Defining Flexible Workflow Execution Behaviors

Gregor Joeris
Intelligent Systems Department, TZI - Center for Computing Technologies
University of Bremen, PO Box 330 440, D-28334 Bremen
joeris@informatik.uni-bremen.de

Abstract. *In this paper, we focus on the definition and adaptation of the execution behavior of a task in order to support flexible workflows in the presence of distributed workflow enactment. We argue that an adequate behavior definition is the basis for both, modeling less-restrictive workflows in advance as well as supporting dynamic workflow changes. We show how different control flow dependency types can be specified in our approach and can be used to define less-restrictive workflows. Furthermore, we discuss the definition of an adequate behavior for dynamic modifications in different situations. In particular, we describe how the application of and reaction to dynamic changes can be adapted in our approach depending on the process context and the behavior of a task itself.*

1. Introduction

The development of process-model based workflow management systems (WFMS) has been driven mostly by focussing well-structured business processes from the viewpoint of transactional processing. WFMSs are applied following a rigorous methodology of business process (re-)engineering and formal workflow specification which leads to well-defined processes and is well-suited for production workflows. This restricted view – which is not inherently caused by the workflow paradigm – is the main reason for the inflexibility of today's WFMS and their underlying process representation formalisms. Support for flexible workflows in process-model based WFMS has to cope with two fundamental challenges:

(a) *A-priori flexibility* focus on the specification of a flexible workflow execution behavior to express an accurate and less restrictive behavior in advance; flexible and adaptable control and data flow mechanisms have to be taken into account in order to support ad hoc and cooperative work at the workflow level (cf. [ElNu96]).

(b) *A-posteriori flexibility* (flexibility by dynamic adaptation) is provided by the change and evolution of workflow models in order to modify workflow specifications on the schema and instance level due to dynamically changing situations of a real process (cf. [EKR95, CCPP96, ReDa98, JoHe98]). Note, that in the case of dynamic modifications we also have to define a-priori when, i.e. in which context and in which state of execution, certain modifications are allowed in order to ensure the dynamic and semantical consistency of a process.

Thus, the definition of the behavior of workflow execution as well as workflow evolution are the basis for supporting flexible workflows. In this paper, we focus on the workflow behavior definition and adaptation in both cases in the presence of a distributed workflow enactment approach which forms the basis for a scalable workflow management system. First, we sketch our workflow modeling language and show how the behavior of a task can be defined and adapted in different contexts. In particular, user-defined control flow dependencies, which allow to define and reuse complex behavior patterns, can be specified and applied in different contexts. Next, we outline how these concepts can be used to define a-priori less-restrictive workflows, i.e. workflows where a certain degree of freedom is left open to the actor. Furthermore, we discuss the definition of an adequate behavior for dynamic modifications in different situations without restricting our self on a transactional view on processes. In particular, we show how the application of and reaction to dynamic changes can be adapted in our approach depending on the process context and the behavior of a task itself.

2. The Workflow Modeling Approach

2.1 Task and Process definition

The building block of our workflow modeling approach is a *task definition* (or task type) which consists of a *task interface,* that specifies 'what is to do', and potentially several *process definitions,* which specify how the task may be accomplished. The task interface is defined by attribute, parameter, and process constraint defini-

**This work has been supported by the German Ministry for Research and Technology (BMBF) , project MOKASSIN under grant number 01 IS 606 B*

tions (all neglected throughout this paper) and by a task behavior definition which specifies the external context-free behavior of a task (e.g., transactional or non-transactional). The context-dependent behavior of task is defined by its application within a process definition. A process definition[1] may be atomic consisting only of a task description or system invocation, or complex. A complex process is defined in an activity-oriented manner by a task graph. The decision is taken at run-time, which process definition of a task definition is used to perform a task (late binding). This late binding mechanism also allows to create a new process definition at run-time (late modeling).

A *task graph* consists of task components*,* connectors, start and end nodes, and data inlets and outlets, which are linked by control and data flow dependencies: A *task component* is an applied occurrence of a task definition within a process definition. For every task component a split and join type (AND / OR) can be specified. Furthermore, connector components are predefined which just realize splits and joins.

Task components (and start and end node) are linked by *control flow dependencies*. Iterations within this task graph are modeled by a special predefined feedback relationship. A condition can be associated to every dependency to support conditional branches. We allow to define different control flow dependency types which can be applied and reused within several process definitions. The semantics of a control flow dependency type is defined by ECA rules as shown in the next section and illustrated in figure 2.

