Flexible Modeling and Execution of Workflow Activities

Mathias Weske
Lehrstuhl für Informatik, Universität Münster
Steinfurter Straße 107, D-48149 Münster, Germany
weske@helios.uni-muenster.de

Abstract

While today's workflow management systems are well suited for the controlled execution of completely specified processes, support for dynamically changing processes is rather weak. However, new applications in the business domain and in non-traditional domains like the natural sciences or laboratory environments require support for flexibility like user interventions in workflow executions and dynamic modifications. Based on an activity meta model and an activity instance model, this paper discusses dynamic modifications and user interventions and shows how their implications to activity models and to concurrent and future activity instances can be described. Finally, we show how the basic concepts presented in this paper are realized in a prototypical implementation.

1 Introduction

Today's workflow management systems have been developed for modeling and controlling the execution of application processes, mainly in office environments [4, 17, 14, 19]. Since the target processes are typically completely specified and executed in a routine fashion, these systems support quite well the modeling and controlled execution of completely specified processes. On the other hand, the support for incompletely specified or dynamically changing processes is rather weak [3]. However, new applications in the business domain and applications in non-traditional domains, e.g., in the natural sciences or in laboratory or manufacturing environments, require enhanced flexibility, like support for dynamic modifications [22, 20] or controlled user interventions. In [18], a number of questions to enhance the applicability and flexibility of workflow systems were raised, including suitable languages and methodologies for flexible and dynamic modeling of workflow activities. In this paper we address some of these issues. In particular, based on an activity meta model we propose a set of dynamic change operations and user intervention operations and discuss how these can be used to support the controlled execution of potentially incompletely specified and dynamically changing complex activities.

The work we report on in this paper was carried out in the context of the WASA project, which aims at providing flexible workflow support for non-traditional applications, mainly in the scientific domain [20]. This paper is organized as follows. Section 2 discusses related work on process and workflow modeling. Section 3 provides a graph-based activity meta model and a model to describe activity instances. Section 4 discusses a sample application process, which requires flexibility in modeling and executing complex activities. In Section 5, flexibility properties of workflow management systems are described in terms of supported operations. Section 6 discusses how dynamic changes of activity models are supported by the WASA prototype. Concluding remarks complete this contribution.

2 Related Work

Representations of application processes to be used by workflow management systems to control the execution of workflow instances are known as workflow models. The structure of workflow models is defined by workflow meta models, which define the components of workflow models and their relationships. There is not a universal workflow meta model which is generally agreed upon – the variety of workflow management systems in the market today is reflected by the number of different workflow meta models. We now briefly review important approaches to modeling and executing workflows and discuss how these are related to our work.

IBM's workflow management system FlowMark [9, 12] uses process graphs to define workflow models. In FlowMark, each workflow model is specified by a directed graph, whose nodes represent activities and whose edge set is partitioned in a set of control flow edges and a set of data flow edges. Control flow edges represent potential control flow, defined by transition conditions, which are predicates evaluated at run time. Activities may have typed input and output parameters, and data flow edges connect parameters of different activities. FlowMark is based on a separation of

1060-3425/98 $10.00 (c) 1998 IEEE

a workflow's built time and its run time. Workflows are modelled during built time and executed during run time. In particular, workflow models may not be changed after built time. Therefore, support for flexibility is rather limited; e.g., dynamic modifications of workflow models are not supported by FlowMark. In addition, users may not change control flow of active workflow instances, e.g., by stopping or skipping certain activities. Since this is rather a limitation of the FlowMark system than of the workflow modeling language and since graph-based representations of workflow models are intuitive, the activity meta model presented in this paper is based on process graphs and enhances them to support flexibility in activity modeling and execution.

Another important category of workflow approaches use enhanced Petri-nets to model workflows. The Funsoft-approach [5] is based on higher Petri-nets. An interesting property of this approach is that Funsoft nets can be used from early phases of business process modeling until later phases of workflow modeling and execution. In a recent paper, the suitability of Funsoft nets for enhancing the flexibility of modeling and executing workflows is investigated [6]. In that contribution, approaches to enhance flexibility based on modeling sub-nets during executions and "flexibility by variants" are discussed, where dynamic modeling is governed by analyzing activity transitions, considering persons and data involved. Ellis et al. [3] use more traditional Petri-nets to specify workflows. In particular, they present a formalism to cope with dynamic modifications, focusing on structural dynamic changes of procedures, like the concurrent execution of formerly sequential steps. The changes considered are restricted to isolated procedures, i.e., the implications of performed changes to other activity models or activity instance are not investigated. The Mobile approach uses programming language constructs to specify workflows [11], and workflow models are represented by programs, written in the Mobile language. This project emphasizes on modularity of workflow aspects and system development rather than on flexibility issues.

