Towards Flexible and High-Level Modeling and Enacting of Processes

Gregor Joeris and Otthein Herzog

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

Abstract. Process modeling and enacting concepts are at the center of workflow management. Support for heterogeneous processes, flexibility, reuse, and distribution are great challenges for the design of the next generation process modeling languages and their enactment mechanisms. Furthermore, flexible and collaborative processes depend also on unpredictable changes and hence require human intervention. Therefore, high-level process modeling constructs are needed which allow for an easy, adequate, and participatory design of workflows. We present a process modeling language which covers these requirements and is based on object-oriented modeling and enacting techniques. In particular, we outline how tasks and task nets are specified at a high level of abstraction, how flexible and user-adaptable control and data flow specifications are supported, and how reuse of workflow models can be improved. The approach is characterized by the uniform and integrated modeling of workflow schema and instance elements as objects and by the integration of flexible rule-based techniques with the high-level constructs of task graphs. Finally, we present our object-oriented approach for the distributed enactment of workflow models: A workflow is directly enacted by task agents which may be treated as reactive components, which interact by message passing, and whose execution behavior is derived from the context-free and context-dependent behavior of the tasks defined in the workflow schema.

1 Introduction

Workflow modeling and enacting concepts are at the center of workflow management. Support for heterogeneous processes (human-centered and system-centered), flexibility, reuse, and distribution are important challenges for the design of the next-generation process modeling languages and their enactment mechanisms (cf. [8, 25]). In particular, flexible and collaborative processes require human intervention. Therefore, a process modeling language on a high level of abstraction is needed which is easy to use and supports the visualization of its elements. In particular, the trade-off between high-level formalisms, such as graph-based modeling approaches, and flexible and executable low-level mechanisms, such as rule-based specifications, has to be resolved.

Based on these design goals and requirements, we have developed a process modeling language which supports flexible and user-adaptable control and data flow specifications, as well as dynamic modification of workflows. The approach aims at

M. Jarke, A. Oberweis (Eds.): CAiSE'99, LNCS 1626, pp. 88-102, 1999.

combining the flexibility of rule-based techniques with the high-level constructs of task graphs. Our approach is characterized by the uniform and integrated representation of workflow schema and instance elements based on object-oriented technology. Further characteristics are the separate definition of 'what to do' and 'how to do' in the workflow schema, the versioning of the workflow schema, and the separate definition of context-free and context-dependent behavior of tasks within a workflow. All these concepts improve the reusability of workflow models.

In this paper, we concentrate on the workflow modeling and enacting concepts for heterogeneous processes under the given design goals. On a conceptual level, we focus on the specification of complex user-defined control flow dependencies which can be used at a high level of abstraction, which are reusable in different contexts, and which allow for the definition of an adequate and flexible execution behavior in advance. On a technological level, we show how the modeled processes can be enacted by interacting distributed task agents which behave according to the process specification. Instead of interpreting and scheduling processes by a central and monolithic workflow engine, our approach leads to a flexible and distributed architecture.

In section 2, we identify requirements for an advanced and comprehensive approach to workflow management. Section 3 gives an overview on our process modeling approach, and section 4 introduces the basic enactment concepts. Section 5 explains the interplay between modeling and enacting and shows how the execution behavior can be adapted and reused on schema level. Finally, section 6 discusses related work, and section 7 gives a short conclusion.

2 Requirements for Advanced Workflow Management

2.1 Design Goals

In order to apply workflow technology to a broader range of processes and in particular to support dynamic, human-oriented, and distributed processes, we emphasize the following requirements for a comprehensive and integrated process modeling and enacting approach:

Adaptability to heterogeneous processes: Most business processes are rather heterogeneous entities consisting of well-structured and less-structured parts and encompassing transactional and non-transactional. All of them have to be integrated within a single process model. Thus, for adequate process modeling support, different modeling paradigms (net-based and rule-based, proactive and reactive control flow specification, etc.) have to be taken into account and have to be integrated (cf. [20, 25]).

Flexibility: Flexibility of a WFMS comprises two fundamental aspects: (1) The specification of a *flexible 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 routing and cooperative work at the workflow level (cf. [8, 15]). (2) The *evolution of workflow models* in order to flexibly modify workflow specifications on the schema and instance level due to process (re)engineering activities and dynamically changing situations of a real process (cf. [4]). Workflow evolution management is an important

part of our approach, but it is out of the scope of this paper (see [17] for a detailed overview).

Distribution: Distributed workflow enactment is a key requirement for a scalable and fault-tolerant WFMS (cf. [11]). But distribution and dynamic modifications of workflows cause contradicting architectural requirements. This trade-off has to be resolved for a distributed and flexible WFMS.

Reusability: Increasing business value and complexity of process models require the support of reuse of process specifications (cf. [9, 23]). Thus, a process modeling language should provide abstraction and encapsulation mechanisms.

Ease of use: A very crucial design goal is, that the language is *easy to use*, i.e., it should allow for modeling of processes at a *high level of abstraction,* and it should support the visualization of its elements (cf. [25, 6]) (this is naturally given in a net-based approach; on the other hand, rule-based specifications are hard to understand for people, but provide a great flexibility).

