Decentralized and Flexible Workflow Enactment Based on Task Coordination Agents

Gregor Joeris

Intelligent Systems Department, Center for Computing Technologies
University of Bremen, PO Box 330 440, D-28334 Bremen, Germany
joeris@tzi.de

Abstract. *Flexibility and distribution are major challenges of an advanced workflow management system, but have been addressed mostly separately from each other. In this paper, we present an agent-based workflow enactment service which combines flexible and decentralized workflow execution. Every task is coordinated by its own task (coordination) agent which interacts with related task agents by event passing. Realized as reactive agents, the task agents know how to react to state changes as well as to structural workflow changes so that workflow changes can be handled also in a decentralized enactment architecture. Instead of generating different task or workflow objects for the different task types of a workflow schema, the execution behavior of a task agent is extracted from the workflow schema and explicitly represented within the task agents. Finally, on schema level, the behavior definition can be customized in order to express an adequate behavior for heterogeneous and flexible processes. This is not only the basis for defining less restrictive workflows in advance, but also for supporting dynamic workflow changes in every possible situation.*

1. Introduction

Flexibility and distribution are major challenges of an advanced workflow management system, but have been addressed mostly separated from each other. Support for flexible workflows in schema-based workflow management systems (WFMS) has to cope with two fundamental challenges (cf. [ElNu96, Joe99]):

(a) *A-priori flexibility* focuses 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 and to allow for a certain degree of freedom in workflow execution. Furthermore, a flexible workflow management approach should capture and support different kinds of processes consisting of well-structured and less-structured parts and encompassing human-oriented as well as system-oriented tasks.

(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, 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. This workflow evolution behavior depends on the involved task types and the particular situation, it cannot be defined globally and uniformly for any types of workflows.

Since a posteriori flexibility requires human intervention, it implies an additional and very crucial design goal: a workflow modeling language on a *high level of abstraction* is needed which is *easy to u*se and supports the visualization of its elements (cf. [SuOs97]). In particular, the trade-off between high-level formalisms, such as graph-based modeling approaches, and low-level mechanisms, such as rule-based specifications which are hard to understand for humans and difficult to analyze but provide a great flexibility, has to be resolved.

Beside flexibility, *distributed workflow enactment* is a key requirement for a scalable and fault-tolerant WFMS and hence the basis for enterprise-wide and inter-organizational workflow support (cf. [GHS95]). A central and monolithic workflow engine as suggested by the WfMC reference model is therefore not sufficient. But, support for a-priori flexible workflows as well as support for a-posteriori workflow changes make distributed workflow enactment more difficult – in particular in the case of a highly decentralized workflow enactment (cf. [Mil+96]) which we follow.

In this paper, we present an agent-based workflow enactment service which combines flexible and decentralized workflow execution. Every task is coordinated by its own task (coordination) agent which interact with related task agents by event passing. Realized as reactive agents, the task agents know how to react on state changes as well as on structural workflow changes so that workflow changes can be handled also in a decentralized enactment architecture. Instead of generating different task or workflow objects for the different task types of a workflow schema, the execution behavior of a task agent is extracted from the workflow schema, explicitly represented within the task agent and updated when the workflow schema is changed. In contrast to autonomous or intelligent agents, the task agents behave as defined in the workflow schema where the behavior of heterogeneous and flexible processes can be modeled on a high level of abstraction.

Section 2, characterizes different approaches of agent-based workflow management. In section 3, we introduce the underlying decentralized enactment model which is based on reactive task coordination agents. With these enactment concepts in mind, we outline in section 4 the workflow meta model and in particular the concepts of behavior definition and customization on schema level. Section 5 gives examples of the definition of flexible workflows within this approach. Finally, section 6 gives a short conclusion.

2. Agent-Based Workflow-Management

Agent technology can be used in different ways for managing workflows and realizing workflow management systems:[1]

1. ***Agents as cooperating actors (role-based; autonomous agents)***: In this approach, different agent types are built for different tasks and fulfill different roles (cf. [Kir96]). E.g., a specialized agent for risk analysis in the field of claim processing performs or assists in performing a specific task and interacts with other agents. Based on an agent architecture designed for managing business processes, agencies are designed for and adapted to different application areas where the designed agents perform a workflow autonomously. This approach reflects directly the organizational structure where a business process takes place. It is characterized by the role-based design of different

[1] A fourth approach which is proposed by Chang and Scott [ChSc97] does not directly apply agents to process management but uses agents to facilitate various parts of existing workflows. Different agents which act on behalf of a user are introduced as a front end to existing WFMS.

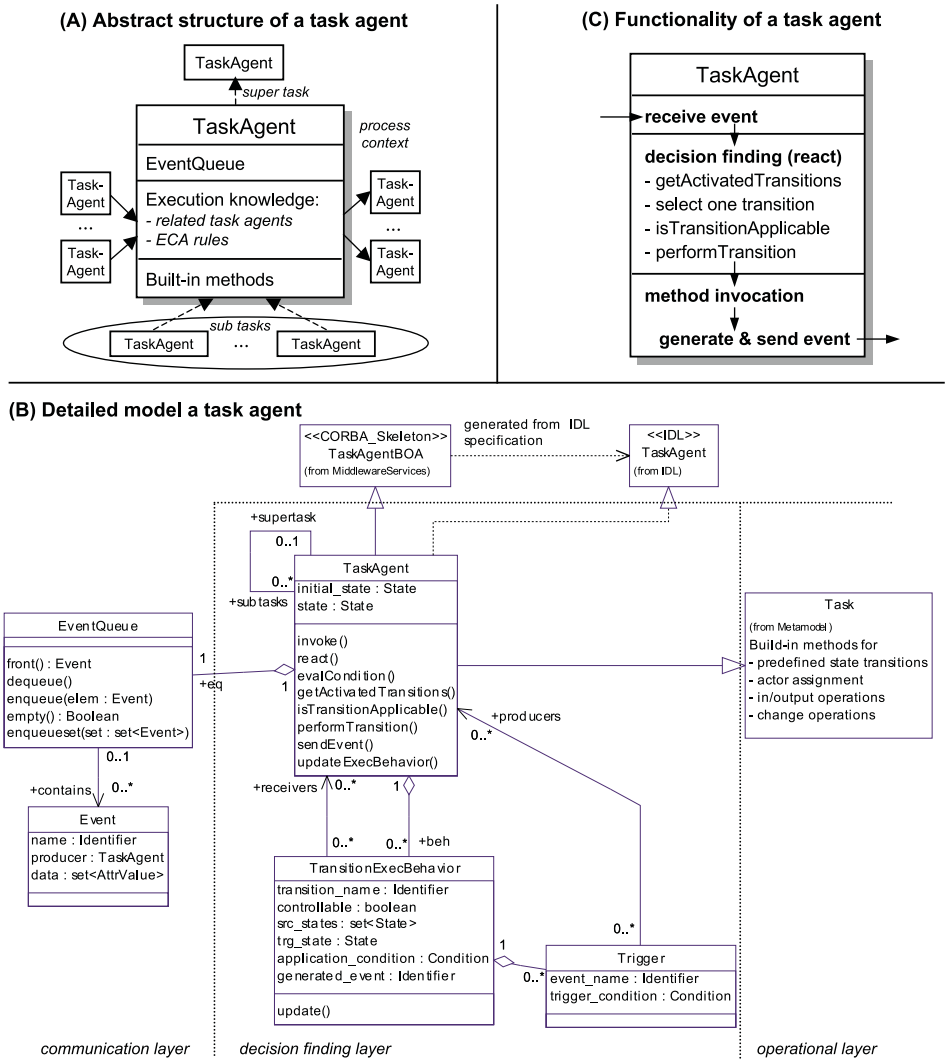


Figure 1: Structure and functionality of a reactive task agent

agent types, but does not follow an activity-oriented process modeling methodology. Rather, the workflow is encapsulated in domain-specific application knowledge and in the interaction/negotiation strategies used by the agents. ADEPT [Jen+96, Jen+00], PEACE+ [ALO96], [Sin97, SiHu98], and [YuSc99] follow this approach.