The statechart formalism is an extension of finite state machines; it was developed by Harel [7] for specifying the behavior of reactive technical systems; to describe these systems, statecharts specify states and state transitions while accompanying activitycharts describe events that may lead to state transitions. Provided with a formal semantics and with a tool (Statemate [8]), statecharts are used in designing technical systems, like remote control systems or car radio systems. The Mentor project [23] makes use of state- and activitycharts to model workflows. This project emphasizes on scalability and correctness of distributed workflow executions; it uses statecharts to partition workflow models into smaller units to be processed in a distributed environment [23]. Provided with a formal semantics, it is shown that workflow specifications described by statecharts and their partitioning to be used for their distributed execution are equivalent [24].

The work of Craven and Mahling [2] stems from the area of computer supported cooperative work. In particular, they analyze the relationship between project management and workflow management. Fundamental commonalities between the two areas are discovered, namely coordination requirements, dynamic modifications, and re-use of activity models. While the need for dynamic modification is identified, Craven and Mahling do not elaborate on this aspect. Instead they put the main focus on the specification and decomposition of tasks and goals, on maintaining domain knowledge and on coordination requirements of agents. Reichert and Dadam present ADEPT$_{flex}$, an approach for controlled dynamic modifications of workflow specifications based on non-nested, symmetric workflow specifications [15].

As indicated above, our approach to modeling and executing flexible workflows is based on nested process graphs, similar to those used in FlowMark. Our formalism extends that approach to explicit modeling of activity modeling operations, activity instances, and operations to allow users to intervene in system-controlled activity instances. By including modeling operations (like adding or deleting activity models), we are able to specify which dynamic modeling operations are valid in which state of an activity execution. Thereby we aim at providing an environment which supports users in executing complex activities in a flexible manner, involving controlled user intervention to allow flexible reaction to unforeseen events and dynamic changes of activity models.

3 Basic Model

In general, meta models describe how models are structured. In our context, an activity meta model describes how activity models are built; using activity models, workflow management systems control the execution of activity instances.

3.1 Modeling Activities

To model application processes as workflows with the aim of controlling their execution, a suitable formalism has to be provided. In this section, we present an activity meta model which is based on process graphs. In general, activities are units of work as percepted by the modeler. Activities are specified by activity models, and each activity model includes a description and the types of the data used and generated by it. Activity models are maintained in an activity model library, represented by a set $M = \{i \mid i \geq 1\}$ of activity models. This library is partitioned in a set A of atomic activity models and a set C of complex activity

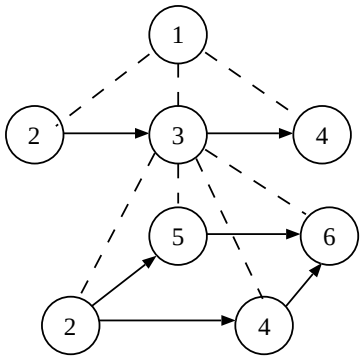

Figure 1. Nested Structure of Complex Activity Model

models. As indicated by this terminology, atomic activity models do not have an internal structure, while complex activity models do have an internal structure: Each complex activity model consists of a set of (atomic or complex) activity models and control flow and data flow constraints.

Activity models are represented by directed graphs, activity model graphs, whose nodes represent activity models and whose edges represent relationships between activity models. An atomic activity model is represented by a graph with a single node. The activity model graph of a complex activity model k contains multiple nodes, component activity models, which may be related to each other by directed edges, representing control flow and data flow. For each activity model $i \in M$, the list of input parameters is denoted by $ip(i)$; $op(i)$ refers to the list of its output parameters. We assume that a data flow edge can only connect parameters of sub-activity models i and j of a complex activity model k if there is a control flow path (consisting of one or more control flow edges) connecting the two and the types of the parameters connected are compatible. We use the bundle construct [12] to model parallel execution of a number of activity instances of a given activity model, whose quantity is evaluated at run time. In this paper we do not consider cyclic structures in activity models.

Figure 1 shows a nested activity model graph for a complex activity 1. Complex activity model 3 is a component activity model of 1, the other component activity models are atomic. (Component activity models are connected to their parent by dotted lines; control flow edges are represented by solid arrows; data flow constraints are not displayed explicitly). Each activity model can be used in multiple complex activity models. In Figure 1, e.g., activity model 2 appears in complex activity models 1 and 3.

We assume that each atomic activity model can be executed by an agent, typically a person or a software system or a person using a software system [17]. We assume that agents are skilled and competent to perform requested activities. Since this paper is centered around flexibility issues, it does not elaborate on a role concept [12].

Activity models are created using the following set of activity modeling operations:

- $CreateAtomic(i)$: Create an atomic activity model i, including the definition of persons and application programs to perform i and sets of input and output parameters with their respective data types.
- $CreateComplex(k)$: Create a complex activity model k, including the definition of input and output parameters.
- $AddActivity(j,k)$: To a given complex activity model k add an (atomic or complex) activity model j as a component activity model, assuming both j and k are already existing.
- $DelActivity(j,k)$: Delete component activity model j from complex activity model k. This involves the deletion of edges adjacent to j in k.
- $AddEdge((i,j),k)$: In k, add an edge $i \rightarrow j$, where i and j are component activity models of k.
- $DelEdge((i,j),k)$: In k, delete an edge $i \rightarrow j$.