2.2 Advanced Control Flow Modeling

Based on the above presented requirements and design goals we refine the requirements for control flow modeling in order to support heterogeneous and flexible processes. For the representation of heterogeneous processes several *fine-grained inter-task dependencies* have to be supported, which define control flow dependencies between particular states of a task (cf. [1, 21]). Several dependencies of these are well-known from transaction management, e.g., the two-phase commit protocol shows different state-dependencies and also the need for reactive triggering mechanisms in order to commit or abort the corresponding activities. There are many further examples where advanced control flow modeling constructs are useful particularly in order to define an accurate and a priori less prescriptive flow of work, a few of which we are listing in the following (cf. [14, 24]): (1) Our first example is the *SEQ-operator* which forces a set of activities to execute sequentially, but in any order/permutation. In this case, the actor may determine ad hoc the ordering resulting in an a priori more flexible workflow. (2) The so-called *soft synchronization* is useful for dependencies between different branches. Here, an activity may start when the preceding activity is finished or will definitely not be executed. (3) The *deadline* operator says that an activity A can be started only if activity B has not been started. (4) *Simultaneous engineering* is another good example which also shows the need for advanced data flow modeling capabilities (cf. [13, 15]): in this case, dependent activities can overlap and an activity may pass intermediate results to subsequent activities.

In contrast to these requirements, several process modeling languages are based on Petri Net-like or equivalent semantics: an activity is represented as one monolithic block, which consumes all input information once it is started, and produces all outputs, when it has finished. In particular, since control flow dependencies cannot be specified independently for the start and end point of an activity, only end-start dependencies can be defined. Among parallel and conditional splits with corresponding join operators sometimes some special control flow operators are provided. Such approaches support modeling of workflows on a high level of abstraction and the graphical visualization of both workflow schemata and running workflow instances. However, there is no possibility to model fine-grained inter-task dependencies and to define user-adaptable control flow constructs.

On the other hand, the above listed advanced control flow dependencies can be expressed by more low-level, rule-based formalisms, e.g., by event-condition-action (ECA) rules. Unfortunately, whereas the representation of a workflow model as a set of rules is sufficient for enactment, it is inadequate for workflow design and hardly understandable for people (cf. [2]). Furthermore, the reusability of complex control flow dependency patterns is mostly weak since there is no encapsulation or abstraction mechanism (an exception are the rule patterns of [19]). Therefore, several approaches (e.g., [18, 3, 7, 29, 6]) provide high-level workflow modeling constructs which are transformed into global ECA rules for enactment – in this way the flexibility and expressive power of ECA rules on the modeling level is lost.

To summarize, among the general and well-known requirements of separated control and data flow modeling and provision of multi-paradigmatic control flow mechanisms, a flexible process modeling language should support the specification of fine-grained control flow dependencies, but still provide mechanisms to abstract from these detailed control flow specifications. In particular, an advanced workflow designer should be able to define new control flow dependencies (cf. [14]).

3 Overview of the Processes Meta Model

Our approach to process modeling and enacting is based on object-oriented modeling techniques (not to confuse with an OO process modeling method, where processes would be defined in a product- or object-centered way). All relevant entities are modeled as attributed, encapsulated, and interacting objects. Following the principle of separation of concerns, we divide the overall model into sub-models for tasks and workflows, documents and their versions, and organizational aspects in order to capture the different aspects of processes. We concentrate in the following on the modeling and enacting of tasks and workflows and disregard the integration of document, version, and workspace management capabilities in our approach (see [15, 16] for details). Furthermore, all elements of the organizational sub-model are omitted in this paper. We introduce our object model step by step starting with the definition of task types:

3.1 Task Definition and Task Interface

First of all, a *task definition* (or task type) is separated into the definition of the *task interface* which specifies 'what is to do', and potentially several *workflow definitions* (task body), which specify how the task may be accomplished (how to do) (see figure 1 and 2). Thus, the building block is the class `TaskDefinition` which may contain several `WorkflowDefinitions`. The decision is taken at run-time, which workflow definition of a task definition is used to perform a task (late binding). Every workflow definition has a condition which acts as a guard and restricts the allowed workflows according to the current case. Note, that when talking in the context of our formal meta model, we use workflow definition in the more restricted sense of defining only how a task has to be done (the task body) and use workflow synonymously to workflow definition when misunderstandings are excluded by the context.