2. ***Agent technology as a key infrastructure technology for building flexible workflow engines (activity-based approach; reactive task coordination agents)***: This approach follows the classical *activity-oriented* process modeling and workflow management methodology. Activities, roles, and actors are separate aspects within a workflow

schema. The control and data flow is modeled explicitly between activities. The workflow schema is used to coordinate the execution of tasks. Agent technology is used in order to realize a flexible *and* decentralized enactment architecture. Reactive agents coordinate the execution of tasks as defined within the workflow schema without the need for a central workflow enactment service. Furthermore, they can support complex control flow patterns needed for coordinating flexible and heterogenous processes and handle dynamic workflow changes. We will follow this approach since it meets our design goals for a flexible and distributed but still activity-oriented and schema-based WFMS. We compare this approach to other decentralized enactment architectures in section 3.3.

3. ***Mobile agents realizing a migrating workflow***: Mobile agent technology is a good starting point for realizing the concept of migrating workflows [CiRu98]. A workflow instance is migrated to different "service stations" where tasks can be performed. Mobile agents can control the migration by selecting appropriate "service stations" and can control the execution of tasks and gather their results. In particular, this approach can be used in an inter-organizational setting (cf. [Mer+96]). But, the migrating workflow instances make it very difficult to support workflow schema changes and to coordinate flexible workflows which consist of complex control flow dependencies.

3. Decentralized Workflow Enactment by Reactive Task Agents

We start introducing our workflow modeling and enacting approach by describing the (low-level) execution model which follows the idea of treating tasks as reactive components (cf. [HJKW96, Das+97, TGD97]): instead of interpreting a workflow instance by a (centralized) workflow engine, a workflow is directly enacted by distributed task coordination agents (short: task agent) which interact by event passing. The execution behavior of every task agent is defined by a structured set of specific ECA rules. After introducing the underlying execution model in this section, we show in section 4 how workflows can be modeled on a high-level of abstraction on the schema level, how the context-free and context-dependent behavior can be customized, and how the execution behavior of the task agents can be derived from the workflow schema.

3.1 Overview on Decentralized Workflow Enactment by Reactive Task Agents

A *task agent* is a quite simple but powerful *reactive agent* which has a typical three-layered architecture consisting of a communication layer, a layer for decision finding, and a operational layer (figure 1a). The operational layer is represented by the class `Task` and contains all built-in operations/transitions[2] which can be categorized into state transitions, actor assignment operations, operations for handling of (versioned) inputs and outputs, and workflow change operations.

For every operation, the task agent has the knowledge about when to trigger the operation, a condition that must hold for executing the operation, and a list of receivers to which events are passed. Furthermore, the task agent knows its related tasks agents (i.e., predecessors, successors, subtasks, super-task, supplier of inputs, and consumer of outputs). Thus, the knowledge of how to react on events is explicitly represented in the decision- finding layer in terms of

[2] We use operation and transition synonymously throughout this paper (but with different emphasizes on arbitrary operations or state transitions, respectively)

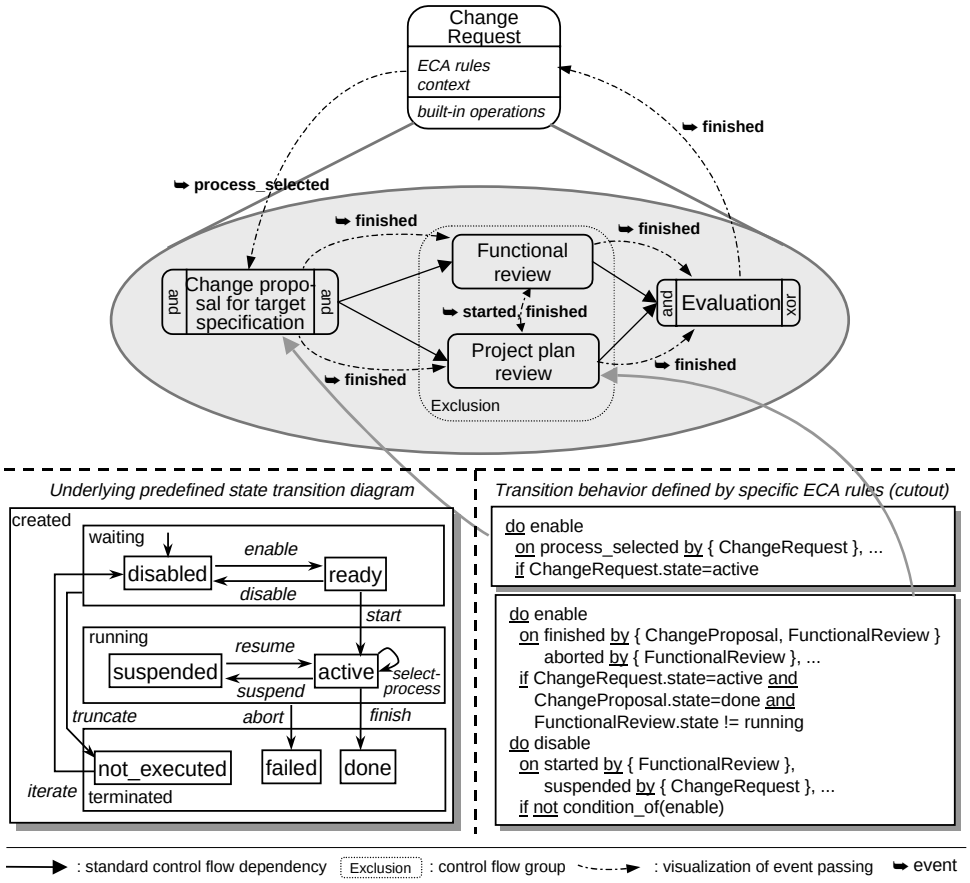


Figure 2: Example of decentralized workflow enactment by reactive task agents

specific event-condition-action (ECA) rules and decoupled from the built-in operations. This is the most basic characteristic of an agent architecture [GeKe94]. However, these agents are neither intelligent nor autonomous agents [WoJe95], since their execution behavior is derived from the workflow schema (see section 4) and they behave exactly as defined within the schema. Thus, the information system characteristic of the WFMS predominates where the agent-oriented execution model leads to a flexible and distributed architecture.

