM(Employee#) = CHAR(10), DOM(Salary) = FLOAT, and DOM (Level) = INTEGER. Thus, A is perceived as the map

$$\text{Increase\_Salary}: \mathrm{CHAR}(10) \times \mathrm{FLOAT} \times \mathrm{INTEGER} \to \mathrm{FLOAT}$$

Control connectors. Once an activity $A \in N$ terminated successfully, certain follow-on activities are possible or required. All potentially occurring follow-on activities $A_1, \ldots, A_n \in N$ of A are connected with A via directed edges. Each edge is directed toward the successors of a given activity. The edges $(A, A_1), \ldots, (A, A_n)$ are thus representing the fact, that $A_1, \ldots, A_n$ are *the* potential follow-on activities of A (and the *only* potential follow-on activities of A). In this way, a set of edges $E \subseteq N \times N$ is generated. The directed graph $G = (N, E)$, which represents at the current stage a process model in our meta-model, reflects the *potential* work flow of a process model.

Within the meta-model, the parameters V are representing the context of the potential instances of the process model. Because activities are maps defined in those parameters, thus changing that context, not all potential follow-on activities of a certain activity will be meaningful in each context. The set of actual follow-on activities is context dependent. The meta-model provides predicates in order to allow for the modeling of this context dependency:

If P denotes the set of all *predicates* belonging to a process model, there is a map $\iota: P \to \wp(V)$ associating with each predicate $p \in P$ its set of input parameter (types) $\iota(p) \subseteq V$, representing the input container of p. A predicate p is then a Boolean function

$$p: \underset{v \in \iota(p)}{\times} \mathrm{DOM}(v) \to \{0, 1\},$$

where "0" denotes "false" and "1" denotes "true"; always, the two constant Boolean functions "0" and "1" are valid predicates ($q_1 \equiv 0, q_2 \equiv 1 \in P$).

In order to model which follow-on activities are possible or required in the *actual* context (actual values of all parameters in V), the edges connecting activities are weighted by predicates. The set of all edges is thus $E \subseteq N \times N \times P$. Let

$$\pi_{i_1 \ldots i_k}: M_1 \times \ldots \times M_n \rightarrow M_{i_1} \times \ldots \times M_{i_k}$$

denote the projection map between Cartesian products $(i_1, \ldots, i_k \in \{1, \ldots, n\})$. Then $p \in \pi_3(E) \subseteq P$ is called a *flow condition* or *transition condition*. The members $e \in E$ are called *control connectors*. $\mathbb{P}_{\Leftarrow}(A) := \pi_3(\{e \in E \mid \pi_1(e) = A\})$ is the set of all flow conditions of control connectors *leaving* A, and $\mathbb{P}_{\Rightarrow}(A) := \pi_3(\{e \in E \mid \pi_2(e) = A\})$ is the set of all flow conditions of control connectors *entering* A.

Data connectors. Data connectors specify how input containers of activities or predicates are composed of output parameters of other activities of the corresponding process model. The set of data connectors pointing to the input container of the activity (or predicate) B from the output container of the activity A is given in our meta-model as a set $\Delta(A, B)$. An element $d \in \Delta(A, B)$ is a pair consisting of an element of $o(A)$ and an element of $\iota(B)$. $d = (v_1, v_2) \in \Delta(A, B)$ specifies that at run time when B is invoked the actual value of v_2 of the input container of B is the current value of v_1 from the output container of A. Thus, there is a map

$$\Delta: N \times (N \cup P) \rightarrow \bigcup_{A \in N, B \in N \cup P} p(o(A) \times \iota(B))$$

called *data connector map* having the following properties:

1. $\Delta(A_1, A_2) \in p(o(A_1) \times \iota(A_2))$
2. $(x, z), (y, z) \in \cup_{A \in N} \Delta(A, B) \Rightarrow x = y$
3. $\Delta(A_1, A_2) \neq \emptyset \Rightarrow A_2$ reachable from A_1

We call A_2 *reachable* from A_1 $(A_1 \neq A_2 \in N)$ if and only if there is a directed path from A_1 to A_2, i.e.,

$$\exists B_1, \ldots, B_k \in N: (A_1, B_1), (B_1, B_2), \ldots, (B_k, A_2) \in \pi_{12}(E)$$

Condition 3 above ensures that the data required by a particular activity are really produced by an activity that ran before and that terminated successfully; in case dead path elimination occurred and values from output containers of traversed activities are required in the input containers of follow-on activities, a special function is used to determine these values. The reachability condition further ensures that output produced by a particular activity cannot be expected as input by an activity that is running in parallel. Also, results that are produced by two activities (within their local context) running in parallel cannot be mapped to the same input value of a third activity. This is ensured by condition 2 above. As a consequence, the Bernstein Criterion[27] is fulfilled ensuring correct parallel executions.