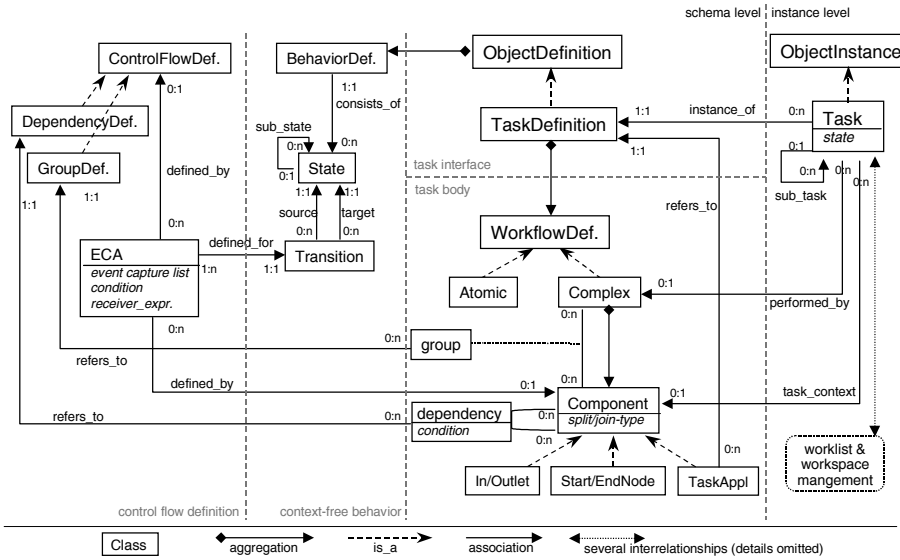


Fig. 1. Functional and behavioral aspects of the process meta model

The task interface of a task definition is defined by a set of parameter definitions, a behavior definition, and a set of business rules which constrain the valid workflow definitions (parameter definitions and business rules are omitted in figure 1). The behavior definition specifies the external context-free behavior of a task by a state-chart variant (e.g., transactional or non-transactional), whereas the context-dependent behavior is defined by control flow dependencies within a workflow definition (see below). A parameter definition specifies the kind of the parameter (in, out, inout), the type of the parameter, an access type, and a condition, which defines the properties the actual inputs and outputs have to comply with. The access type is based on our integrated workspace management services (see [15] for details).

3.2 Workflow Definition

The basic elements of modeling the control and data flow on a high level of abstraction are well-known from several process modeling languages. Usually, a workflow is modeled in terms of tasks or activities, specific control and data flow dependencies, and branching and synchronization operators. The execution semantics of these workflow-specific terms is mostly defined in workflow-independent terms like Petri Nets, statecharts, or ECA rules. We will follow such an approach, which is based on the high-level constructs of task graphs briefly introduced in this sub-section. In the next sections, we then show how the execution semantics of the control flow dependencies are defined and how user-defined and complex control flow constructs are supported.

A workflow definition may be atomic or complex. An atomic workflow consists only of a process description. A complex workflow is defined in a process-oriented

manner by a *task graph* which consists of task components, start and end nodes, and data inlets and outlets, which are linked by control and data flow dependencies:

Components: A *task component* is an applied occurrence of a task definition representing the invocation hierarchy. If a task definition is applicable only in a certain context, it can be locally declared within another task definition, restricting their visibility to this task type. Thus, the declaration and invocation hierarchy of task definitions are separated (as it is well-known from programming languages). For every task component a split and join type can be specified. AND- and OR-splits realize total and conditional branching, respectively. The corresponding join types synchronize the activated branches. In order to provide connectors independently of a task component, *connector components* are predefined which just realize splits and joins. Furthermore, a task graph consists of a start and end node which provide only syntactic sugar.

Control flow dependencies: 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 (by default, this condition is set to true). We allow to define different control flow dependency types which can be applied and reused within several workflow definitions. The semantics of a control flow dependency type is defined by ECA rules as introduced in the next section. Figure 4 illustrates an example with an end-start dependency and a deadline dependency.

Groups and blocks: Similar to the definition of control flow dependencies we support the definition of groups and block relationship types. A group relationship is used within a workflow definition in order to group arbitrary task components of a task graph; it applies the behavior defined by the group relationship to its components (e.g., to realize mutual exclusion). A block is a group, which contains a subtask-graph with exactly one start and end component (omitted in figure 1). Blocks can be nested. This mechanism is particularly useful for exception handling.

Dataflow relationships: 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 in a task graph 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.

4 Distributed Enactment by Reactive Components

4.1 Overview on Distributed Workflow Enactment by Reactive Components

This section sketches the basic *execution model* of our approach which is the basis for the detailed consideration of the definition of the execution behavior of a task in the next section. We follow the idea of treating tasks as reactive components (cf. [13, 5, 27, 26]): instead of interpreting a workflow instance by a (centralized) workflow engine, a workflow is directly enacted by distributed task instance agents which interact by event passing.