Similar to the definition of control flow dependencies we support the definition of *group relationship types*. A group relationship is used within a process definition in order to group arbitrary task components of a task graph; it applies the behavior defined by the group relationship to its components. A task component can be part of several not necessarily nested groups. A special kind of a group is a block. Blocks are nested and contain a subtask-graph with exactly one start and end component (particularly useful for exception handling).

Finally, task components can be linked by *dataflow relationships* according to the input and output parameters of their task definitions. Furthermore, a data inlet (or outlet) is used as a data source (or sink) in order to realize a vertical dataflow between the parameters of the task definition and their use within the workflow.

2.2 Distributed Workflow Enactment and its Execution Semantics

The execution semantics of tasks and of the task graph is defined by a statechart variant and event-condition-action (ECA) rules. Our enactment model is based on treating tasks as *reactive components* which encapsulates their internal behavior and interact with other tasks by message/event passing. This is a natural basis for a distributed enactment of workflows which was one of the design goals of our approach (beside of flexibility), and it is essential for scaling up to enterprise-wide workflow support.

A task has several built-in operations, which can be categorized into state transition operations, actor assignment operations, operations for handling of (versioned) inputs and outputs, and workflow change operations. For every operation, the task has the knowledge about when to trigger the operation, a condition that must hold for executing the transition and that acts as guard, and a list of receivers to which events are passed (to avoid communication overhead, we use no broadcast).

Before we introduce the behavior definition and adaptation on schema level, we briefly sketch syntax and semantics of our ECA rules (see [JoHe99a] for details; examples are given in figure 1 and 2): First of all, an ECA rule is always associated with an operation/transition, which defines the action part of the rule. Thus, ECA rules are structured according to the task's transitions. Additionally, an ECA rule consists of a list of event captures, a condition, and a receiver expression. Events define when an operation is to be triggered: when a task receives an event that matches an event capture in the event capture list, and when the task is in the source state of the corresponding transition, the event is consumed and the task tries to perform the transition. The invocation of a transition causes the evaluation of its condition. This transition condition acts as a guard, i.e., the transition is performed only when the condition holds (otherwise nothing is done). Thus, we follow a state-based semantics where a transition can be triggered by (internal and atomic) events or externally by user invocation. Invocation and applicability of a transition are strictly separated. The matching of an event with an event capture can be qualified to the causing task. For example, this allows a task to react differently on the event 'fin-

[1] Note, that we use process definition in a more restricted sense defining only how a task has to be done. To avoid misunderstandings, we use workflow and workflow specification as a general term (independent from our approach) in the usual more broader sense.

ished' depending on whether the event was received from a predecessor or from a sub-task. Furthermore, a trigger condition can be specified within an event capture which must hold for a valid event capture. Otherwise, the next event capture which matches the event is searched.

2.3 Definition and Adaptation of the Execution Behavior

The *context-free* behavior of a task is given by the behavior definition of a task definition by means of a statechart variant. It defines the states, their decomposition, and the operations/transitions which can be invoked in a certain state. Furthermore, exactly one context-free ECA rule can be defined for every transition. A task definition can *inherit* from an abstract task definition, i.e., a task definition with no process definitions, in order to define behavior classes of tasks (e.g., non-transactional, transactional; cf. [KrSh95, Wes98]). Within an inherited statechart, states can be refined and transitions can be added and redefined by changing their associated ECA rule. This allows to adapt the context-free behavior of tasks. Every task definition inherits from a *predefined task definition,* which consists of a statechart that defines the basic states, transitions, and context-free ECA rules as illustrated in figure 1.