In summary, an activity model is represented by a nested directed graph whose nodes are activity models and whose edges represent control flow and data flow constraints.

3.2 Modeling Activity Instances

Activity instances correspond to real world processes, in which typically a number of persons and software systems are involved. In general, an activity instance is created whenever a complex or an atomic activity is started. Executions of activity instances are controlled by workflow management systems, using activity models. In this section we provide a representation of activity instances, based on operations to manage them.

Activity instances are executed according to execution rules, described as follows. Consider an activity instance based on a complex activity model k. The execution starts by retrieving the activity model graph from the activity model repository. All component activity models of k which do not have any incoming edges are retrieved and instantiated, and the input parameters are set up as defined. After the termination of a component activity, its output parameters are used to evaluate the transition conditions of its outgoing edges. Now, component activities for which all transition conditions of incoming edges evaluate to true can be performed. This process iterates until all component activity models of k are executed or will not be executed in that particular case.

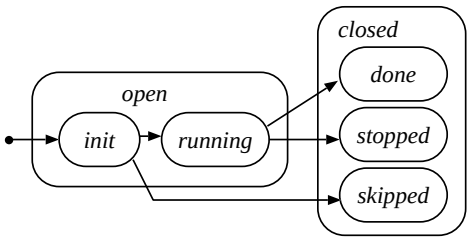

Figure 2. State Transition Diagram.

An activity instance based on the activity model depicted in Figure 1 is characterized as follows: After retrieving activity model graph 1, an activity instance for component activity model 2 is instantiated and executed. Assuming 2 to be an atomic activity, the system determines an application program and a person to execute that activity. On its termination, the activity provides output parameters which may be used to evaluate the transition condition of control flow edge $2 \rightarrow 3$. Assuming the transition condition evaluates to true, the system creates an activity instance based on activity model 3, during which activity 2 is executed, followed by the concurrent execution of 4 and 5, which in turn is followed by activity 6. This brings complex activity 3 to an end, followed by the execution of 4, which in turn completes complex activity 1. In the remainder we use the term 'activity i' to indicate activity instance based on activity model i.

The following activity instance operations are available:

- $CreateActivity(j)$: This operation instantiates an activity according to activity model j; the identifier of the created activity is returned, let's say i_j.
- $StartActivity(i_j)$: Starting an atomic activity instance involves setting up the input parameters and the application program to execute it. If the activity does not require human interaction then the activity instance is executed under control of the workflow management system. If a person is involved in the execution of the activity then a work item is put on the work item list of that person, and the human executes the activity after selecting the work item.
- $TerminateActivity(i_j)$: When an atomic activity completes or no more sub-activities of a complex activity have to be performed, an activity instance is terminated.

Activity instance operations trigger state transitions of activity instances, displayed in a simple transition state diagram using nested states (cf. Figure 2). An activity can be either in state *open* or *closed*, indicating not completed and completed, resp. In particular, when an instance is created using the CreateActivity operation, it enters state *init*; StartActivity triggers the state transition to *running*. Finally, TerminateActivity brings the activity instance in state *done*. (States *stopped* and *skipped* will be discussed shortly.)

4 Sample Application

Our example is based on a business process in manufacturing, similar to one discussed in [6]. Consider a company which manufactures complex products. These products are assembled from numerous parts, some of which are manufactured locally, while others are supplied by remote companies. A common activity in these settings is processing a request by a customer: "When and at which price can complex product P be delivered?" To respond to this request, a complex application process is started, in which numerous persons located at different sites are involved.

Informally, this process can be described as follows: Assume a customer requests the earliest shipping date and the price of a complex product P. When the request is submitted, first the sub-parts of P have to be determined, including their respective sub-parts. The availability of these parts has to be checked next. For each part not available in stock, the respective supplier has to be determined, and requests have to be sent to determine when at which price the missing parts are available. After collecting the responses from the suppliers, the local manufacturing capacities are analyzed, and a time slot for the (potential) production of the complex product is reserved. Finally, the date of potential shipping is calculated. To respond to the initial request, this information is passed to the customer, who by then decides whether to order the complex product. If the customer decides to order the product, the requested parts are effectively ordered and the reserved time slot for manufacturing is confirmed. If the customer decides to cancel the order then the requests for the sub-parts have to be cancelled, and the time slot for manufacturing the product is no longer reserved.

4.1 Modeling and Executing Sample Process

Using the activity meta model presented above, the application process can be specified by a complex activity model *Request Delivery Date* as shown in Figure 3. The top-level activity model consists of five activity models (one of which is complex), executed sequentially. After determining the sub-parts of the complex product requested (*determine sub-parts*), the availability of each of them is checked and times of availability are determined (*check avail*). Since this activity has to be performed for each sub-part, whose number is not known before the activity starts, we use the bundle construct to model it; the number of parallel instances of *check avail* is given by the number of sub-parts of the complex product, calculated during the execution of *determine sub-parts*.