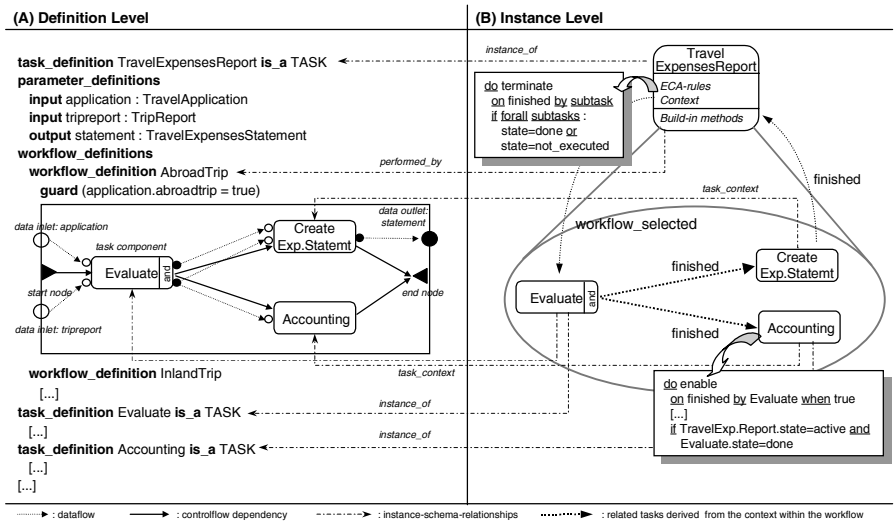


Fig. 2. Example of process definition and enactment

A task has several built-in operations/transitions, 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 transition, a condition that must hold for executing the transition, and a list of receivers to which events are passed. Furthermore, the task knows its related tasks (the context of the task, i.e., predecessors, successors, sub-tasks, super-task, supplier of inputs, and consumer of outputs). Thus, workflow enactment is distributed to task agents, which correspond to the components of a workflow definition and behave as defined in the schema. We denote this approach as light-weight agent-oriented, since the knowledge of how to react on events is explicitly represented and decoupled from the built-in operations (corresponding to [10]). On the other hand, task agents lack properties which characterize fully-fledged agents as defined by [28].

Example: A typical workflow enactment with traditional control flows is as follows (see figure 2B): A task is started and in most cases a workflow is selected automatically creating sub-tasks in the case of a complex workflow. An event is passed to the first tasks of this workflow, triggering the evaluation of the 'enable' transition. When a task is enabled, the role resolution is activated if no actor has yet been assigned explicitly. In the case of automatic tasks, the start transition will be directly triggered by the enable event (by the trigger ‘on enabled by self’). When a task is terminated, a corresponding event is sent to all successor tasks. This again results in the evaluation of the 'enable' transition. Furthermore, for all tasks which are connected to the end node, the finish event is also sent to the super-task, triggering the termination of the super-task, when all sub-tasks have been terminated.

4.2 Representation of Task Instances

For *instantiation*, neither a copy of a task definition is created (e.g., as it is done in the Petri Net-based approaches) and enriched by execution-relevant information (e.g., assignment of start tokens) nor a compilation into another formalism is used (e.g., as in IBM FlowMark [22], METEOR$_2$ [5]). We rather follow a tightly integrated approach, where a task instance is related to the relevant schema elements and where these interrelationships are explicitly maintained (as illustrated in figure 1 and 2). First of all, a task instance is related to its task definition. Next, when a workflow was chosen for execution at run-time, a 'performed_by' relationship is inserted, the subtasks are created according to the chosen workflow definition, and for every subtask the corresponding component within the workflow definition is identified. Thus, the dynamic task hierarchy is created step-by-step and all execution-relevant information of a workflow schema can be accessed by the instances. Only the execution state of a task instance, the dynamic invocation hierarchy, and the actual dataflow are persistently covered at instance level. The execution behavior, i.e., the ECA rules and the context of a task is cached within the task object, but is not made persistent. The behavior is derived from the context-free behavior of the task definition and the context-dependent behavior of the component that the task plays within a workflow definition. The tight coupling of schema and instances is also reflected by our architecture, which does not distinguish build- and run-time environments. This is the basis to support dynamic workflow changes (see [17] for details).

4.3 Representation and Semantics of the Task Execution Behavior

The *execution behavior* of tasks is defined by a statechart variant by means of states, transitions, and event handling rules. Event-condition-action (ECA) rules determine when an operation/transition is invoked, and when it is applicable. We adopt the concept of ECA rules as follows:

Syntax of ECA rules: First of all, an ECA rule is always associated with an operation/transition, which defines the action part of the rule. Furthermore, there is exactly one ECA rule for every transition. Thus, ECA rules are structured according to the task's transitions, and therefore the transition name is listed at the top of a rule (see BNF of an ECA rule below). Additionally, an ECA rule consists of a list of event captures, a condition, and a receiver expression.

```
ECArule        ::=  "DO" <transition>
                    "ON" <event_capture> { "," <event_capture> }
                    "IF" <transition_condition>
                    "SEND_TO" receiver_expr
event_capture  ::=  <event_name> ["BY" <event_producer_name>]
                                 ["WHEN" <trigger_condition>]
```

Semantics of event handling: 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 the transition condition defined by the ECA rule. This transition acts as a guard, i.e., the transition is performed only when the condition holds (otherwise nothing is done). We allow only the definition of atomic events, which are

used only for triggering the evaluation of the transition condition. These state-based semantics avoid the difficulties of defining complex event-based semantics. Moreover, user controllable operations can be invoked externally. In this case, the condition still ensures that the operation is applicable. Thus, invocation and applicability of a transition are strictly separated.

Event capture: The matching of an event with an event capture can be qualified to the causing task, e.g., this allows a task to react differently on the event finished, depending on whether the event was received from a predecessor or from a sub-task (triggering the enable or finish transition, respectively; see figure 2). 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.