Moreover, our meta-model allows input data to be passed from the input container of the process to the input containers of its encompassed activities and predicates, and allows output data of the activities to be passed to the output container of the process. Again, this is achieved via data connectors provided by the following *process data connector map*:

$$\hat{\Delta}: N \rightarrow \bigcup_{A \in N} (p(\iota(G) \times \iota(A)) \cup p(o(A) \times o(G)))$$

where $\iota(G)$, $o(G) \subseteq V$ denotes the input container and the output container of the process model G, respectively. The obvious condition

$$\forall A \in N: \hat{\Delta}(A) \in p(\iota(G) \times \iota(A)) \cup p(o(A) \times o(G))$$

has to be fulfilled. The input and output container of the process model G is encompassed in the Bernstein Criterion via the following two conditions:

1. $\forall B \in N: (x, z), (y, z) \in p(\iota(G) \times \iota(B)) \cup \cup_{A \in N} \Delta(A, B) \Rightarrow x = y$
2. $(x, z), (y, z) \in \cup_{B \in N} p(o(B) \times o(G)) \Rightarrow x = y$

Coloring activities. An *exit condition* or *end condition* is a Boolean function associated with a node $A \in N$ used to check whether A finished its work successfully or not. Within our meta-model, exit conditions are associated with activities via a map $\varepsilon: N \rightarrow P$. When an activity A terminates, its exit condition $\varepsilon(A)$ is evaluated based on the actual values of $\iota(\varepsilon(A))$. The termination of A is considered to be successful if and only if $\varepsilon(A)(\iota(\varepsilon(A)))$ = "true." As long as termination of A is not successful, it has to be worked on at a later time, and no navigation is performed starting at A.

In order to allow for user-defined synchronization of parallel work within processes, the meta-model allows us to associate a *synchronization expression* with each activity $A \in N$. This is a Boolean expression formed with the flow conditions of all control connectors entering A, i.e., a Boolean expression in $\mathbb{P}_{\Rightarrow}(A) = \pi_3(\{e \in E \mid \pi_2(e) = A\})$. With

$$\phi_A := \left\{ \bigvee_{j=1}^{k} \bigwedge_{i=1}^{l_j} p_i' \mid p_i' \in \{p, \neg p \mid p \in \mathbb{P}_{\Rightarrow}(A)\} \right\}$$

our meta-model associates a synchronization expression with a node via the map

$$\Phi : N \rightarrow \bigcup_{A \in N} \phi_A, \qquad \Phi(B) \in \phi_B$$

In order to be activated the synchronization expression $\Phi(B)$ of an activity B must be "true." Because of this, synchronization expressions are also referred to as *activation conditions*. It is important to note that a conjunction $\wedge p_i'$ will return "unknown" until all affected flow conditions p_i have been evaluated. A flow condition p_i is evaluated for $(A, B, p_i) \in E$ once A terminated successfully, or the act of dead path elimination traversed A. If all flow conditions of at least one conjunction $\wedge p_i'$ are evaluated and returned "true," the whole synchronization expression

$$\Phi(B) = \bigvee_{j=1}^{k} \bigwedge_{i=1}^{l_j} p_i'$$

will return "true." If none of the conjunctions of a synchronization expression returned "true" and at least one of its conjunctions returned "unknown," the synchronization condition returns "unknown."