Checking the availability and calculating the time of potential arrival of the sub-parts involves the following steps:

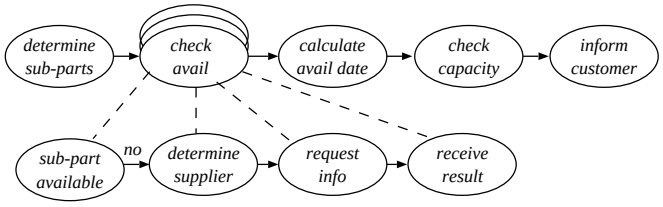

Figure 3. Sample Activity Model.

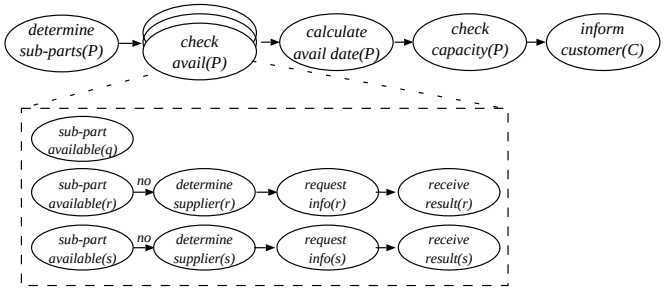

Figure 4. Sample Activity Instance.

In *sub-part available* we check if the sub-part is in local stock. If so, no further requests have to be made, and that activity instance terminates. If it is not in stock (indicated by edge label *no*) then the supplier has to be determinated (*determine supplier*), followed by requesting the earliest date for shipping and the price of the missing part (*request info*). This activity is implemented by sending a request to the respective supplier. When an answer is received (*receive result*), that activity instance terminates. When all activity instances of the bundle have terminated, the earliest availability date of all parts needed (*calculate avail date*). The local manufacturing capacities are analyzed next, and a time slot for the potential production is reserved. Finally, the customer is informed of the delivery date of the product and its cost.

Notice that the description of the application process as presented above is rather simplified. Besides higher complexity, in real settings situations in which the process has to be adapted to changes of the environment are likely to occur frequently, e.g., due to unforeseen behavior of customers, suppliers, or to changes in company policies. This scenario provides a setting in which we will discuss flexibility issues in activity modeling and execution.

We now discuss an activity instance of activity model *Request Delivery Date*. The activity instance created when a customer C requests a product P, consisting of sub-parts q, r, s is shown in Figure 4. To distinguish activity instances from activity models, we add data information to graphical representations, e.g., the node representing the determination of the sub-parts of product P is marked *determine sub-parts(P)* in Figure 4. Creating and starting the top-level activity instance is followed by creating and starting *determine sub-parts(P)*. After its termination, a bundle consisting of instances *check avail(x)*, $x \in \{q, r, s\}$, is started. Assuming q is in stock and r, s are not, *check avail(q)* terminates immediately after *sub-part available(q)*. However, r and s have to be ordered. After determining the suppliers of r and s and requesting and receiving the respective delivery information, the bundle-activity instance *check avail(P)* completes. On the top-level, calculating the arrival date and reserving local capacities for the production of P and informing the customer completes the complex activity instance.

4.2 Flexibility Requirements

The application process discussed above simplifies real world processes considerably. There is a myriad of unforeseen events which may lead to failure of the application process. These events cannot be modelled completely before activity instances start. Sample unforeseen events which might occur in our setting can occur on the side of the customer, the supplier, and on the side of the company performing the application process. We discuss sample exceptions for these participants in turn.

Supplier So far we have assumed that each supplier answers timely to a request and is able to deliver requested parts within acceptable time spans at acceptable prices. However, if suppliers do not answer timely, the application process gets stuck, and the system is no longer able to control its execution. What may happen in this situation is that people involved in the execution of that process decide to stop the activity and to continue it on a manual basis, which provides the required flexibility, but without system control. Besides not responding, the supplier may not be able to deliver the requested part in an acceptable time span. In this case the user may decide to consult the customer whether a later delivery date is acceptable. If so, the activity execution continues as specified by the activity model. In addition, the user may start other activity instances to determine alternative suppliers. If an activity model suitable for this purpose is present in the activity model library then the user shall be supported in localizing and starting it. If a suitable activity model is not present then the user may want to define such an activity model. The newly created activity model will then be available in future activity instances.

Customer Customers are also a potential source of non-anticipated behavior. For instance, if and when a customer decides to cancel an order, the corresponding activity has to be cancelled, followed by undoing effects of the activity like undoing the reservation of sub-parts or the reservation of time slots for manufacturing. In another scenario, a customer provides a latest shipping date which then may lead

to prematurely stopping the activity instance during execution, if one or more sub-parts are not present in time and no alternative supplier can be found. Users shall be able to stop activity instances which are no longer needed. This may be followed by starting the *inform customer* activity and billing the customer for the work that was performed on his behalf.