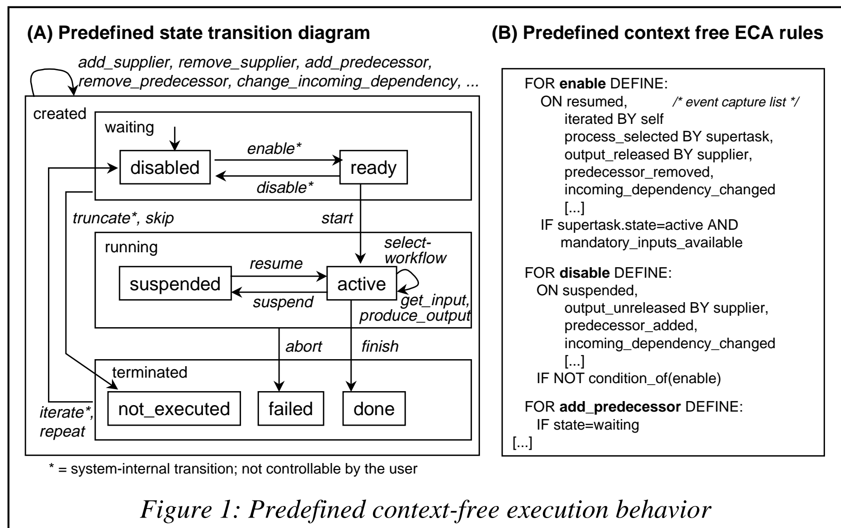


Figure 1: Predefined context-free execution behavior

The *context-dependent* behavior is given by the control flow dependencies and groups within a process definition. Rather than providing a limited set of different built-in control flow dependency types and group relationship types, arbitrary control flow types can be defined and adapted by a process engineer (cf. [JaBu96]). They are defined by a label, an informal description, and a set of ECA rules, which give the semantics of the dependency type. Within the task graph, the control flow dependencies or group relationships can be used by their labels abstracting from the detailed definition and reusing complex behavior patterns. Control flow dependency types are used to define ECA rules which establish intertask dependencies (e.g. end-start, start-start, deadline). Group relationship types are used to apply a behavior pattern to an arbitrary set of components.

As an *example*, we briefly explain the definition of the standard end-start dependency which consists of several rules partially shown in figure 2. We concentrate on the first rule which is defined for the enable transition and is applied to the target component of the dependency. The event capture defines that the enable transition is triggered whenever an event 'finished' has been received along the standard dependency under the condition that the corresponding dependency condition (specified by the placeholder 'dependency_condition') evaluates to true. The condition of the transition is defined by an reference to the source component requiring that it is an state done. The receiver expression is omitted in all examples.

In order to obtain the behavior of a task instance, the (partially) defined ECA rules of the context-free behavior are joined with the ECA rules that the task instance inherits from its context, i.e. from its dependencies and group relationships within a process definition. An example is shown in figure 2 for the 'FunctionalCheck' task which obtains the behavior of the incoming standard end-start dependency from 'Design', of the group type 'Exclusion', and of the context-free statechart. There are different modes for merging the context dependent behavior definitions. These modes are defined for every ECA rule (omitted in Fig. 2) and are applied to the transition condition: (a) conjunctive (default for group relationships) (b) disjunctive (c) using the join type of the component (default for dependencies) (d) using the inverted join type of the component (e) overriding. Dependency or group types using the overriding mode for the same transition cannot be applied simultaneously to a component. The overriding mode is normally used for adapting the application condition of change operations (see section 3.3).