The communication between task agents is based on passing primitive events. An event consists of an event name (denoting the event type), a reference to the producer of the event, and a set of arbitrary attribute-value pairs (which we omit in the following). This simple communication language is sufficient for this application and also shows the differences to intelligent agents. Every task agent encompasses its own event queue (EventQueue). The event queue and the execution state (state) together form the execution state configuration of a task agent.

Example: Figure 2 shows a cutout of an change request process that is enacted by interacting

task agents. For the time being, we neglect the details of the partially illustrated ECA rules of the different task agents (see below) and first explain a typical workflow enactment. Furthermore, we assume a state transition diagram which defines the fundamental state and state transitions of a task agent (see upper right corner in figure 2). As explained in section 4, this diagram is part of the context-free behavior definition that every task type inherits.

After starting the super-task "ChangeRequest", choosing a task realization for execution – following a late binding concept (in this case a complex process definition) –, and creating sub-tasks, an event "process_selected" is passed to the first tasks of the process (here: "ChangeProposal"). This event normally triggers the enable transition of task agent, i.e. its applicability is evaluated and, on success, the transition fires. When a task is enabled for execution, 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. When a task is finished, a corresponding event is sent to all succeeding task agents. This again results in the evaluation of the enable transition of the corresponding task agents. Furthermore, for all end tasks, the finish event is also sent to the super-task, triggering the termination of the super-task, when all subtasks are state "done" or have been truncated. The truncate transition is used for dead path elimination.

The example gives also an impression of the realization of more complicated control flows. The control flow group "Exclusion" shown in the example forces its members to execute mutually exclusively. This behavior is expressed by a restricted application condition (e.g., enabling of the task "ProjectPlanReview" requires that "FunctionalReview" is not active and vice versa) and by additional triggers and event passing rules in order to react to the start-/finish-transition of the related tasks.

3.2 Representation and Semantics of the Execution Behavior of a Task Agent

The *execution behavior* of a task agent is defined by means of state transitions and the transition's behavior description (TransitionExecBehavior) which determine when an operation/transition is invoked and when it is applicable (see figure 1b). A transition is defined by its name, a "controllable" flag indicating whether the transition may be invoked by an external user or not (borrowed from [KrSh95]), a set of source states, a target state, and the event type that is generated by the transition. Furthermore, a transition consists of a behavior description whose underlying concept is in principle similar to an ECA rule or to a statechart transition label. Its textual notation which we use in the examples (see e.g. fig. 2) is defined by the following EBNF (see also corresponding elements within the meta model in figure 1b):

```
ECArule ::= "DO" transition "ON" trigger { "," trigger }
            "IF" application_condition
            "SEND_TO" receiver_expr
trigger ::= event_name ["BY" event_producer_set]
            ["WHEN" trigger_condition]
```

First of all, the transition or operation itself is the action part of the rule. Beside a state change, the built-in operations may cause additional actions like updating the work lists, consuming input data etc. However, a process engineer cannot customize these actions; rather, he/she can customize only the behavior that determines when an operation is invoked or applicable. For this, the transition's ECA rule consists of a set of triggers, an application condition, and a set of event receivers.

Transition triggering: The transition invocation is defined by a set of triggers. A trigger consists of a primitive event type, a set of event producers and a trigger condition. When an incoming

event matches an arbitrary trigger and when the task is in the source state of the corresponding transition, the transition is *activated* (see definition 1 using OCL expressions). An event *matches* a trigger when the event names are identical, the event producer is listed in the set of legal event producers of the trigger, and the trigger condition holds. Thus, the reaction on an event can be defined in dependency to the task agent which has generated the event. E.g., a task agent reacts 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). The same holds for the trigger condition which is used to select an appropriate transition rather than to define the applicability of a transition. The most important usage of a trigger condition is to react differently in the case of an OR-split. Depending on the split condition a task may be enabled or truncated when a preceding task has been finished. The truncation event is further propagated leading to a decentralized processing of dead path elimination.

***Definition 1**: Activated Transitions*

```
TaskAgent::getActivatedTransitions(event : Event) :
Set(Transition):
post:  if self.eq.empty() then result->isEmpty
       else result = self.beh->select(src_states->
            includes(self.state))->select( tr |
            tr.trigger->exists( tt |
            tt.event_name = event.name and
            tt.producers->includes(event.producer) and
            self.evalCondition( tt.trigger_condition) ))
       endif
TaskAgent::evalCondition(condition : Condition) :
boolean:
   -- evaluates an OCL-condition in the context of
   -- a task agent                                    ■
```

Semantics of transition firing: From all activated transitions of a task, one transition is chosen non-deterministically, the event is consumed, and the transition is invoked. The invocation of a transition first causes the evaluation of the transition's application condition. In contrast to the trigger condition, this condition acts as a guard, i.e., the transition is performed only when the condition holds (otherwise nothing is done; in particular, no other activated transition is performed). We allow only the definition of atomic events, which are used only for triggering the evaluation of the transition's application condition. These state-based semantics avoid the difficulties of defining complex event-based semantics (and differs from ECA rules or statecharts [HaGe96])[3]. 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.

When a triggered transition is applicable, it is performed (performTransition) and after executing the transition a corresponding event is generated and sent to all receivers (sendEvent). In contrast to statecharts, we use events for inter-agent communication and hence do not prescribe a broadcast of events to all tasks in order to avoid communication overhead. The receivers of an event are rather defined by the receiver set of an ECA rule (receivers). On schema level, the receiver set is defined in terms of a relative path expression over the workflow structure (e.g., horizontally to succeeding tasks as well as vertically to super- or subtasks). The expression is resolved to actual task agents on instance level.

The following definition gives the formal semantics of transition firing regarding the execution states of the task agents and the generated events. The shown definition neglects all

[3] In fact, event-triggercondition-applicationcondition-transition rule would better describe the approach but would lead to an illegible presentation.

additional changes which are performed by the built-in operations (e.g., update of the work lists, changes of the workflow schema in the case of change operations etc.). In particular, transitions may have an effect although they do not change the execution state (in this case, the target state is not specified). Finally, the new state of the event queue results from dequeuing the first event (which has no effect on an empty event queue) and enqueuing all generated events from all fired transitions where the task agent is part of the receiver set. All new events of one step are enqueued in an arbitrary order (enqueueSet).

***Definition 2**: Transition firing*

Let T be the set of task agents of a workflow case. Let $\varepsilon_k = t_k.eq.front()$ be the first event of a task $t_k \in T$ and $\delta_k \in t_k.getActivatedTransitions(\varepsilon_k)$ be a (non-deterministically chosen) activated transition for the task t_k according to the event ε_k. For $t_k.eq.empty()$ we set $\delta_k = \perp$.