Company Besides changes in the market environment of the company, i.e., with suppliers and customers, there may be changes in the policies of the company, e.g., due to re-engineering projects, aiming at optimizing application processes. Changes include parallel execution of activities which have been executed sequentially formerly. In general, changes in the organization of the company typically result in dynamic changes of the corresponding activity model. Once the change is applied, all future instances of that activity will be effected by that change. Another form of non-anticipated behavior occurs if the user learns from external sources (e.g., by a phone call or by documentation material) that a supplier is able to deliver the missing part timely. In this case, requesting the date and receiving the result do not have to be executed. Since the shipping information is already available, the activities *request info* and *receive result* can be skipped in that situation. Hence the user should be able to select activities to be skipped in the particular activity instance.

5 Functionality to Support Flexibility

We now identify a set of operations, which a workflow management system has to support in order to satisfy the flexibility requirements discussed in the previous section. In general there are two types of flexibility operations, namely user intervention operations and dynamic modification operations. With user intervention operations, users may actively intervene with the system-controlled execution of activities, i.e., by changing the predefined control flow of activities. Dynamic change operations may be used to allow the modification of activity models while activity instances are executing.

5.1 User Intervention Operations

By user intervention operations we mean operations to change the control flow in the execution of activity instances by users. Operations involve skipping, stopping or repeating activities. However, user interventions do not involve changes to activity models. Therefore, their effects are limited to the activity instance during which the intervention occurred. To provide this new functionality, the set of activity instance operations as proposed in Section 3 is enhanced by user intervention operations *SkipActivity*, *StopActivity*, and *RepeatActivity*.

5.1.1 SkipActivity

Activities which are in state *init* can be skipped, which triggers a state transition to *skipped* (cf. Fig. 2). Hence, an activity can be skipped only before its execution starts. Skipping activities allows users to save time and effort for activities which are not needed during a particular case. However, skipping activities presents data-related issues, as shown in the following example.

Consider the activity model shown in Figure 5. Skipping activity k, for instance, results in starting l immediately after j terminates. If there is a data flow defined from k to l and k is skipped then data needed for the execution of l is not available. In addition, transition conditions of the outgoing edges of the skipped activity cannot be evaluated due to missing data. To cope with these issues, whenever an activity is skippable, information on how to provide data needed in the remainder of the complex activity has to be defined in the activity model. Possible solutions are providing data by default values, entering data manually [15] or mapping data parameters, i.e., using data provided by previously executed activities. As an example, consider activity j gets customer data while k validates the data and provides the validated data to activity l. Hence, customer data flows from activities j to k and from k to l. If the user decides that checking the data of a particular customer is not needed then he or she may skip activity k. In this case, the output parameter of j can be mapped to the input parameter of l, and the customer information is passed directly from j to l.

5.1.2 StopActivity

Running activities can be stopped using the StopActivity operation. In terms of states of activity instances, this operation triggers a state transition from *running* to *stopped*. There are different forms of stopping an activity. The first form corresponds to stopping an activity and resuming execution with the next activity, as defined by the activity model. In this case, data issues with transition conditions and input parameters of the next activity emerge, as was discussed in the context of the SkipActivity operation.

In Figure 5, assume activities j and z are active concurrently when activity j is stopped. In the first form of stopping, execution is resumed with k. After l (and z and p) are completed, the execution of y completes the complex activity. In the second form, the execution path starting from the stopped activity will not be executed, i.e., activities k and l will also be skipped. In this case, after stopping the activity, dead path elimination has to be performed to make sure later parts of the complex activity will be executed properly. (Recall that dead path elimination [12] is used to deal with paths whose nodes are not and will not be instantiated.) After z and p are completed, the execution is resumed with y. Dead path elimination makes sure y does not wait for the completion of l. In addition, stopping an activity may rule

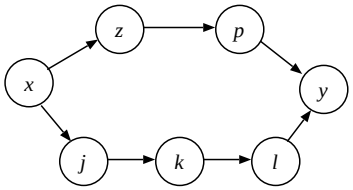

Figure 5. Complex Activity Model.

out the successful completion of the complex activity and therefore leads to stopping the complex activity. If stopping j rules out the successful completion of the complex activity then the complex activity has to be stopped.

5.1.3 RepeatActivity

Repeating an activity can be described as "manually starting an activity which was executed already". Repeating an activity cannot easily be captured in state transition diagrams, since these diagrams specify state transitions of a given activity instance, and repeating an activity creates a new activity instance. The newly created activity instance will be executed independently from the earlier instances of that activity.