Process models: The formal definition. We are now ready to provide the formal definition of a "process model"; when the abstract properties of a process model are of more interest we will talk about PM-graphs:

Definition: A tuple $G = (N, E, P, V, \Phi, \varepsilon, \Delta, \hat{\Delta}, N')$ is called a *process model* (or a *PM-graph*): $\Leftrightarrow$

1. N is a finite set of *activities.*
2. V is a finite set of *parameters.*
3. P is a finite set of *predicates.*
4. There is a map

   $$\iota : N \cup P \cup \{G\} \rightarrow \wp(V),$$

   and a map

   $$o : N \cup \{G\} \rightarrow \wp(V),$$

   where ι associates with each element of $N \cup P \cup \{G\}$ *input parameter (types)* and o associates with each element of $N \cup \{G\}$ *output parameter (types)*; $\iota(B)$ and $o(B)$ are also called the *input container* and *output container* of B, respectively.
5. Each $v \in V$ has associated with it a set DOM(v).
6. Each activity $A \in N$ is a map

   $$A : \underset{v \in \iota(A)}{\times} \mathrm{DOM}(v) \rightarrow \underset{v \in o(A)}{\times} \mathrm{DOM}(v)$$

7. Each predicate $p \in P$ is a map

   $$p : \underset{v \in \iota(p)}{\times} \mathrm{DOM}(v) \rightarrow \{0, 1\}$$

8. The set $E \subseteq N \times N \times P$ is *unified*, i.e.,

   $$\forall e, e' \in E : \pi_1(e) = \pi_1(e') \wedge$$
   $$\pi_2(e) = \pi_2(e') \Rightarrow e = e';$$

   a member of $\pi_3(E)$ is called a *flow condition.*
9. $N' \subseteq N, N' \neq \emptyset$, is the set of *start activities.*
10. $\varepsilon : N \rightarrow P$ associates with each $A \in N$ an *exit condition.*
11. The map Φ is called a *synchronization map*,

    $$\Phi : N \rightarrow \bigcup_{A \in N} \phi_A, \qquad \Phi(B) \in \phi_B$$

    where each member of

    $$\phi_A := \left\{ \bigvee_{j=1}^{k} \bigwedge_{i=1}^{l_j} p_i' \mid p_i' \in \{p, \neg p \mid p \in \mathbb{P}_{\Rightarrow}(A)\} \right\}$$

    is called a *synchronization expression.*

12. The map

$$\Delta : N \times (N \cup P) \rightarrow \bigcup_{A \in N, B \in N \cup P} p(o(A) \times \iota(B))$$

is called a *data connector map* having the following properties:

- $\forall A_1, A_2 \in N : \Delta(A_1, A_2) \in p(o(A_1) \times \iota(A_2))$
- $\forall B \in N : (x, z), (y, z) \in \cup_{A \in N} \Delta(A, B) \Rightarrow x = y$
- $\forall A_1, A_2 \in N : \Delta(A_1, A_2) \neq \emptyset \Rightarrow A_2$ reachable from A_1

13. The map

$$\hat{\Delta} : N \rightarrow \bigcup_{A \in N} (p(\iota(G) \times \iota(A)) \cup p(o(A) \times o(G)))$$

is called a *process data connector map* having the following properties:

- $\forall A \in N : \hat{\Delta}(A) \in p(\iota(G) \times \iota(A)) \cup p(o(A) \times o(G))$
- $\forall B \in N : (x, z), (y, z) \in p(\iota(G) \times \iota(B)) \cup \cup_{A \in N} \Delta(A, B) \Rightarrow x = y$
- $(x, z), (y, z) \in \cup_{B \in N} p(o(B) \times o(G)) \Rightarrow x = y$

A closer look at a PM-graph $G = (N, E, P, V, \Phi, \varepsilon, \Delta, \hat{\Delta}, N')$ reveals that it describes two interrelated graphs. The first graph represents the *control flow* of the process model G and is described by the tuple $(N, E, P, V, \Phi, \varepsilon, N')$; the second graph describes the *data flow* of the process model G and is described by the tuple $(N, V, \Delta, \hat{\Delta})$. The control flow is a weighted, colored, directed graph $G' = (N, E)$, with maps as nodes, flow conditions as weights of edges, and pairs of synchronization expressions and exit conditions as colors of the nodes. The data flow is a weighted, directed graph $(N, \mathcal{E})$ with edges $\mathcal{E} \subseteq \mathrm{N} \times \mathrm{N} \times p(\mathrm{V} \times \mathrm{V})$ and $(A, B, \Delta(A, B)) \in \mathcal{E} \Leftrightarrow \Delta(\mathrm{A}, \mathrm{B}) \neq \emptyset$, the weights of which are determining the mapping of values from output containers to input containers.