The set *of the chosen and applicable transitions* is then defined as

$$\Delta = \{ \delta_k \mid \delta_k \neq \perp \wedge k \in \{1,..,|T^*|\} \wedge t_k.evalCondition(\delta_k.application\_condition) = \underline{true} \}$$

Then, the *new execution state* for the task agents t_k ($k \in \{1,..,|T|\}$) results from the *firing of the transitions* δ_k as follows:

$$t_k.state = \begin{cases} \delta_k.trg\_state & \text{if } \delta_k \in \Delta \wedge \delta_k.trg\_state \neq \text{""} \\ t_k.state & \text{otherwise} \end{cases}$$

$$t_k.eq = t_k.eq@pre->dequeue()->enqueueSet(\{ (\delta_h.generated\_event, t_h, data(\delta_h)) \mid \delta_h \in \Delta \wedge t_k \in \delta_h.receivers \})$$

■

3.3 Comparison of Decentralized Workflow Execution Architectures

In comparison to other decentralized architectures like CodAlf/BPAframe [ScMi96], Meteor$_2$ [Das+97], EvE [TGD97], our approach differs in two important points which both concern mainly flexibility features: first, we do not generate and compile different task managers from the workflow schema which implicitly contain the distributed execution knowledge (as illustrated in figure 3a), but configure task agents with their specific execution behavior (as it is described in section 4.4 and illustrated in figure 3b). Thus, the execution knowledge is explicitly represented in terms of ECA rules which are interpreted by the task agents. In particular, this rule base can be updated when the workflow schema has been changed and hence dynamic workflow changes can be supported (see section 5.2). Since the rules are structured by the transitions (or more precisely describe the transitions's behavior), an update of the execution behavior can be done incrementally.

Second, we do not create a copy of the workflow schema when a workflow case is instantiated and do not migrate it between different enactment services because workflow schema changes are hardly supported in this case. We rather follow an integrated modeling and enacting architecture, where a task agent is related to its relevant schema elements, where these interrelationships are explicitly maintained, and where the relevant execution knowledge is derived from the schema. In conjunction with a workflow schema versioning concept, this allows to support different workflow schema evolution strategies and to update the execution behavior upon schema modifications (see [JoHe98] for details). Furthermore, schema changes can be analyzed regarding to their impact on running instances.

4. Workflow Modeling and Behavior Definition on Schema Level

So far, we have introduced an execution model whose execution behavior is defined by specific ECA rules. The ECA rules are structured according to a task and a transition, respectively, so that a decentralized enactment is di-

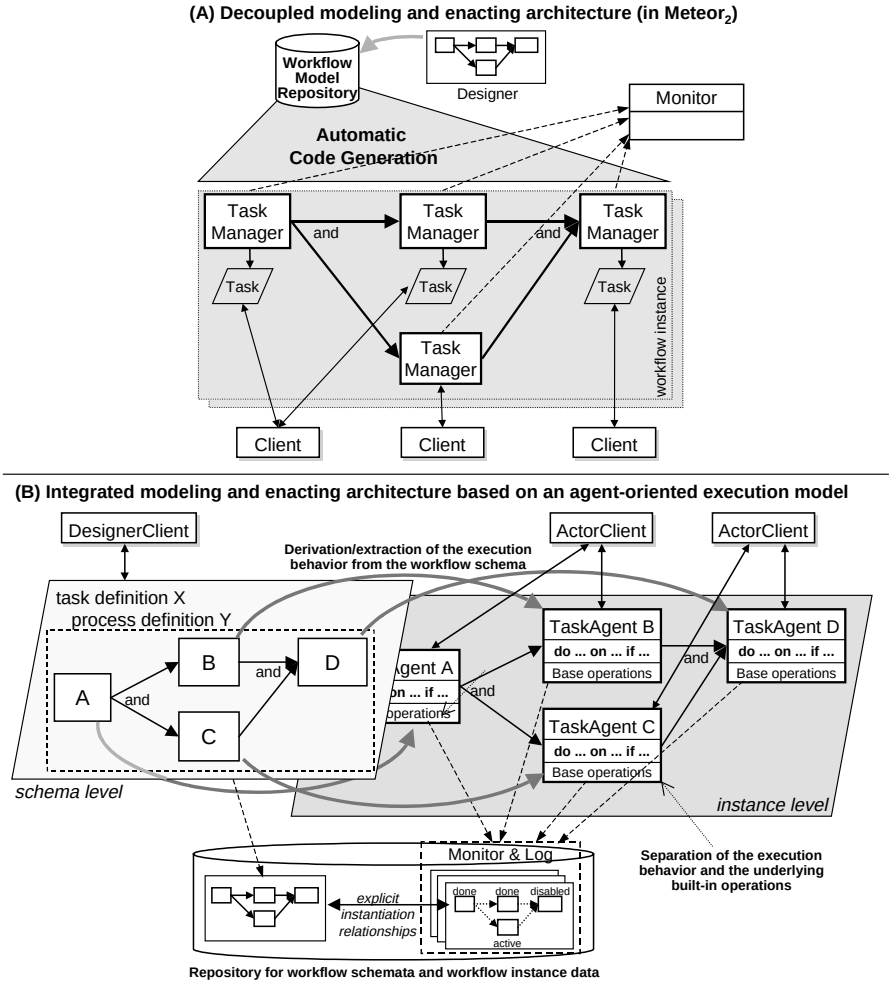


Figure 3: Decentralized workflow enactment models

rectly supported. Although this structuring would help to specify workflow schemata on a low-level of abstraction, the representation formalism remains inadequate for workflow modeling and lacks support for reusability of behavior definitions.

Therefore, we introduce in this section how the context-free and context-dependent behavior of a task can be defined and customized on schema level and how workflow can be modeled by means of task graphs on a high-level of abstraction without loosing the flexibility of rule-based specifications. The interplay of both concepts is based on user-definable control flow types which encapsulate partitioned sets of ECA rules (section 4.2 & 4.3) and the configuration of the execution behavior of a task from the workflow schema (section 4.4). We start introducing the workflow modeling concepts with an overview on the workflow meta model.