We assume that only sequential executions are allowed, i.e., one repetition can start only after the previous one has completed. Data-related issues also emerge in the context of repeating activities. Besided the options discussed above, input data from former (e.g., the first, most recent) instances of the repeated activity can be used. Consider a sequential execution of activities j, k and l, shown in Figure 5. Assume after the execution of l the user decides to continue the complex activity with repeating j. Then after the termination of l, another instance of j is created. When this instance terminates, the execution continues with repetitions of activities k and l. Notice that the order of execution seems to violate the activity model, since j should execute before l, but an instance of j is started after an instance of l has terminated. Since we assume that repetitions are sequentially, for each repetition the control flow constraints as defined in the complex activity model are satisfied.

Assume z and l are active and the person performing the latter decides to repeat j (cf. Fig. 5). This operation can be granted since there is no violation of control flow constraints. After the repetition of j is completed, new activities k and l are created, and the two concurrent strands of the execution finally meet in activity y. There are restrictions on which activities to repeat depending on the state of the activity execution. Assume after completing l, the user decides to repeat an instance of activity x. In this case, z is still active, while an instance of x is also active. This situation violates control flow constraints, since activities x and z are defined to execute sequentially but in fact they are executed concurrently (although they belong to different repetitions). In order to avoid this situation, we assume that after the termination of an instance k, an activity j can only be repeated, if all outgoing edges of j reach k. For instance, in Figure 5, jumping back from l to j obeys the assumption. Repeating x from l, however, violates it and is therefore not allowed: There is a path starting in x and not passing k. On the contrary, repeating x from y is feasible, since all paths starting from x finally reach y.

5.1.4 Use of Operations

We now briefly comment on how to use these operations in the scenario described above. Stopping and repeating an activity requesting the date of delivery of a sub-part q from a particular supplier can be implemented by *StopActivity(request info(q), check avail(P)), RepeatActivity(request info(q), check avail(P))*. When the user knows that the supplier can provide a missing part r timely (e.g., based on external information) then after determining the supplier, the activities *request info(r)* and *receive result(r)* can be skipped: *SkipActivity(request info(r), check avail(P)), SkipActivity(receive result(r), check avail(P))*. When the customer withdraws the request on complex product P while the complex activity instance is active then all active instances can be stopped. The stopping is followed by a start of the activity *inform customer(C)*, which acknowledges to the customer the cancelling of the activity and sends an invoice (subject to company policies).

5.2 Dynamic Change

Besides manually intervening in activity executions, dynamic changes of activity models are also an option. In general, changes are performed using modeling operations, as discussed in Section 3.1, involving the creation and deletion of activity models and relationships between activity models.

5.2.1 Dynamic Change Operations

To perform a dynamic change operation, the user invokes a dedicated activity modeling activity. During this activity, the operations *AddActivity, DelActivity, AddEdge* and *DelEdge* are available within a complex activity k.

AddActivity Adding an activity to an existing complex activity is done by performing the AddActivity operation: $AddActivity(j, k)$ adds an activity j as a sub-activity of k. Adding an activity specifies how it is embedded in the activity model. This is done by specifying which activities have to be completed before it starts and which activities can start only after it completes. Notice that no activity of the latter kind can be in state *running* when the dynamic change occurs.

DelActivity Activities can be deleted using the DelActivity operation: $DelActivity(j, k)$ purges activity j from complex activity k. This operation can only be executed before j begins execution. In this case, execution can resume with the follow-up activity of the deleted activity. Notice that data-related issues as discussed in the context of user intervention operations re-emerge in this context.

AddEdge Adding control flow constraints between component activities can be done using the AddEdge operation: $AddEdge((i, j), k)$ adds a control flow edge $i \rightarrow j$ to complex activity k; $(i \rightarrow j)$ can be added to k only when j has not yet started; activity i can be in any state, including closed states. Adding this constraint at run time, however, is useful when both activities have not yet started. In this case the system makes sure that the start of j is delayed until i is completed.

DelEdge When edges are deleted using the DelEdge operation, constraints on the execution of sub-activities are relaxed. $DelEdge((i, j), k)$: In k, delete control flow edge $i \rightarrow j$, resulting in an independent execution of the two activities.

In Figure 5 assume we want to add an activity m to be executed after z and before p. This can be done if and when p has not started execution, i.e., if p is in state init. The insertion of the activity is then done by adding activity m, adding edges $z \rightarrow m$ and $m \rightarrow p$ and deleting $z \rightarrow p$. Data flow can be defined from, e.g., z to m and from m to p. Notice that existing data flow constraints between z and p present in the original activity model can be retained. Given the original activity model as shown in Figure 5, p can be deleted in case p has not started yet. This involves the deletion of edges $z \rightarrow p$ and $p \rightarrow y$ and adding edge $z \rightarrow y$. Data flow constraints have to be handled as was discussed in the context of the SkipActivity operation.