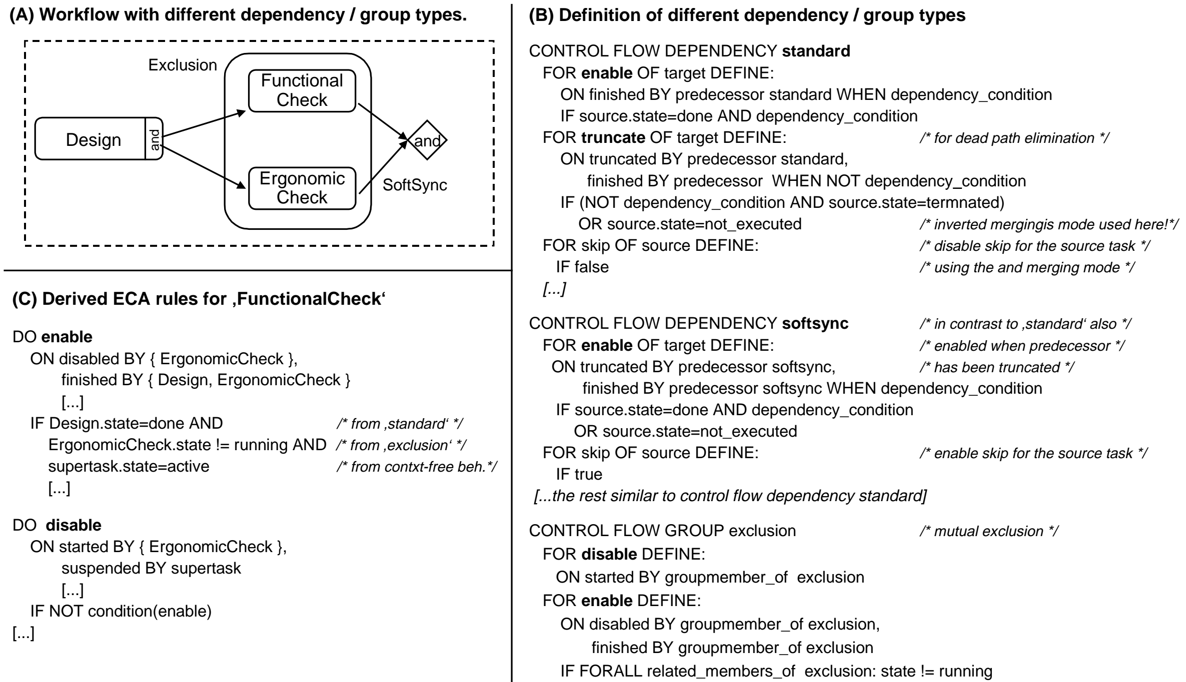


Figure 2: Different control flow types and their application for the definition of flexible workflows

3. Behavior Definition and Adaptation for Flexible Processes

In this section, we show how the introduced concepts of behavior definition can be used to define a priori flexible workflows and to adjust the application condition of and the reaction to dynamic changes depending on the behavior of the task itself and the context.

3.1 Defining Less-restrictive Workflows

We give two examples of defining a-priori less-restrictive workflows, i.e. workflows where a certain degree of freedom is left open to the actor, using the introduced concepts of user-defined control flow types.

The first example is the definition of the group relationship type ‘exclusion’ which is shown in figure 2. It forces its members to execute mutually exclusive. In conjunction with parallel branches, we can define that certain tasks should be executed sequentially, but without defining the actual ordering of the tasks. So, the user can choose which task he or she wants to perform next. In our example, a QA engineer may decide whether he/she wants to check a specification first against the functional or against the ergonomic requirements.

The second example focuses on skipping a task. Since skipping a task is normally not desirable (except from the viewpoint of the actor) and probably cause serve problems when needed output data has not been produced, its application should be restricted. Furthermore, in our state-based semantic, skipping a task would result in a deadlock since the successor tasks would wait for termination of the skipped task. Both problems are solved by the control flow dependency type ‘softsync’ (cf. [ReDa98], which also show useful applications in the case of dynamic changes). It waits only for the termination of a task when the task still can be executed (see relaxed enable condition in figure 2). Furthermore, the dependency allows to control when skipping is enabled. The skip transition is disabled for tasks which have at least one outgoing standard dependency. Thus, the process modeler can define which tasks can be skipped in a certain workflow.

3.2 Supporting Collaborative Workflows

Collaborative workflows are supported in our approach by version and workspace control capabilities which are integrated with the workflow model on conceptual level. Consumed and produced versions are managed within a task-oriented workspace. Versions can be released for dedicated tasks which allows a versioned data

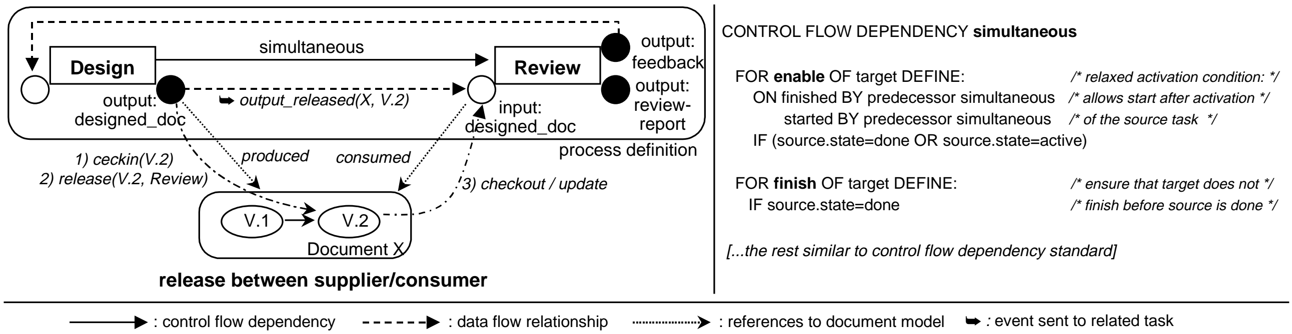


Figure 3: Data exchange between running tasks – controlled cooperation on workflow level