Interpreting activity networks. Instantiating a process model, i.e., the execution of a process, mainly consists of navigating through the process model and the execution of activities. Activities are allowed to be executed only if they are selected beforehand. We describe now the operational semantics of the syntactical elements of our meta-model as provided in the above definition. Thus, we provide the rules on how to interpret PM-graphs.

Activities and their states. The interpretation of the syntactical elements of a process model in context with the actual parameters of the various containers results in a running process, i.e., in a dynamical process instance. Interpretation happens at particular discrete points in time, for example, once an activity terminates successfully. Thus, the aspect of time can be covered by the set of natural numbers $\mathbb{N}$. $0 \in \mathbb{N}$ represents the point in time in which a new instance of a process model G is started. Each activity of a process has associated with it at each point in time $i \in \mathbb{N}$ exactly one state $s \in S$; the state set S includes the states "executable," "activated," and "successful" (and "evaluated" and "not-evaluated," which are relevant for predicates only).

The map $\omega : \mathbb{N} \times N \rightarrow S$, which associates at any time $i \in \mathbb{N}$ with each $A \in N$ the actual state $\omega(i, A)$, is called *state map*; via $\omega_i(A) := \omega(i, A)$ a map $\omega_i : N \rightarrow S$ is induced for each $i \in N$. The map $\alpha : \mathbb{N} \rightarrow p(N), i \mapsto \omega_i^{-1}$ ({activated}) is called ($\omega-$)*active map*; $\alpha_i := \alpha(i)$ is the set of all currently active activities at time i (remember that we have shown before that the activities in α_i are satisfying the Bernstein Criterion).

$\lambda : \mathbb{N} \rightarrow p(N), i \mapsto \omega_i^{-1}$ ({successful}) is called ($\omega-$)*successful map*; $\lambda_i := \lambda(i)$ is the set of successfully terminated activities at time i (for a more precise definition of this set see below).

Dead path elimination. Dead path elimination occurs when it is detected that a particular activity can never reach the state "executable" in the current instance of the process model. The map $\Xi : \mathbb{N} \times E \rightarrow \{0, 1\}$ specifies at each time i whether an edge $e \in E$ was traversed by the dead path elimination procedure ($\Xi(i, e) = 1$) or not ($\Xi(i, e) = 0$). The map Ξ induces a map $\Xi_N : \mathbb{N} \times N \rightarrow \{0, 1\}$ via $\Xi_N(i, A) = 1 \Leftrightarrow \exists B \in N \exists p \in P : \Xi(i, (A, B, p)) = 1$. Note that the dead path elimination procedure ensures that Ξ_N is well-defined.

Actual values in containers. Let $B \in N \cup P \cup \{G\}$. Then ${}^i\iota(B)$ and ${}^io(B)$ (the latter exists only for $B \notin P$) denote the input container and the output container, respectively, of B at the time $i \in \mathbb{N}$, where all formal parameters are bound to their actual values. If $v \in \iota(B) \cup o(B)$ denotes a formal parameter, its actual value at time $i \in \mathbb{N}$ is denoted by ${}^{i}v$. The binding of formal parameters to actual values respects both the data flow Δ between activities and the data flow $\hat{\Delta}$ between activities and the process itself:

1. When an instance of the process model G is started, all input containers are considered to be initialized with default values; the resulting input container is denoted by ${}^{0}\iota(B)$:

   $$\forall B \in N \cup P \cup \{G\} \forall v \in \iota(B): {}^{0}v \in \mathrm{DOM}(v).$$

2. The values for the parameters in the output container of a successfully terminated activity are the return values of this activity; the output containers of all other activities are considered to consist of the default values of their specified formal parameters. The resulting output container is denoted by ${}^{i}o(B)$:

   Let $i \neq 0$ and $\iota(A) = \{v_1, \ldots, v_k\}$.
   - $\forall A \in \lambda_i: {}^{i}o(A) = A({}^{i-1}v_1, \ldots, {}^{i-1}v_k)$
   - $\forall A \notin \lambda_i: {}^{i}o(A) = A(v_1^0, \ldots, {}^{0}v_k)$