Event generation and propagation: After an operation is executed, an event is automatically generated for that operation. In contrast to statecharts [12], we use events for inter-object communication and hence do not prescribe a broadcast of events to all tasks in order to avoid communication overhead (note, that a broadcast would not change the execution semantics; therefore and for the sake of readability, we omit the receiver expressions in all examples). The receivers of an event are rather defined by the receiver expression of an ECA rule, taking into account the workflow structure, i.e., passing events horizontally to related tasks as well as vertically among super- and sub-tasks.

5 Behavior Definition and Adaptation

So far, we have introduced how workflows are modeled in terms of a task graph and how tasks are enacted on the basis of event handling mechanisms. But we have not yet presented the interplay of the high-level modeling constructs with the flexible rule-based enacting techniques. In this section, we therefore show how the execution behavior of tasks can be defined and adapted on schema level, how user-definable control flow dependency types and fine-grained state dependencies are represented, and how the execution behavior of a task is configured from this specifications.

5.1 Definition of the Context-Free Behavior of a Task

The context-free behavior of a task is defined by a *statechart variant,* which is encapsulated by the class `BehaviorDefinition`. The statechart defines the states and the operations/transitions, which can be invoked in that state. We allow for the composition of states into complex states (OR-states), but we disallow concurrent states (AND-states). Furthermore, one context-free ECA rule can be defined for every transition. Further ECA rules can be added by the definition of control flow dependencies types and group relationship types, which we will introduce below.

A task definition can *inherit* from an abstract task definition, i.e., a task definition which has neither parameter definitions nor workflow definitions. Thus, the is_a hierarchy is used to define the behavior classes of tasks (e.g., non-transactional, transactional, etc.; cf. [21, 27] for detailed examples). Within an inherited statechart, new states can be added and atomic states can be refined. Also, transitions can be added and redefined by redefining the source state, refining the target state, and by redefining and adding ECA rules. Every task definition inherits from a *predefined task defi-*

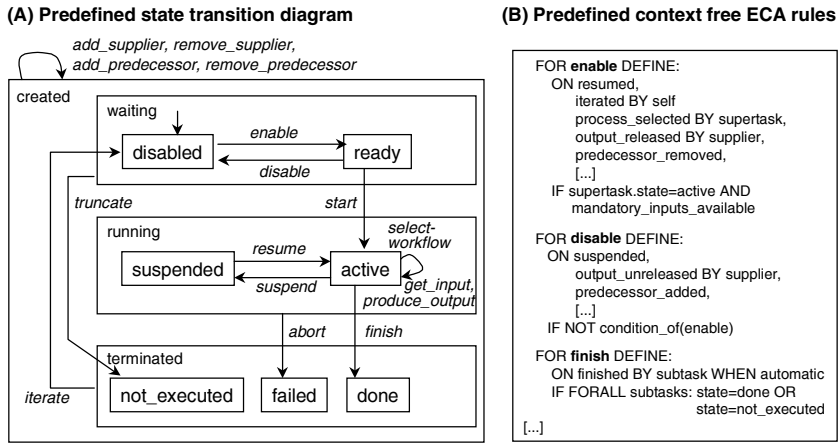


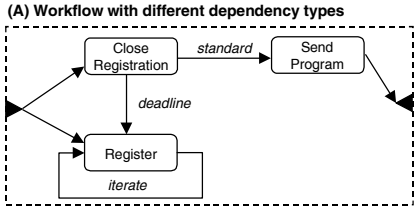
Fig. 3. Predefined context-free execution behavior

nition, which consists of a statechart that defines the basic states, transitions, and context-free ECA rules as illustrated in figure 3. In particular, the activation and termination condition of a task is associated with the enable and finish transition, respectively. The ready state is needed for worklist handling and the truncated state is used to handle the synchronization of dynamically determined parallel branches.

5.2 Definition of Control Flow Types

Different control flow dependency types and group relationship types can be defined by a process engineer. 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 relationship can be used by their labels abstracting from the detailed definition and reusing complex control flow schemes. Thus, the ECA rules defined by a control flow type define how to react on events dependent on the context. This leads to a combined approach which integrates the flexibility of rule-based techniques with the high-level constructs of task graphs.

As a first example and omitting the details of defining ECA rules on the schema level, we briefly explain the definition of the standard end-start dependency, which consists of several rules shown in figure 4. We concentrate on the first rule, which defines that the enable transition is triggered 'on finished by predecessor', and that the condition 'source.state=done' must hold. The application of this dependency type in the example of figure 4 results in the following behavior: when 'CloseRegistration' terminates, the corresponding event is sent to the successor tasks (here 'SendProgram'), triggering the evaluation of the enable condition (which requires 'CloseRegistration.state=done', but also that all mandatory inputs are available and that the supertask is active as defined by the context-free behavior), and probably performing the enable transition. Thus, the rules are associated with the target component of a dependency and are merged with other defined rules as introduced below. In a similar way, a soft synchronization dependency can be realized with the relaxed enabling condition 'source.state=done OR source.state=not_executed'.