5.2.2 Scope of Dynamic Changes

An important issue in dynamic modification addresses the implication of a dynamic change operation to other activity models and, thereby, to other activity instances, characterized by the scope of dynamic changes. The scope is local, if the change applies only to the instance, in which the dynamic change was conducted. The scope is global, if it applies to the activity model and, hence, to all future activity instances using the changed model.

If the scope is local then other activity models using the changed activity model or other activity instances, e.g., concurrently executing instances, are not affected by the dynamic change. This requires to create a new version of the changed activity model which is used only during the execution of the complex activity instance during which the dynamic change operation was conducted. If the scope is global, the dynamic change can be done "in place", namely the original activity model j can be overwritten by the new activity model x. Notice that j is assigned the value of x, i.e., j keeps its identifier. Since complex activity models store references to the activity models used, the overwriting of j suffices to perform the global change.

We point out that special care has to be taken to control concurrent activities of the changed model in presence of global changes. Depending on the state of these activities, the global change either applies to them or does not apply, i.e, if the changed part started execution already. In terms of an implementation, activities which are not effected by a dynamic change have to be controlled based the original activity model, which requires that a local copy of the original activity model has to be provided to the activity instance.

5.2.3 Use of Dynamic Change Operations

We now briefly discuss how these operations can be used in our sample application. During the execution of the toplevel workflow, the user may learn that another company now also produces a missing sub-part q. The user now decides to request shipping information from that supplier. Assume this company is able to ship the missing part timely, there is a new option of getting part q. The person now decides to have two alternatives for getting part q. Each future instance can check in parallel which alternative will be most appropriate. This situation corresponds to a change of the activity model using the dynamic modeling operations specified above.

6 Systems Considerations

The WASA-Project [22, 20] aims at supporting workflow management in non-standard application areas, among which scientific applications play a major role, e.g., in the domains of molecular biology [13], geo-processing [1], and laboratory management [16]. Based on a generic WASA architecture, we have developed a prototype of a workflow management system, which allows the coordinated execution of complex activities defined by workflow models and provides a high degree of flexibility and platform independence. In this section we focus on flexibility issues by very briefly discussing the conceptual design of the prototype and sketching how dynamic modification operations are supported.

In the current WASA prototype, workflow models are stored in a relational database system; the workflow server is implemented in Java, and the database is accessed via a JDBC interface. Hence, we are not restricted to a specific database system but are basically free to use any relational database system. When a top-level workflow with a number of sub-workflows is started, the workflow server retrieves the respective workflow model from the database. The models of the sub-workflows, however, are not yet re-

trieved: They are read successively during the execution of the workflow instance, as they are needed. This approach allows for dynamic modification; namely the implementation of sub-workflows will be done at the run time of the workflows.

We have implemented a basic dynamic change activity, which allows the changing of workflow models while instances of this model are active. In particular, it allows to dynamically modify future parts of the workflow model. Given a top-level workflow and a corresponding workflow instance, the refinements of the component workflow models can be done while the workflow runs. In particular, defining the refinements can be done by workflow applications invoked within the changed workflow instance. The prototype supports model changes only. As an ongoing research project, the WASA prototype will be enhanced to support instance changes and local change as well; work on implementing user intervention operations, like the stopping, skipping and repeating of activities, is under way. We do not go deeper into the discussion of the prototype but refer the reader to [21, 20].

7 Conclusions

This paper proposes a graph-based activity meta model, which allows the definition of atomic and complex activity models and the specification of data flow and control flow constraints between activity models. Unlike other approaches, we explicitly model operations to model activities, like the creation or deletion of activity models. These operations are not only available when activity models are initially built, but also when activity instances based on that model are executing, realizing dynamic modification operations. Using an application process from the area of manufacturing, we discuss how activity models can be specified and what kinds of flexibility issues arise in that context. Basically, two sorts of flexibility requirements emerge, namely active user intervention and dynamically modifying activity models. In this paper we define a set of user intervention operations and a set of dynamic change operations, which are suitable to support the flexibility requirements imposed by the sample application. While different application areas require different flexibility support, we believe that our approach as introduced in this paper meets some of the key features in flexible workflow management.

In this paper we have been looking into the technical aspects of activity modeling and flexibility issues. Using dynamic modifications, however, has severe sociological and organizational consequences: Agents can be given new responsibilities to organize their work, and an agent rises from an executer of system-defined work to a person responsible for his or her work, with the freedom to change local working procedures, which may even have global effects. In this context, a number of questions emerge, ranging from quality management of activity models w.r.t. dynamic changes to the management of scopes and maintenance of agent's access rights. As pointed out in [18], sociological and organizational consequences of dynamic changes require research efforts from a variety of disciplines. This contribution aims at enhancing the flexibility of workflow management systems. The concepts presented herein will hopefully lead to more flexible workflow systems, to be used in a variety of application areas for which workflow management seems too restrictive in today's systems.

Acknowledgements: The author is grateful to Gottfried Vossen for valuable comments on an earlier version of this paper.