3. For times $i \neq 0$ the actual values of the input containers are determined by the data connectors; if there is no data connector specified to determine the actual value of a particular formal parameter, its default value is taken. In cases where the data connector originates from an activity traversed by the dead path elimination, a special function determines the actual value of the affected formal parameters. The resulting input container is denoted by ${}^{i}\iota(B)$:

   - $\forall B \in N \cup P \cup \{G\} \forall v \in \iota(B): (w, v) \in \Delta(A, B) \cup \hat{\Delta}(B) \wedge \Xi_N(i-1, A) = 0 \Rightarrow {}^{i}v = {}^{i-1}w$

   - $\forall B \in N \cup P \cup \{G\} \forall v \in \iota(B): (w, v) \in \Delta(A, B) \cup \hat{\Delta}(B) \wedge \Xi_N(i-1, A) = 1 \Rightarrow {}^{i}v = \Lambda(i, B, v)$

     where $\Lambda: \mathbb{N} \times N \times V \rightarrow \cup_{w \in V} \mathrm{DOM}(w)$ is a partial map defined for $(A, v) \in N \times V$ with $v \in \iota(A)$ satisfying $\Lambda(i, A, v) \in \mathrm{DOM}(v)$; Λ is called a *dead parameter map.*

   - $\forall B \in N \cup P \cup \{G\} \forall v \in \iota(B): \{(w, v) \mid A \in N \wedge (w, v) \in \Delta(A, B) \cup \hat{\Delta}(B)\} = \emptyset \Rightarrow {}^{i}v = {}^{0}v$

We are now able to give a precise definition of the ($\omega-$)successful map: It is $A \in \lambda_i : \Longleftrightarrow$

1. $\exists j < i : \omega(j, A) = \text{activated}$
2. $\varepsilon(A)({}^{j+1}\iota(\varepsilon(A))) = 1$
3. $\omega(j+1, A) = \cdots = \omega(i, A) = \text{successful}$

Predicates and their states. At any point in time predicates have a state associated with them too. The states of the predicates P are determined by the states of the activities N; thus the state map ω induces the *predicate state map* $\xi: \mathbb{N} \times P \rightarrow S$ as follows:

1. At time $i = 0$ the state of all predicates is "not-evaluated":

   $$\forall p \in P: \xi(0, p) = \text{not-evaluated}$$

2. At times $i > 0$ the flow conditions of all successfully terminated activities are in the state "evaluated":

   $$\forall A \in \lambda_i \forall p \in \mathbb{P}_{\Leftarrow}(A): i \geq 1 \Rightarrow \xi(i, p) = \text{evaluated}$$

3. At times $i > 0$ the flow conditions of all edges traversed via dead path elimination are in the state "evaluated" (and returned "false"!):

   $$\forall (A, B, p) \in E: \Xi(i, (A, B, p)) = 1 \Rightarrow \xi(i, p) = \text{evaluated}$$

4. Once a predicate has been evaluated it has the state "evaluated" from that time on:

   $$\forall p \in P: \xi(i, p) = \text{evaluated} \wedge j > i \Rightarrow \xi(j, p) = \text{evaluated}$$

5. In all other situations, $p \in P$ has the state "not-evaluated."

The return value of a synchronization expression

$$\Phi(B) = \bigvee_{j=1}^{k} \bigwedge_{i=1}^{l_j} p_i^j$$

is determined based on the setting

$$\forall 1 \leq j \leq k \forall 1 \leq i \leq l_j: \xi(t, p_i) = \text{not-evaluated} \Rightarrow p_i^j({}^{t}\iota(p_i^j)) = \text{unknown}$$

by the usual rules of three-valued logic (e.g., $\neg$ unknown = unknown, unknown $\wedge$ true = unknown, unknown $\wedge$ false = false, unknown $\vee$ false = unknown, unknown $\vee$ true = true). Thus, if all conjunctions $\wedge p'_i$ encompass at least a flow condition p with $\xi(t, p)$ = not-evaluated, then $\Phi(B)$ returns at time t the value "unknown." If there is at least one conjunction $\wedge p'_i$ in which all flow conditions have the state "evaluated" and for all of these predicates p'_i it is $p'_i({}^t\iota(p'_i)) = 1$, then $\Phi(B)$ returns at time t the value "1."