(B) Definition of control flow dependency types

```
CONTROL FLOW DEPENDENCY standard
  FOR enable OF target DEFINE:
    ON finished BY predecessor standard WHEN dependency_condition
    IF source.state=done AND dependency_condition
  FOR truncate OF target DEFINE:          /* for dead path elimination */
    ON truncated BY predecessor standard,
        finished BY predecessor  WHEN NOT dependency_condition
    IF NOT dependency_condition OR source.state=not_executed
FOR iterate OF target DEFINE:             /* for re-activation of a path */
    ON iterated BY predecessor standard
        finished BY predecessor standard WHEN dependency_condition

CONTROL FLOW DEPENDENCY iterate     /* predefined feedback dep. */
  FOR iterate OF target DEFINE:
    ON finished BY predecessor iterate WHEN dependency_condition

CONTROL FLOW DEPENDENCY deadline
  FOR truncate OF target DEFINE:          /* skip restricted activity if */
    ON started BY predecessor deadline    /* restricting activity starts */
    IF source.state<>waiting
  FOR enable OF target DEFINE:
    IF source.state=waiting
```

(C) Cutout of the derived ECA rules for task ‚Register‘:

```
DO enable
  ON workflow_selected BY supertask,
      iterated BY { Register },
      finished BY { }                     /* can be removed */
  IF CloseRegistration.state=waiting AND
     supertask.state=active AND mandatory_inputs_available
DO truncate
  ON started BY {CloseRegistration }
  IF CloseRegistration.state<>waiting
DO iterate
  ON iterated BY { },                     /* can be removed */
      finished BY { Register },
  IF true
```

Fig. 4. Examples of the definition and use of control flow dependency types

Furthermore, we extend our approach of modeling task graphs by supporting the association of ECA rules directly to a task component. By this mechanism, local adaptations as well as reactions to externally generated events of application systems can be defined within a workflow.

5.3 Definition of ECA Rules on Schema Level

As illustrated in the examples, ECA rules can be partially defined on the schema level by an event capture, a transition condition, or a receiver expression. In the case of dependency types, it is further defined whether the ECA rule is associated with the source or target component of the application of the dependency in a task graph. In addition to the keywords 'source' and 'target', relationships of the workflow structure can be used in the specification of an ECA rule (e.g., to the super- and subtasks, to predecessor and successor task (possibly qualified by a specific dependency type), to task related by the feedback relationship ('iterators'), to consumer and supplier of outputs, to all tasks of a complex workflow or a group). This is essential for reusability since it avoids context-dependent definitions (such as 'on X.done do ...'). Finally, the keyword 'dependency_condition' refers to the control flow dependency condition defined in a task graph, and the keyword 'condition_of' refers to the condition of another transition.

5.4 Configuration of the Execution Behavior of a Task

The (partially) defined ECA rules of the context-free and context-dependent behavior definitions are joined together defining the behavior of a task instance. An ECA rule is relevant for a task T, if

- the ECA rule is defined by the statechart of the task definition of T (context-free ECA rule), or

- T refers to a source (target) component of a control flow dependency of type C in a task graph, and the ECA rule is defined by C for the source (target) component, or
- T refers to a component in a task graph, which is part of a group relationship of type G, and the ECA rule is defined by G (thus, the ECA rules of a group relationship are associated with all members of the group), or
- T refers to a component of a task graph, for which the ECA rule is locally defined.

One ECA rule for every transition is combined from all relevant ECA rules by

- creating the union of the event capture lists of the ECA rules (two event captures are identical, if and only if their event_name, event_producer, and trigger_condition are identical),
- creating the union of the tasks represented by the receiver expressions,
- generating a transition condition as follows (where G is the set of ECA rules which are defined for the transition and which are derived from group relationships, C is correspondingly the set of ECA rules derived from control flow dependencies, $\Theta \in \{\wedge, \vee\}$ depending on the join type of the task):

$$contextfree.condition \wedge \bigwedge_{\forall g \in G} g.condition \wedge \bigoplus_{\forall c \in C} c.condition$$

Finally, the relative statements within the ECA rules are resolved according to the structure of the performed workflow. Figure 4 gives an example of this behavior configuration for the 'Register' task (illustrating only the most relevant ECA rules and event captures for this example) and shows an additional dependency type, the deadline dependency.

We finish this section by giving some more examples which show the usage of group relationships: task components that are part of a parallel branch can be related by a SEQ group which declares an enable condition 'forall members: state!=running', and hence guarantees mutual exclusion. Another example is a two-phase commit. First, we have to specify a behavior definition with a new state ‘prepared’ and redefined transitions for transactional tasks supporting a 2PC protocol. Next, we may group the components of a workflow which are 2PC dependent. For this, we define a 2PC group relationship which ensures by two rules that an abort will cause the abort of all members of the group, and that the final commit is enabled when all members are in the state prepared. Note, that once defined, the workflow modeler can use this complex behavior by grouping task components according to the defined 2PC grouping relationship.