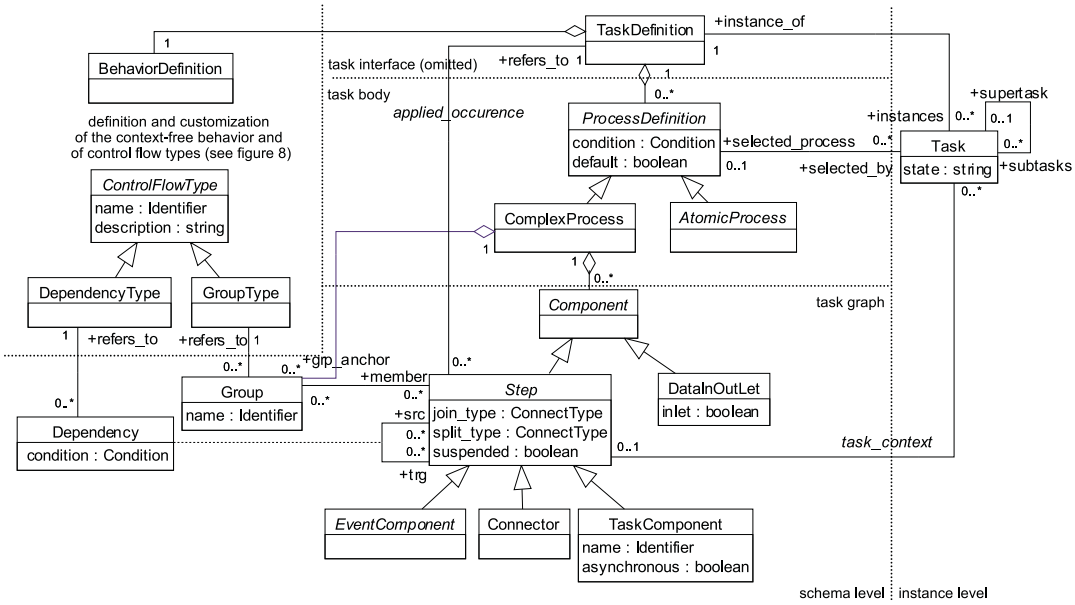


Figure 4: Workflow meta model for task and process definitions and their instance-relationships

4.1 Overview on the Workflow Modeling Concepts

A *task definition* (or task type) is the building block of our workflow meta model on schema level. It is separated into the definition of the *task interface* which specifies 'what is to do', and potentially several *process definitions* (task realization), which specify how the task may be accomplished (how to do) (see figure 4 and 5).

The *task interface* is defined by attribute, parameter, exception, and process constraint definitions (all neglected throughout this paper) and by a behavior definition. The behavior definition at the task interface specifies the external context-free behavior of a task type by a statechart variant (e.g., transactional or non-transactional) as explained in section 4.2; on the other hand, the context-dependent behavior is given by the application of a task type within a certain process definition (see below). Whereas the separation of context-free and context-dependent behavior is irrelevant for the representation of the execution behavior on instance level, it is fundamental for a modular and reusable workflow model on schema level.

A *process definition* defines how a task has to be done. The decision which process definition of a task definition is used to perform a task is taken at run-time (late binding). Every process definition has a condition which acts as a guard and restricts the allowed task realizations according to the current case. A process definition can be atomic, consisting only of a process description or system invocation, or complex.

A complex process is defined in an activity-oriented manner by a *task graph* which consists of process steps and data inlets and outlets, which are linked by control and data flow dependencies. Process steps are further divided into task components, connectors, and event components:

- *Process Steps:* A *task component* is an applied occurrence of a task definition repre-

senting 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. For every process step a split and join type (none/and/or/ xor) can be specified. In order to provide connectors independently of a task component, *connector components* are predefined as "empty" tasks which just realize splits and joins.

- *Control flow dependencies:* Process steps 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 process definitions. The semantics of a control flow dependency type is defined by ECA rules as introduced in section 4.3. These rules define fine-grained state dependencies (cf. [Att+96]) between the source and target component. On the other hand, the application of a control flow dependency within a task graph abstracts from these details and allows to model flexible processes on a high-level of abstraction.
- *Groups and blocks*: 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 process steps of a task graph; it applies the behavior, which is defined by the group relationship in terms of ECA rules, to its components (e.g., to realize mutual exclusion).
- *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.

The workflow meta model in figure 4 as well as the example in figure 5 show the different instantiation relationships which have been men-

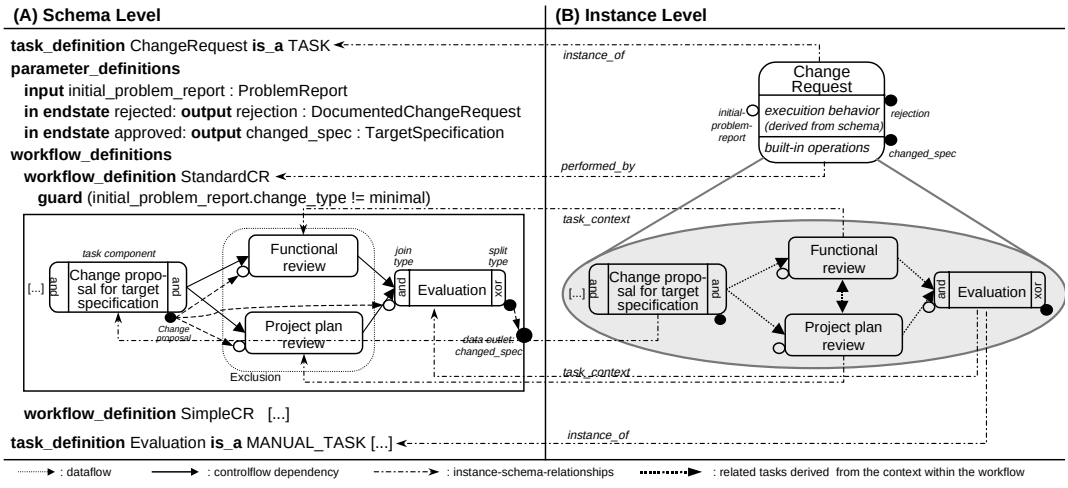


Figure 5: Example of task and process definitions and their instance-relationships

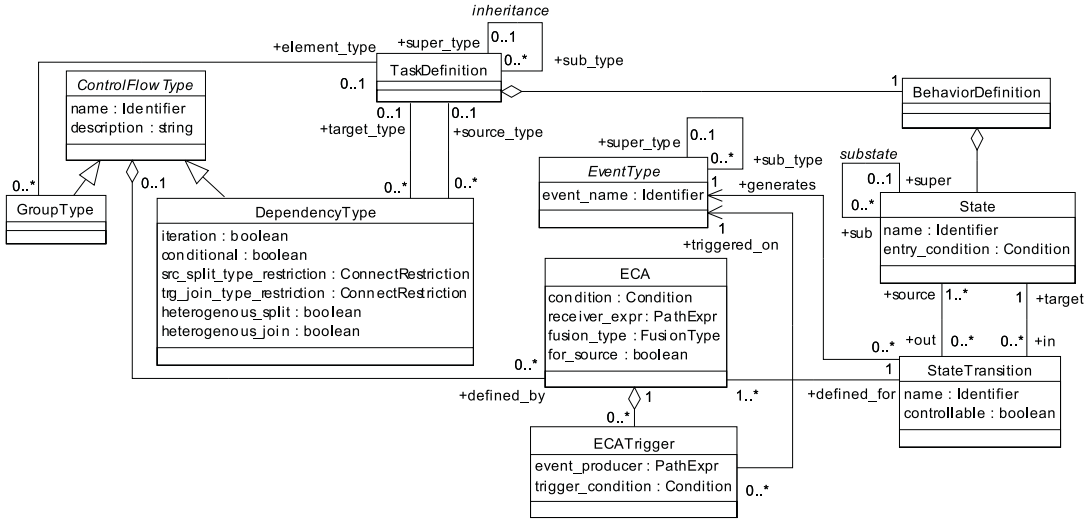


Figure 6: Meta model for context-free behavior definitions and control flow type definitions

tioned in section 3.3. Beside the explicit representation of the type-instance relationship (instance_of), also the process definition which has been selected dynamically for execution (selected_process) as well as the context of a task instance, i.e. the component it plays within a process definition (task_component), is captured. These relationships allow to derive the execution behavior of a task agent from the workflow schema, to analyze workflow schema changes according to their impact on running instances, and to update the execution behavior upon changes.