Actual successors. We first define the set of all follow-on activities of a given activity A which are reachable from A at a certain point in time along control connectors the weight of which returns "true" at that time. This is achieved via the map

$$\sigma:\mathbb{N} \times N \rightarrow p(N)$$

$$(i, A) \longmapsto \{B \in N \mid \exists\, (A, B, p) \in E : p({}^i\iota(p)) = 1\}$$

which is called *formal selection map*. The semantics of our meta-model do allow the formal selection map to apply only to activities that have terminated successfully. Moreover, only those members of $\sigma(i, A)$ that satisfy their associated synchronization expression $\Phi(A)$ are actually selected. Thus we have to define an additional map

$$\Sigma:\mathbb{N} \times N \rightarrow p(N)$$

called a *selection map* via

1. $A \in \lambda_i \Rightarrow \Sigma(i, A) := \{B \in \sigma(i, A) \mid \Phi(B)(p_1^B, \ldots, p_{k_B}^B) = 1\}$ with $\{p_1^B, \ldots, p_{k_B}^B\} = \mathbb{P}_{\Rightarrow}(B)$
2. $A \notin \lambda_i \Rightarrow \Sigma(i, A) := \emptyset$

$\Sigma_i(A) := \Sigma(i, A)$ is called the set of *i-actual successors* of A.

The navigation through a process consists of both the act of *selecting* follow-on activities and the act of *executing* selected follow-on activities. This strong distinction between selection and execution of activities permits the delegation and distribution of the follow-on activities of successfully terminated activities, and reflects the fact that delegated activities are in general not started immediately after their delegation.

Executable activities. Once a navigation step is finished the newly determined actual successors are dispatched to the resources that can execute them. The corresponding activities are "executable" (also called "startable"). All activities that were executable at an earlier time and that have not changed to the state "successful" in the meantime are still in the state "executable," of course. When an instance of the process model $G = (N, E, P, V, \Phi, \varepsilon, \Delta, \vec{\Delta}, N')$ is started, all of its entry activities N' can be executed. We describe these semantics via the set family $(\Sigma_i)_{i \in \mathbb{N}}$ called a ($\omega -$)*execution family* which satisfies the following conditions:

1. $\Sigma_0 := N'$
2. $\Sigma_i := \Sigma_{i-1} \cup \bigcup_{A \in N} \Sigma_i(A) - \{A \in \Sigma_{i-1} \mid \omega_i(A) = \text{successful}\}$

Process instances: The formal definition. Finally, we can now define what an instance of a process model is in the sense of our meta-model, i.e., how PM-graphs are interpreted:

Definition: Let $G = (N, E, P, V, \Phi, \varepsilon, \Delta, \vec{\Delta}, N')$ be a process model. A *G-process* or a *G-instance* or an *execution* of G is a tuple $G^\omega = (\omega, \Lambda, ({}^0\iota(A))_{A \in N}, ({}^0o(A))_{A \in N}, {}^0\iota(G), {}^0o(G))$ consisting of a status map $\omega:\mathbb{N} \times N \rightarrow S$, a dead value map $\Lambda:\mathbb{N} \times N \times V \rightarrow \cup\, \mathrm{DOM}(w)$, and the families of default values of the input containers and output containers of all activities as well as of the process model G.

A G-process is thus determined by a status map, a dead value map, and the default values for its containers. According to our definitions given for those terms, all other constructs are derived from that. Note that the status map ω is both influenced by user interventions (e.g., by starting the execution of an activity that results in a state transition from "executable" to "active" of the subject activity) and by interventions of the interpreting system itself (e.g., by changing the state of an activity to "successful" if the activity terminates and the exit condition results in "1").