6 Related Work

With respect to the adaptability to heterogeneous processes, flexibility, and modeling of workflows at a high-level of abstraction, APEL and JIL are worth mentioning. JIL [25] provides a rich set of control flow modeling concepts and it combines in particular proactive and reactive mechanisms. APEL [6] is based on object-oriented modeling concepts, provides a graphical process modeling language, and uses state

transition diagrams and event-trigger rules for the specification of flexible workflows. However, the specification of user-adaptable control flow dependencies, dynamic modifications of workflows, and distributed enactment of workflows are not addressed by these approaches.

User-defined control flow constructs have been discussed by [14] (MOBILE). The semantics of different control flow constructs are defined by Petri Nets, but without considering the definition of control flow dependencies in the context of a distributed workflow enactment. Furthermore, it is not possible to define fine-grained state-dependencies since the control flow constructs are specified as execution statements/predicates which are independently defined of the execution state of an activity (e.g., deadline(A, and(B,C)) where A, B, and C are elementary/complex workflows).

The idea of treating tasks as reactive components is influenced by our previous work on the DYNAMITE approach [13]. But significant revisions and extensions of the underlying concepts and their integration into a coherent object-oriented framework have been made in our current approach.

With respect to distributed enactment, METEOR$_2$ [5], WASA [27], and EvE [26] follow similar approaches. In contrast to METEOR$_2$, we do not compile the workflow schema into executable code, but we follow an integrated approach for the representation of workflow schemata and instances which supports dynamic modifications of workflow schemata. METEOR$_2$ provides no high-level modeling constructs but supports modeling of fine-grained state dependencies by constructs similar to ECA rules. The WASA model is based on a less complex workflow modeling approach, so that only end-start dependencies have to be synchronized. Both approaches do not support late binding of workflows. Finally, EvE provides a framework for distributed workflow enactment based on distributed event-handling by reactive components, but it is not intended for workflow modeling on a high-level of abstraction. Furthermore, in contrast to our state-based semantics, EvE follows a purely event-based approach.

Finally, ECA rules are used by several WFMS for workflow execution. In those approaches, a workflow specification which is defined in a more high-level workflow modeling language, is transformed into a global set of ECA rules (e.g., WIDE [3], TriGSflow [18], Panta Rhei [7], Waterloo [29], APEL [6]). Thus, different concepts are used for modeling and enacting. The workflow engine is mostly realized using a centralized active database. In our approach, ECA rules can be used on the modeling level in a structured way in order to adapt control flow dependencies and to enhance the flexibility of the WFMS. The rules are encapsulated and hence can be applied at a high level of abstraction for workflow definition. Finally, in contrast to the centralized and transformation-based approaches, we follow a distributed and configuration-based approach, where the ECA rules are derived from the specification for every task instance object and hence define the inter-object communication of the distributed object system.

7 Conclusion

In this paper, we have proposed an object-oriented approach to modeling and enacting of heterogeneous processes that deals with the challenging requirements of flexibility,

reuse, distribution, and provision a process modeling language at a high level of abstraction. The encapsulation of a workflow definition by the task interface, the definition of different behavior classes, and the definition of user-adaptable control flow types characterize our modeling formalism and enhance reusability of process models. Furthermore, the combination of rule-based techniques with the high-level constructs of task graphs results in a great flexibility without losing the ability of high-level workflow modeling. The tight integration of schema and instance elements and schema versioning concepts are the basis for supporting dynamic workflow changes. Finally, based on the introduced modeling concepts, distributed enactment is realized in a natural way by distributed and interacting task objects. The presented concepts have been prototypically implemented using CORBA. The architecture of the system will be addressed in subsequent papers.