4.2 Definition of Context-Free Behavior Types with Partial Definition of ECA rules on Schema Level

The *context-free behavior* of a task type is defined by a *statechart variant*, which is encapsulated by the class BehaviorDefinition (see figure 6) The statechart defines the states and the operations/transitions of a task type. We allow for the composition of states into complex states (OR-states), but we disallow concurrent states (AND-states). 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 which has neither parameter definitions nor process definitions. Thus, the is_a hierarchy is used to define different behavior classes of tasks (e.g., non-transactional, transactional, etc.; cf. [KrSh95] 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.

Definition of ECA rules on schema level: Thus, the separation of context-free and context-dependent behavior definition leads also to a separated and relative definition of ECA rules on schema level. When we recall the ECA rule of the enable-transition of the task “ProjectPlanReview” in figure 2, we can see that some parts of the rule are context-independent (e.g., that a subtask can be enabled only if the super-task is active) whereas other parts depend on the context (e.g., that enabling the task de-

pends on finishing the change proposal). In particular, one ECA rule can be specified context-free for a transition defining context-independent triggers and application conditions; any further ECA rules which are associated with a transition are defined by control flow types which we will introduce below.

Therefore, on schema level ECA rules are defined *partially*, i.e. expressing only a certain aspect of the execution behavior. Triggers, an application condition, or a receiver expression can be omitted in a partially defined ECA rule (see EBNF in definition 3). Furthermore, on schema level we use relative path expressions when referring to related tasks instead of the absolute references on instance level. These expressions refer to super- and subtasks, to predecessor and successor task (possibly qualified by a specific dependency type), 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 traditional rule based workflow specifications like "on X.done do ...").

Example: Every task definition inherits from a *predefined task definition*, which consists of a statechart that defines the basic states, transitions (as already shown in figure 2), and context-free ECA rules. Figure 7 shows a cutout of the predefined ECA rules and gives examples of their definition on schema level. E.g., several triggers can be defined context-free for the enable transition (e.g., a supplier has released an output so that a new input is available, the execution of the super-task is resumed, dynamic changes of the process context have been performed so that the task may now be enabled and so on). Furthermore, we can partially define the application condition for the enable-transition requiring that the supertask is active (if it exists) and that all mandatory inputs are available (using a predefined predicate). On the other hand, all context-dependent aspects are left open (e.g., triggers that react on the finishing of preceding process steps; corresponding state dependencies in the application condition). Finally, a special predicate "condition_of" can be used on schema level in order to refer to the condition of another transition. This is particularly useful for the disable-transition and avoids redundant specifications.

4.3 Definition of Control Flow Types

As already mentioned, rather than providing a fixed set of control flow types, different control flow dependency types (DependencyType) and group relationship types (GroupType) can be defined by a process engineer in our approach [JoHe99]. They are defined by a label, an informal description, and a set of partially specified ECA rules – at most one for a transition – which give the semantics of the control flow type. Within the task graph, the control flow dependencies (Dependency) or group relationship (Group) 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 depending 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. Furthermore, structural restrictions on the application of the control flow types within a task graph can be specified: the allowed task types between which a control flow is modeled or the allowed combination of different control flow types.

As a first example, we briefly explain the definition of the standard end-start dependency which consists of several rules shown in figure 8b. We concentrate on the first rule, which defines an "end-start behavior" for the enable transition of the target component (using the keyword "OF target"; represented by the flag

ECA::for_source): first, the trigger 'on finished by predecessor(Standard) when condition_of(dependency)' defines that the enable transition should be invoked when a predecessor according to the dependency "Standard" has been finished and the condition of the actual dependency within a task graph holds (referred to by the placeholder condition_of(dependency)). When the condition evaluates to false, the complementary trigger of the truncate transition matches so that the task (and transitively the whole path) is truncated. Second, the application condition 'source.state= done and condition_of (dependency)' gives the state-based semantics of the end-start dependency where "source" is a placeholder for the actual source component of the dependency.

Figure 8b shows also the control flow group type "Exclusion" as an example of a group relationship type which defines a control flow dependency between an arbitrary set of process steps. The semantics of mutual exclusion is ensured by the application condition 'related-members_of(Exclusion)->forall(state != running)'. Furthermore, additional receiver expressions are defined in order to propagate the start- and finish-events between the group members.

***Definition 3**: Grammar for the textual notation of ECA rules on the schema level*

```
contextfreeRule ::= "FOR" transition "DEFINE" ECArule_body
dependencytypeRule ::= "FOR" transition "OF"
                       ("source" | "target") "DEFINE"
                       ["WITH_FUSION" fusionype] ECArule_body
grouptypeRule  ::=  "FOR" transition "DEFINE"
                       ["WITH_JOIN_TYPE" fusiontype] ECArule_body
fusiontype     ::=  "and" | "or" | "of_component" | "inverted" |
                       "overwrite"
ECArule_body   ::=  [ "ON" trigger { "," trigger } ]
                       [ "IF" application_condition ]
                       [ "SEND_TO" receiver_expr { "," receiver_expr } ]
trigger        ::=  event_name ["BY" event_producer_expr]
                       ["WHEN" trigger_condition]
```

An additional trigger for the disable transition ensures that all group members are disabled once another member has been started so that the corresponding work items will disappear from the actors' worklists.

Since a task component can be involved in several dependencies and group relationships types, the user can specify

- which task types can be related by the control flow type (source-/target_type, element_type) so that only only subtypes of the specified task types are allowed,
- whether different control flow types can be combined (heterogenous_split-/join),
- which split- and join-types are allowed for a component (src_split-/trg_join_type_restriction) and whether a dependency condition can be used (conditional) in conjunction with a particular dependency type (in particular, conditional branches (or-splits) can be used only with certain dependency types), and
- how the ECA rules defined by different control flow types are fused together for a component (fusion_type).