Summary and future work

We described FlowMark, a system supporting both the modeling and the execution of business processes. The constructs provided by FlowMark to define process models and the operational meaning of these constructs assumed when executing instances of a process model are defined via a meta-model. The syntax and the semantics of this meta model were presented in an abstract manner showing the mathematical foundation of FlowMark. Processes are modeled by the potential control flow and data flow between activities of FlowMark. Activities can be represented by programs, and they are bound to the executing agents of FlowMark which are resolved into work requests sent to users at run time. FlowMark has a client/server structure in which persistent data are maintained by servers via an object-oriented database system. The implementation is fault-tolerant, e.g., FlowMark itself will not initiate abortions of processes; processes affected by soft crashes and hardware failures will be forward recovered.

Potential future extensions of the meta-model encompass additional colors of activities, and business transactions, for example. One new color could be an additional Boolean function that must be evaluated to "true" when trying to start the associated activity. This might avoid starting activities whose execution is no longer desirable because changes in the context of the process occurred before an activity is actually started. Business transactions could add the notion of compensation activities and spheres of compensation to process models.

Apart from these extensions FlowMark is considered to have been expanded to a multiserver product capable of servicing the business process needs of large enterprises. A logical step in this direction includes porting FlowMark to a number of other platforms besides OS/2 and exploiting other transport protocols.

Acknowledgment

FlowMark is a team effort; we are very grateful to each member of the FlowMark development team. The team made process management in the sense of this paper a reality.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. E. Best and C. Fernandez, "Nonsequential Processes—A Petri Net View," *EATCS Monographs on Theoretical Computer Science*, Vol. 13, Springer-Verlag, Berlin-Heidelberg (1988).
2. *High-Level Petri Nets: Theory and Applications*, K. Jensen and G. Rozenberg, Editors, Springer-Verlag, Berlin-Heidelberg (1991).
3. H. J. Genrich, "Predicate/Transition Nets," in *High-Level Petri Nets: Theory and Applications*, K. Jensen and G. Rozenberg, Editors, Springer-Verlag, Berlin-Heidelberg (1991).
4. K. Jensen, "Colored Petri Nets: A High-Level Language for System Design and Analysis," *High-Level Petri Nets: Theory and Applications*, K. Jensen and G. Rozenberg, Editors, Springer-Verlag, Berlin-Heidelberg (1991).
5. J. Guyot, "A Process Model for Data Bases," *ACM SIGMOD Record* **17**, No. 4, 22–30 (1988).
6. F. Kröger, "Temporal Logic of Programs," *EATCS Monographs on Theoretical Computer Science*, Vol. 8, Springer-Verlag, Berlin (1987).
7. U. W. Lipeck, *Dynamische Integrität von Datenbanken*, Springer-Verlag, Berlin (1989).
8. U. W. Lipeck and G. Saake, "Monitoring Dynamic Integrity Constraints Based on Temporal Logic," *Information Systems* **12**, 255–269 (1987).
9. A. M. Kotz, *Triggermechanismen in Datenbanksystemen*, Springer-Verlag, Berlin (1989).
10. H. Garcia-Molina and K. Salem, "Sagas," *Proceedings of ACM SIGMOD* (1987), pp. 249–259.
11. H. Wächter and A. Reuter, "The ConTract Model," in *Database Transaction Models for Advanced Applications*, A. K. Elmagarmid, Editor, Morgan Kaufmann Publishers, Inc., San Mateo, CA (1992).
12. M. Hsu, A. Ghoneimy, and C. Kleissner, "An Execution Model for an Activity Management System," *Proceedings 4th International Workshop on High Performance Transaction Systems*, Asilomar (September 1991).
13. C. T. Davies, Jr., "Data Processing Spheres of Control," *IBM Systems Journal* **17**, No. 2, 179–198 (1978).
14. J. Behrmann-Poitiers and J. Edelmann, "A Model to Support Routine Office-Work," *Proc. GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft*, Kaiserslautern, FRG (March 1991), Informatik-Fachberichte 270, Springer-Verlag (1991), pp. 72–88.
15. G. Kappel, "Reorganizing Object Behavior by Behavior Composition—Coping with Evolving Requirements in Office Systems," *Proc. GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft*, Kaiserslautern, FRG (March 1991), Informatik-Fachberichte 270, Springer-Verlag (1991), pp. 446–453.
16. D. Tsichritzis, "Form Management," *Communications of the ACM* **25**, No. 7, 453–478 (1982).
17. G. Chroust, H. Goldmann, and O. Gschwandtner, "The Role of Work Management in Application Development," *IBM Systems Journal* **29**, No. 2, 189–208 (1990).
18. G. F. Hoffnagle and W. E. Beregi, "Automating the Software Development Process," *IBM Systems Journal* **24**, No. 2, 102–120 (1985).
19. K. D. Saracelli and K. F. Bandat, "Process Automation in Software Application Development," *IBM Systems Journal* **32**, No. 3, 376–396 (1993).
20. G. Chroust and F. Leymann, "Interpretable Process Models for Software Development and Administration," *Proceedings of the 11th European Meeting on Cybernetics and Systems Research EMCR92*, Vienna, Austria, April 21-24, 1992, World Scientific (1992), pp. 271-278.
21. F. Leymann, "A Meta Model to Support the Modelling and Execution of Processes," *Proceedings of the 11th European Meeting on Cybernetics and Systems Research*