flow and the exchange of intermediate results between tasks. Furthermore, the data flow can be also used for control flow purposes. The availability of input data can be checked and the operations for releasing outputs generate events as any other transition so that tasks can react on these events. For example, the event 'output_released' triggers the evaluation of the enable transition of the consumers (see figure 1 and 3).

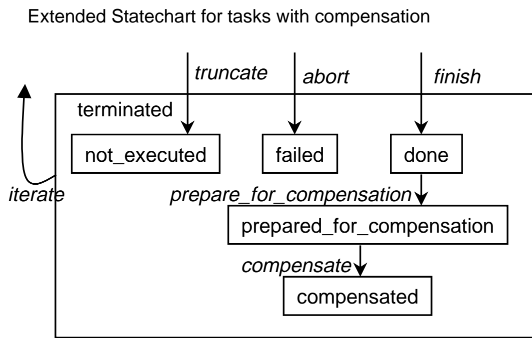
It is out of the scope of this paper to present details about the workspace capabilities (see [Joe98]). Rather, we like to show, how data exchange between simultaneously active instances can be controlled on the workflow level by means of the simultaneous dependency (cf. [HJKW96]). The definition of this dependency is illustrated in figure 3. It relaxes the activation condition so that the dependent task does not have to wait for the termination of the preceding task; a task is enabled when all mandatory inputs are available and the preceding task has been started. The main purpose of the simultaneous dependency is to ensure that the dependent task does not terminate before the preceding task. This termination synchronization guarantees that the latest results of the supplier is processed by the consumer. An example application is the design and review of a technical document where the designer may request for an early feedback from the reviewer (see figure 3).

3.3 Situation-dependent Handling of On-the-fly Changes

Our approach to dynamic changes of enacting workflow instances is based on applying ECA rules also to change operations. Every change primitive is encapsulated by a pre-condition which restricts its application, and by raising a corresponding event which is handled by the affected instances in order to ensure the behavioral consistency of the execution states. Thus, conceptually a change operation can be treated like a state transtion, and on-the-fly changes are supported in the presence of distributed workflow enactment since every task instance object has the knowledge about how to react on a change. The basic idea of this approach has been presented in our previous work DYNAMITE [HJKW96] on software process management, and all details of our approach to managing evolving workflow specifications can be found in [JoHe99b].

Whether a change is allowed and how to react on it highly depends on the particular situation and the behavior of the involved tasks. In this paper, we concentrate on the situation-dependent handling of dynamic changes. As an example, we outline useful application conditions and reactions for adding a new predecessor task (as an insertion between two sequential tasks or as a new parallel branch). In this case, the insertion of a new control flow dependency is important. It affects the target component which depends on a new predecessor task. Depending on the behavior of the affected task and on the context in which the task is applied, different application conditions and reactions can be useful for this change which all can be realized in our approach:

1) In general, this modification can be allowed for target components which have *not yet been started*. When the dependent task is already in the state ready, the event 'predecessor_added', which is raised by the change operation, results in triggering the disable transition. This transition is performed only if the enable condition no longer holds (see figure 1). Thus, re-evaluation of the enable condition ensures the behavioral consistency. However, adding a new predecessor task may be also useful and can be handled for target components which are already active or have been finished. In contrast to [EKR95], [CCPP96], [ReDa98], or [HoJa98], we do not restrict ourselves to situation-independent theoretical correctness criteria:
2) If the dependent task is already *active*, a meaningful reaction for example for an automatic batch process is to abort the task and to restart it later on (probably performing additional compensation activities). Fur-




```
CONTEXT-FREE ECA rules
  FOR prepare_for_compensation DEFINE:          /* changes to finished task triggers */
    ON predecessor_added, ...                   /* preparation for compensation */
  FOR compensate DEFINE:                        /* compensation starts when all */
    ON prepared_for_compensation BY self,       /* successors have been compensated */
    IF FORALL sucessors compensational: state = compensated

CONTROL FLOW DEPENDENCY compensational
  FOR disable OF target DEFINE:                 /* disable ready tasks when predecessor */
     ON prepared_for_compensation BY predecessor compensational   /* will be compensated */
  FOR abort OF target DEFINE:                   /* and abort running tasks in this case */
     ON prepared_for_compensation BY predecessor compensational
  FOR prepare_for_compensation OF target DEFINE: /* transitively prepare for compensation */
     ON prepared_for_compensation BY predecessor compensational,
  FOR compensate OF source DEFINE:              /* trigger compensation in reverse order */
     ON compensated BY successor compensational
```