ECA rules of the predefined task type 'TASK'

```
FOR enable DEFINE:
  ON resumed BY supertask,
     process_selected BY supertask,
     output_released BY supplier,
     context_changed,
     [...]
  IF supertask.state=active AND
     mandatoryInputsAvailable()

FOR disable DEFINE:
  ON suspended BY supertask
     output_unreleased BY supplier,
     context_changed,
     [...]
  IF NOT condition_of(enable)

FOR finish DEFINE:
  ON finished BY subtask WHEN automatic
  IF subtasks->forall( state=done OR
                       state=not_executed)
  SEND_TO supertask
[...]
```

Figure 7: Predefined ECA rules

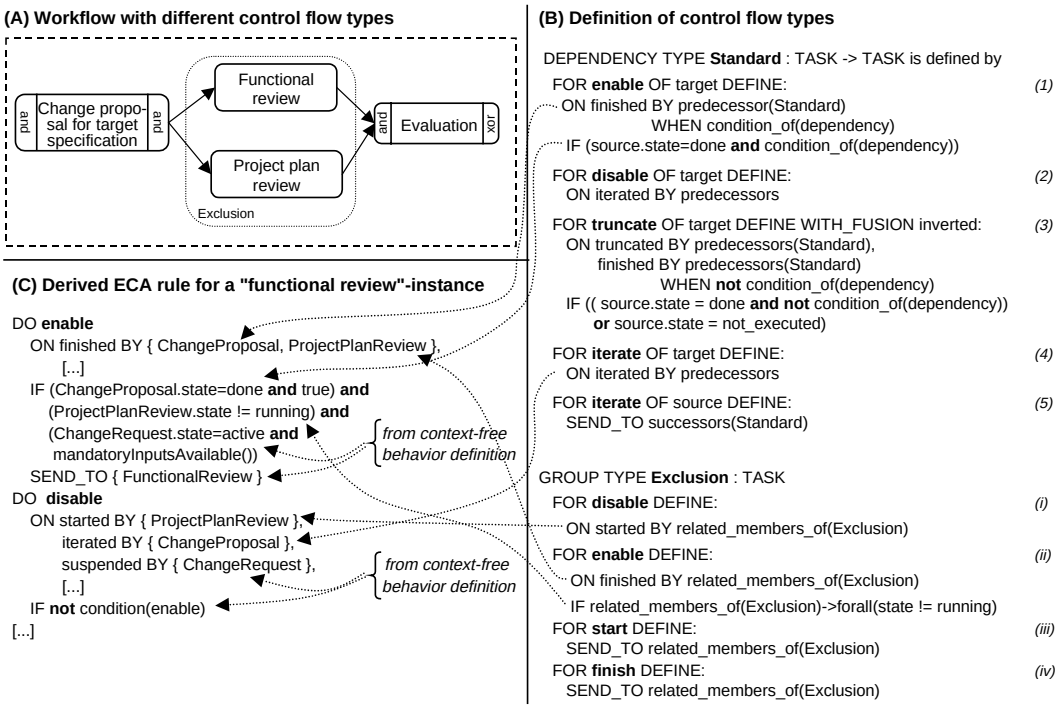


Figure 8: Example of task and process definitions and their instance-relationships

The latter is of particular interest for the derivation of the task's execution behavior and explained in the next sub-section in detail.

4.4 Configuration of the Execution Behavior of a Task Agent

The execution behavior of a task agent is defined by a set of transitions which are declared by the task type and by the transitions's execution behavior which is given by the ECA rules as introduced in section 3. In order to derive/configure the execution behavior of a task agent, we have to fuse all relevant partially specified ECA rules for every transition and to resolve the placeholders and path expressions used in the definition of ECA rules on schema level. The derivation procedure is in contrast to centralized approaches (e.g. [Kap+95], [CCPP96]) which realize a workflow engine on top of an active database and derive one global set of ECA rules. We start introducing this procedure by explaining the example shown in figure 8c. We concentrate on the enable transition of the task "FunctionalReview" and its ECA rule. For this task and transition, the following ECA rules are relevant:

- the context-free ECA rule of the task type which forms the baseline of the resulting ECA rule: in particular, its application condition is conjunctively joined with the context-dependent applications conditions.
- the context-dependent ECA rules of the Exclusion-group where the task is part of: by default, the application conditions defined by a group type are joined conjunctively; in the case of the enable condition, it is required, that the "ProjectPlanReview" is not running.

- the context-dependent ECA rules that are defined by the standard dependency type for the target component regarding to the standard dependency between “ChangeProposal” and “FunctionalReview”: by default, the application conditions which are defined by a dependency type for the target component are joined according to the join-type specified for the component (when no join-type is specified, an and-join is assumed). Therefore, the condition “ChangeProposal.state = done” is added conjunctively.
- the context-dependent ECA rules that are defined by the standard dependency type for the source component regarding to the standard dependency between “FunctionalReview” and “Evaluation”: these rules usually define to whom the event generated by the transition has to be passed and are defined for the standard dependency for propagating the finished, iterated, and truncated events (not shown for the task instance behavior in figure 8c).

The default fusion method of an ECA rule (more precisely: of their application condition) can be overwrote using the fusion types "and" (default for group types), "or", "of_component" (the default for dependency types), "inverted" and "overwrite". The inverted-fusion type fuses a condition with the complement of the component’s connect type. For example, this fusion-type is used for the ECA rule of the truncate transition (see figure 8b): a task has to be truncated in the case of an and-join when one preceding task has not been executed or the dependency condition evaluates to false, but in the case of an or-join it can be truncated only if all preceding tasks have been truncated. Finally, when using the overwrite-fusion type all application conditions defined by other relevant ECA rules are neglected, i.e. the application condition of the ECA rule is already the final condition. This requires, that different control flow types cannot be combined when they include conflicting overwrite-statements.

Beside combining the partially defined application conditions, we have to fuse also the different triggers and receivers which are defined by the different ECA rules. Independently of the specified fusion type, always the union is created for both cases (omitting the details of the union operator on triggers).

The *formal definition* of this procedure is mainly based on the definition of the *relevant* ECA rules for task agent regarding to a transition and the *fusion* of these relevant rules. An ECA rule is relevant for task agent t (see definition A1 in the appendix where OCL in the context of the above UML diagrams is used), if

- the ECA rule is defined by the statechart of the task definition of t (context-free ECA rule) (A1.1), or
- t refers to a source (target) component of a control flow dependency of type D in a task graph, and the ECA rule is defined by D for the source (target) component (A1.2), 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 to all group members) (A1.3).

On this basis, we can define the set of all transition execution behaviors which forms the execution behavior of a task agent. First, every state transition defined by the statechart variant of the task type is adopted with its structural information (A2.1). Next, the relevant ECA rules of the schema level are fused together resulting in the transition’s execution behavior on instance level. This is done by

- creating the union of the triggers that are defined by the ECA rules and by resolving the event producer expression (A2.2) (using a simple equivalence relationship on triggers so that equivalent rules are amalga-

mated by creating the union of the event producers),

- generating a application condition according to fusion-types of the ECA rules (A2.3) (for which the ECA rules are partitioned as follows
- resolving the receiver expressions and creating their union resulting in the receiver set (A2.4).