EMCR92, Vienna, Austria, April 21–24, 1992, World Scientific (1992), pp. 287–294.
22. ISO/IEC 10027: 1990 (E), Information Technology—Information Resource Dictionary System (IRDS) *Framework*, ISO, Geneva.
23. W. D. Potter, R. P. Trueblood, and C. M. Eastman, "Hyper-Semantic Data Modeling," *Data & Knowledge Engineering* **4**, 69–90 (1989).
24. J. Peckham and F. Maryanski, "Semantic Data Models," *ACM Computing Surveys* **20**, No. 3, 153–190 (1988).
25. *Concepts and Terminology for the Conceptual Schema and the Information Base*, Report of ISO TC97/SC5/WG5, J. J. Griethuysen, Editor, ISO, Geneva (1982).
26. F. Leymann, "Towards the STEP Neutral Repository," *Computer Standards & Interfaces* **16**, No. 2 (1994).
27. J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," *Computing Surveys* **5**, No. 1, 31–80 (1973).
28. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc., San Mateo, CA (1993).
29. H. M. Deitel and M. S. Kogan, *The Design of OS/2*, Addison-Wesley Publishing Co., Reading, MA (1992).
30. D. Moskowitz and D. Kerr, *OS/2 2.1 Unleashed*, Sams Publishing, Carmel, IN (1993).
31. R. Orfali and D. Harkey, *Client/Server Programming with OS/2 2.0*, Van Nostrand Reinhold Co., Inc., New York (1992).

Accepted for publication January 21, 1994.

Frank Leymann *IBM German Software Development Laboratory, Hanns-Klemm-Str. 45, D-71034 Böblingen, Germany (electronic mail: frank_ley@vnet.ibm.com).* Dr. Leymann studied mathematics, physics, and astronomy, and received an M.Sc. (Dipl.Math., 1982) and a Ph.D. (Dr.rer.nat., 1984), both in mathematics. He joined IBM in 1984 as a system programmer and database programmer. Next, as a team leader, he was responsible for the development of a nonstandard DBMS. Then, he worked out an architecture for systems facilitating work flow management. After having worked on enterprise modeling, repositories, and object-oriented database systems, he is now active in the area of database extensions, database tools, and advanced transaction processing. Dr. Leymann has published papers in various journals and conference proceedings on subjects such as relational database theory, universal relation model, hybrid DBMSs, DBMS architectures, meta-modeling, and work flow management; also, he is coauthor of a textbook on repositories. Since 1990 he has been teaching database courses at universities. He is a member of the German standards committee DIN NI21.3 (IRDS, RDA, SQL).

Wolfgang Altenhuber *IBM Vienna Software Development Laboratory, Lassallestrasse 1, A-1020 Wien, Austria (electronic mail: altenhub@vabvm1.vnet.ibm.com).* Mr. Altenhuber studied computer science and physics at the Technical University of Vienna and received an M.Sc. (Dipl.Ing., 1982) in computer science. Before joining IBM in 1988, he worked in the computing center of the city of Vienna where he was involved in a number of projects dealing with office automation, distributed processing, and networking. Within IBM he first worked in the area of interactive user interface definition tools and then became team leader for FlowMark in 1991. Presently he is responsible for the architecture and design of FlowMark. He is a member of the technical committee of the Work Flow Management Coalition.

Reprint Order No. G321-5545.