References

[1] C. Bauzer Medeiros, G. Vossen, M. Weske. *GEO-WASA: Supporting Geoprocessing Applications using Workflow Management*. In Proc. 7th Israeli Conference on Computer Systems and Software Engineering, Herzliya, Israel 1996, 129–139, IEEE Computer Society Press, Los Alamitos, CA.

[2] N. Craven, D. Mahling. *A Task Basis for Projects and Workflows.* In Proc. Conference on Organizational Computing Systems (COOCS), Milpitas, CA 1995, 237–248.

[3] C. Ellis, K. Keddara, G. Rozenberg. *Dynamic Change Within Workflow Systems.* In Proc. Conference on Organizational Computing Systems (COOCS), Milpitas, CA 1995, 10–22.

[4] D. Georgakopoulos, M. Hornick, A. Sheth. *An Overview of Workflow Mangement: From Process Modeling to Workflow Automation Infrastructure*. Distributed and Parallel Databases, 3:119–153, 1995.

[5] V. Gruhn. *Validation and Verification of Software Process Models.* Ph.D. Thesis, University of Dortmund, 1991. Available as Technical Report No. 394/1991, University of Dortmund, Germany.

[6] J. Hagemeyer, T. Herrmann, K. Just-Hahn, R. Striemer. *Flexibility in Workflow Systems (in German)*. Software-Ergonomie '97, 179–190, Dresden, March 3–6 1997.

[7] D. Harel. *On Visual Formalisms.* Communications of the ACM 31(1988), 514–530.

[8] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* Part No. D-1100-43, i-Logix Inc., Andover, MA 01810. 1996.

[9] IBM. *IBM FlowMark: Modeling Workflow, Version 2 Release 2*. Publ. No SH-19-8241-01, 1996.

[10] Y. Ioannidis (ed.). *Special Issue on Scientific Databases*. Data Engineering Bulletin 16 (1) 1993

[11] S. Jablonski, C. Bußler. *Workflow-Management: Modeling Concepts, Architecture and Implementation* International Thomson Computer Press, 1996.

[12] F. Leymann, W. Alterhuber. *Managing Business Processes as an Information Resource*. IBM Systems Journal 33, 1994, 326–347.

[13] J. Meidanis, G. Vossen, M. Weske. *Using Workflow Management in DNA Sequencing*. In Proc. 1st IFCIS International Conference on Cooperative Information Systems (CoopIS), Brussels, Belgium 1996, 114–123, IEEE Computer Society Press, Los Alamitos, CA.

[14] C. Mohan. *State of the Art in Workflow Management System Research and Products*. Tutorial notes, 5th International Conference on Extending Database Technology, Avignon, France 1996.

[15] M. Reichert, P. Dadam. *A Framework for Dynamic Changes in Workflow Management Systems.* Proc. 8th International Workshop on Database and Expert Systems Applications 1997, Toulouse, France. IEEE Computer Society Press, 42–48.

[16] T. Reuß, G. Vossen, M. Weske. *Modeling Samples Processing in Laboratory Environments as Scientific Workflows.* Proc. 8th International Workshop on Database and Expert Systems Applications 1997, Toulouse, France. IEEE Computer Society Press, 49–55.

[17] M. Rusinkiewicz, A. Sheth. *Specification and Execution of Transactional Workflows*. In Kim Won (editor): Modern Database Systems The Object Model, Interoperability, and Beyond, Chapter 29, pp 592–620. ACM Press, 1995.

[18] A. Sheth, D. Georgakopoulos, S.M.M. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, A. Wolf. *Report from the NSF Workshop on Workflow and Process Automation in Information Systems*. Technical Report UGA-CS-TR-96-003 University of Georgia, Athens, GA, 1996.

[19] K.D. Swenson, K. Irwin. *Workflow Technology: Tradeoffs for Business Process Re-engineering*. In COOCS'95, 22–29, Milpitas, CA, 1995.

[20] G. Vossen, M. Weske. *The WASA Approach to Workflow Management for Scientific Applications.* NATO ASI Workshop, Istanbul, August 12–21, 1997. To appear in Springer ASI NATO Series.

[21] G. Vossen, M. Weske, G. Wittkowski. Dynamic Workflow Management on the Web. Technical Report Angewandte Mathematik und Informatik 24/96-I, Universität Münster, 1996.

[22] M. Weske, G. Vossen, C. Bauzer Medeiros. *Scientific Workflow Management: WASA Architecture and Applications*. Technical Report Angewandte Mathematik und Informatik 03/96-I, Universität Münster, 1996.

[23] D. Wodtke, J. Weissenfels, G. Weikum, A. Kotz Dittrich. The Mentor Project: Steps Towards Enterprise-Wide Workflow Management. In Proc. 12th IEEE International Conference on Data Engineering (1996), 556–565

[24] D. Wodtke. *Modeling and Architecture of Distributed Workflow Management Systems*. Ph.D. Thesis, University of Saarbruecken, 1996.