Figure 4: Behavior definition for tasks with compensation

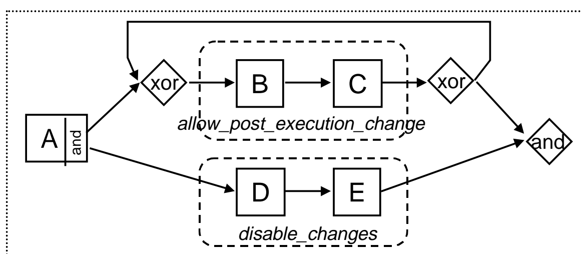
thermore, a manual task may just be suspended and can be resumed when the new preceding task is done. A human actor can easily work on the changed input values/documents.

Both behaviors can be realized by defining different behavior classes for batch tasks and manual tasks. In the former case, we add a trigger 'ON predecessor_added' to the context-free ECA rule of the abort transition, in the latter case, we add the same trigger to the suspend transition. Furthermore, we relax the transition condition of add_predecessor to 'IF state=waiting OR state=running'.

3) If the target task has been already *finished*, we may compensate all succeeding task of the new task (if possible). In figure 4, the behavior of tasks which provide compensation facilities is shown. In our example, compensation is done with a two phase protocol which takes the ordering of the tasks' execution into account (cf. [Ley95, KaRa98]): First, all tasks which are compensational dependent are prepared for compensation. Next, these tasks are compensated in the opposite ordering of their former execution. This complex behavior is realized by the compensation dependency. A compensation which is not order-dependent can be realized easily by a group type which just define an ECA rule that triggers the compensation when a certain event occurs.

   However, assume that the affected tasks are part of an iteration loop and that we are interested only in the future execution; then, we can insert the new task without any impact on the current pass of the loop (in the sense "now it's too late, but next time we should perform the additional task"). The new task becomes relevant only for the next iteration. This is realized by a group relationship type 'allow_post_execution_change' (shown in figure 5) which is used in a process definition to mark those regions where the above mentioned situation should be supported (obviously, this policy is not useful in general). Note, that the interplay of context-free and context-dependent behavior definition results in the appropriate behavior for all mentioned situation. E.g., for a manual task changes may be allowed in any state whereas for a automatic task changes may be disallowed when the task is active (if rollback/compensation is not possible or desirable).

4) Finally, for some parts of a process a process engineer may want to disallow dynamic modifications. In this case, a group relationship type 'disallow_change', which disables all change operations using the overriding mode, can be used to mark the relevant parts of the process (cf. figure 5).




```
CONTROL FLOW GROUP disable_changes
  FOR add_predecessor DEFINE:   /* same definition for all */
    override: IF false          /* change operations */
  [...]

CONTROL FLOW GROUP allow_post_execution_change
  FOR add_predecessor DEFINE
    IF state=terminated
```

Figure 5:Situation-dependent adaptation of the behavior of dynamic changes

4. Conclusion

The definition and situation-dependent adaptation of the tasks execution behavior is the basis for both, providing a priori flexible workflows and supporting dynamic changes in every possible situation. For the latter case, it is essential to note that there exist several practical cases where theoretical correctness properties, in particular the compliant property (cf. [CCPP96]), are too restrictive. We have shown, that the definition of control flow dependency and group relationship types on the basis of ECA rules is a powerful concept for behavior adaptation and dynamic changes in the present of distributed workflow enactment. Finally, human actors can react very flexible on changes of the context of a task. The integration of version and workspace control capabilities substantially supports adequate reactions to workflow changes in the case of manual and cooperative tasks.