References

1. Attie, P.C.; Singh, M.P.; Sheth, A.; Rusinkiewicz, M.: "Specifying and Enforcing Intertask Dependencies", in *Proc. of the 19th Int. Conf. on Very Large Databases (VLDB'93)*, Dublin, Ireland, 1993.
2. Belkhatir, N.; Estublier, J.; Melo, W.L: "Adele 2: a Support to Large Software Development Process", in Dowson, M. (ed.): *Proc. of the 1st Int. Conf. on the Software Process*, IEEE Computer Society Press, 1991; pages 159-170.
3. Casati, F.; Ceri, S.; Pernici, B.; Pozzi, G.: "Deriving Active Rules for Workflow Enactment", in Wagner, R.R.; Thoma, C.H. (eds.) *Proc. of 7th Intl. Conf. on Database and Expert System Applications (DEXA'96)*, Zurich, Swiss, Sept. 1996, Springer, LNCS, pp. 94-115.
4. Conradi, R.; Fernström, C.; Fugetta, A.: "A Conceptual Framework for Evolving Software Processes", *ACM SIGSOFT Software Engineering Notes*, Vol. 18, No. 4, 1993; pp. 26-34.
5. Das, S.; Kochut, K.; Miller, J.; Sheth, A.; Worah, D.: "ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for METEOR_2", Technical Report UGA-CS-TR-97-001, Department of Computer Science, University of Georgia, Feb. 1997.
6. Dami, S.; Estublier, J.; Amiour, M.; "APEL: a Graphical Yet Executable Formalism for Process Modeling", in Di Nitto, E.; Fuggetta, A. (eds.) *Process Technology, Special Issue of the International Journal on Automated Software Engineering*, 5(1), 1998; pp. 61-96.
7. Eder, J.; Groiss, H.: "A Workflow-Management-System based on Active Databases" (in german), Vosser, G.; Becker, J. (eds.) *Geschäftsprozeßmodellierung und Workflow-Management: Modelle, Methoden, Werkzeuge*, Int. Thomson Publishing, 1996.
8. Ellis, C.A.; Nutt, G.J.: "Workflow: The Process Spectrum", in *NSF Workshop on Workflow and Process Automation in Information Systems*, Athens, Georgia, 1996.
9. Estublier, J.; Dami, S.: "About Reuse in Multi-paradigm Process Modelling Approach", in *Proc. of the 10th Intl. Software Process Workshop (ISPW'96)*, Dijon, France, 1996.
10. Genesereth, M.R.; Ketchpel, S.P.: "Software Agents", in *Communications of the ACM*, 37(7), 1994; pp. 48-53.
11. Georgakopoulos, D.; Hornick, M.; Shet, A.: "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure". *Distributed and Parallel Databases*, 3(2), 1995; pp. 119-153.
12. Harel, D.; Gery, E.: "Executable Object Modeling with Statecharts", in *Proc. of the 18th Int. Conf. on Software Engineering,* Berlin, Germany, 1996; pp. 246-257.
13. 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.
14. Jablonski, St.; Bussler, Ch.: "Workflow Management - Modeling Concepts, Architecture and Implementation", International Thomson Computer Press, London, 1996.

15. Joeris, G.: “Cooperative and Integrated Workflow and Document Management for Engineering Applications”, in *Proc. of the 8th Int. Workshop on Database and Expert System Applications, Workshop on Workflow Management in Scientific and Engineering Applications*, Toulouse, France, 1997; pp. 68-73.
16. Joeris, G.: “Change Management Needs Integrated Process and Configuration Management”, in Jazayeri, M; Schauer, H (eds.), *Software Engineering - ESEC/FSE’97*, Proceedings, LNCS 1301, Springer, 1997; pp. 125-141.
17. 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; pp. 310-319.
18. Kappel, G.; Pröll, B.; Rausch-Schott, S.; Retschitzegger, W.: “TriGSflow – Active Object-Oriented Workflow Management”, in *Proc. of the 28th Hawaii Intl. Conf. On System Sciences (HICSS’95)*, Jan. 1995; pp. 727-736.
19. Kappel, G.; Rausch-Schott, S.; Retschitzegger, W.; Sakkinen, M.: "From Rules to Rule Patterns", in Constantopoulos, P.; Mylopolous, J.; Vassiliou, Y. (eds.) *Proc. of the 8th Intl. Conf. on Advanced Information System Engineering (CAiSe'96)*, Springer, LNCS 1080, 1996; pp. 99-115.
20. M.I. Kellner: "Multiple-Paradigm Approaches for Software Process Modeling", in Thomas, I. (eds.) *Proc. of the 7th Intl. Software Process Workshop - 'Communication and Coordination in the Software Process’*. Yountville, CA, USA, Okt. 1991, IEEE Computer Society Press; pp. 82-85.
21. Krishnakumar, N.; Sheth, A.: "Managing Hetereogeneous Multi-system Tasks to Support Enterprise-wide Operations", in *Distributed and Parallel Databases*, 3, 1995; pp. 1-33.
22. Leymann, F.; Altenhuber, W.: “Managing business processes as an information resource”, *IBM Systems Journal*, Vol. 33, No. 2, 1994; pp. 326-348.
23. Puustjärvi, J.; Tirri, H.; Veijalainen, J.: "Reusability and Modularity in Transactional Workflows", in *Information Systems*, 22(2/3), 1997; pp. 101-120.
24. 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), Kluwer Academic Publishers, March 1998; pp. 93-129.
25. Sutton Jr., S.M.; Osterweil, L.J.: "The Design of a Next-Generation Process Language", in Jazayeri, M; Schauer, H (eds.), *Software Engineering - ESEC/FSE’97*, Proceedings, LNCS 1301, Springer, 1997; pp. 142-158.
26. Tombros, D.; Geppert, A; Dittrich, K.R.: "Semantics of Reactive Components in Event-Driven Workflow Execution", in *Proc. of the 9th Intl. Conf. on Advanced Information System Engineering (CAiSe'97)*, Springer, LNCS 1250, 1997; pp. 409-420.
27. 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.
28. Wooldridge, M.; Jenings, N.: “Intelligent Agents: Theory and Practice”, in *Knowledge Engineering Review*, 10(2), 1995; pp. 115-152.
29. Zukunft, O.; Rump F.: "From Business Process Modeling to Workflow Management: An Integrated Approach", in Scholz-Reiter B.; Stickel E. (eds.) *Business Process Modeling*, Springer-Verlag, Berlin, 1996.