The workflows which can be defined in this approach are by far more complex than in the case of workflows which consists only of conditional and parallel branches. Therefore, the analysis of correctness properties (e.g. deadlock-freeness) of the resulting task behaviors and their interaction are is a hard problem on which we currently work. The introduced concepts have been implemented in the project MOKASSIN which is funded by the German Ministry for Research and Technology (BMBF). Experiences as well as the architecture of our system which is realized as an distributed object system based on CORBA will be discussed in subsequent papers.

References

[CCPP96] Casati F., Ceri, S., Pernici B., Pozzi, G.: „Workflow Evolution“. In *Proc. of 15th Int. Conf. on Conceptual Modeling (ER’96)*, Cottbus, Germany, 1996; pp. 438-455.

[ElNu96] Ellis C.A. and Nutt, G.J.: "Workflow: The Process Spectrum", in *NSF Workshop on Workflow and Process Automation in Information Systems*, Athens, Georgia, 1996.

[EKR95] Ellis C.A., Keddara K. and Rozenberg, G.: "Dynamic Change Within Workflow Systems", in *Proc. of the Int. Conf. on Organizational Computing Systems COOCS’95*, Milpitas, CA, 1995; pp. 10-21.

[HJKW96] Heimann, P.; Joeris, G.; Krapp, C.-A.; Westfechtel, B.: "DYNAMITE: Dynamic Task Nets for Software Process Management", in *Proc. of the 18th Int. Conf. on Software Engineering,* Berlin, Germany, 1996; pp. 331-341.

[HoJa98] Horn S. and Jablonski S.: "An Approach to Dynamic Instance Adaption in Workflow Management Applications", in *Proc. of the CSCW-98 Workshop - Towards Adaptive Workflow Systems*, Seattle, WA, Nov. 1998

[JaBu96] Jablonski, St.; Bussler, Ch.: “Workflow Management - Modeling Concepts, Architecture and Implementation”, International Thomson Computer Press, London, 1996.

[Joe98] Joeris, G.: “Aspekte und Konzepte der Flexibilität in Workflow-Management-Systemen”, *in D-CSCW’98 Workshop on 'Flexibilität und Kooperation in Workflow-Management-Systemen‘*, Technical Report Angewandte Mathematik und Informatik 18/98-I, University of Münster, 1998; pp. 3-12.

[JoHe98] Joeris, G; Herzog, O.: "Managing Evolving Workflow Specifications", in *Proc. of the 3rd Int. IFCIS Conf. on Cooperative Information Systems (CoopIS’98)*, New York, Aug. 1998, IEEE Computer Society Press; pp. 310-319.

[JoHe99a] Joeris G.; Herzog O.: “Towards Flexible and High-Level Modeling and Enacting of Processes“, in *Proc. of the 11th Int. Conf. on Advanced Information Systems Engineering (CAiSE’99)*, LNCS 1626, Springer-Verlag, 1999; pp. 88-102.

[JoHe99b] Joeris G.; Herzog O.: “ Managing Evolving Workflow Specifications With Schema Versioning and Migration Rules“, TZI Technical Report 15-1999, Center for Computing Technologies (TZI), University of Bremen, 1999.

[KaRa98] Kamath, M.; Ramamrithan, K.: "Failure Handling and Coordinated Execution of Concurrent Workflows", in *Proc. of the 14th Intl. Conf. on Data Engineering (ICDE’98)*, Orlando, Florida, Feb. '98.

[KrSh95] Krishnakumar, N.; Sheth, A.: "Managing Hetereogeneous Multi-system Tasks to Support Enterprise-wide Operations", in *Distributed and Parallel Databases*, 3, 1995; pp. 1-33.

[Ley95] Leymann, F.: "Supporting Business Transactions Via Partial Backward Recovery in Workflow Management Systems, in Lausen, G. (Hrsg.): *Datenbanksysteme in Büro, Technik und Wissenschaft. GI-Fachtagung*, Springer Verlag, 1995; pp. 51-70.

[ReDa98] Reichert, M; Dadam, P.: “ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control”, *Journal of Intelligent Information Systems - Special Issue on Workflow Managament*, 10(2), 1998; pp. 93-129.

[Wes98] Weske, M: “State-based Modeling of Flexible Workflow Executions in Distributed Environments”, in Ozsu, T.; Dogac, A.; Ulusoy, O. (eds.) *Proc. of the 3rd Biennial World Conference on Integrated Design and Process Technology (IDPT’98), Volume 2 – Issues and Applications of Database Technology*, 1998; pp. 94-101.