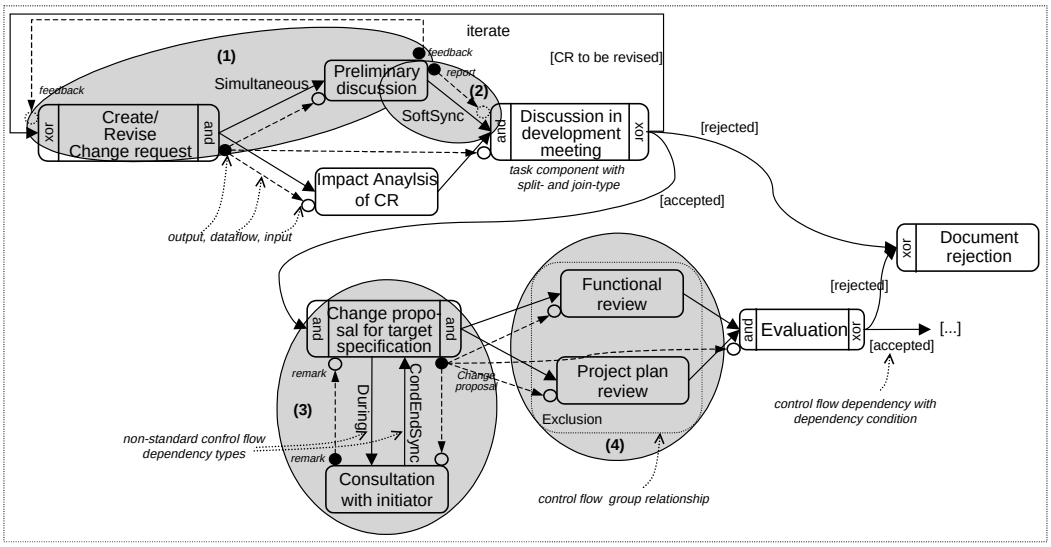
5. Behavior Definition and Adaptation for Flexible Processes

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.

5.1 Examples of Control Flow Types and their Usage for Defining Less-restrictive Workflows

Figure 9 shows a change request management process which can be found alike in any engineering domain and which has been analyzed in the MOKASSIN project [Gro+99]. The skeleton of the process is well-structured and consists of several mandatory tasks which are executed in a prescribed order: create change request, impact analysis, discussion in project meeting, change proposal, review, evaluation and acceptance. However, we find also a high degree of freedom within (see corresponding numbering in figure 9). User-definable control flow types allow to model a workflow schema on a high level of abstraction which supports this a-priori flexibility:

1. When a change request (CR) is initiated a first provisional version of the CR is often passed to the configuration control board in order to early start a preliminary discussion about the CR and the underlying problem. This results in valuable feedback information and improves the technical quality of the CR. In this case, the discussion task can start early (as soon as the first provisional version of the CR is released) and will respond with feedback information. Thus, both activities can overlap and pass intermediate results overcoming a black-box view of a task – in correspondence to the design of a reactive task agent. But, in order to avoid a chaotic process the discussion task must not terminate before the CR creation task. This kind of dependency is denoted as simultaneous [HJKW96]; its definition is given in figure 9. Furthermore, document interchange between both tasks is versioned and hence traceability is ensured.
2. Although the preliminary discussion is often needed, it is not mandatory. Therefore, the discussion in the development meeting only has to wait for the termination of the preliminary discussion if it has been started or still can be started. The Softsync-dependency (cf. [ReDa98]) indicates this behavior. The discussion task can be skipped without causing in deadlock.
3. When the CR has been accepted, a partially similar process starts for working out the change proposal of the target specification. Again, questions may arise and have to be clarified with the initiator for which an optional consultation task is provided. This task can be started as long as the change proposal task is active (during-dependency, see figure 9) and if an inquiry has passed to the consultation task (value dependency given by data flow relationships). When the consultation task has been activated, the change proposal task must not terminate before the consultation task has finished (conditional end synchronization established by CondEndSync-dependency). This scenario also includes data interchanges between running tasks.




```
DEPENDENCY TYPE simultaneous : TASK -> MANUAL_TASK is defined by
  FOR enable OF target DEFINE:                    -- relaxed activation condition:
    ON finished BY predecessor(simultaneous) WHEN condition_of(dependency)
       started BY predecessor(simultaneous) WHEN condition_of(dependency)
    IF (source.state=done or source.state=active) and condition_of(dependency)

  FOR finish OF target DEFINE:                    -- ensure that target does not
    IF source.state=done                          -- finish before source is done

  FOR start OF source DEFINE:
    SEND_TO successors(Simultaneous)
[...]
```

```
DEPENDENCY TYPE During : TASK -> TASK
(unconditional) is defined by
  FOR enable OF target DEFINE:
    ON started BY predecessors(During)
    IF (source.state=running and condition_of(dependency))
  FOR truncate OF target DEFINE:
    ON finished BY predecessors(During),
       aborted BY predecessors(During)
  FOR start OF source DEFINE:
    SEND_TO sucessors(During)
```

Figure 9: Flexible workflow of a change request management process

4. Afterwards, the created change proposal is reviewed according to functional and administrative consistency; minor changes and corrections are made directly. Therefore, both review tasks should not be performed concurrently, but can be performed with a free choice of their order. This behavior is realized by the exclusion group relationship introduced above.

5.2 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 transition, 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. Furthermore, the evolving workflow schemata are managed on the basis of a detailed workflow schema versioning concept which allows to support different evolution strategies [JoHe98].

Whether a change is allowed and how to react on it highly depends on the particular situa-

tion and the behavior of the involved tasks. For example, within several approaches [EKR95, HoJa98, ReDa98] the insertion of a new preceding task is allowed only if the task has not been started. However, an active manual task can be suspended, a batch task can be just restarted, transactional tasks can be aborted, or executed tasks can be compensated possibly. The capabilities for behavior definition and customization support such situations (see [Joe99] for details).

6. Conclusion

In this paper, we have proposed an approach to modeling and enacting of heterogeneous and flexible processes that deals with the challenging requirements of flexibility, reuse, distribution, and provision a process modeling language at a high level of abstraction. 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 supporting a-priori and a-posteriori flexibility in a WFMS. In particular, the agent-based architecture combines decentralized workflow enactment with this flexibility. Finally, 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.

Future work will focus on cross-organizational workflows. A first extension of our approach which goes in this direction has been undertaken [Gro+99]. The introduced concepts have been fully implemented in the project MOKASSIN – which has been funded by the German Ministry for Research and Technology (BMBF) – using IONAs CORBA realization Orbix 2.3.

References

[ALO96] Alloui, I.; Latrous, S.; Qquendo, F.: "A Multi-Agent Approach for Modeling, Enacting and Evolving Distributed Cooperative Software Processes". In *Software Process Technology - Fifth European Workshop EWSPT'96*, LNCS 1149, Springer, Berlin, 1996; pages 225-235.

[Att+96] Attie, P. C.; Singh, M. P.; Emerson, E. ; Sheth, A.; Rusinkiewicz, M.: "Scheduling Workflows by Enforcing Intertask Dependencies". *Distributed Systems Engineering*, 3(4) (1996), pp. 222-238.

[CCPP96] 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, 1996, pp. 94-115.

[ChSc97] Chang, J.W.; Scott, C.T.: "Agent based Workflows: TRP Support Environment (TSE)", in *ECRIM/W4G International Workshop on CSCW and the Web*, 1997.

[CiRu98] Cichocki, A.; Rusinkiewicz, M.: "Migrating Workflows", in Dogac, A. et al. (eds.): *Workflow Management Systems and Interoperability*. Berlin, Heidelberg (Springer Verlag) 1998. pp. 339-355.

[Das+97] 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.

[EKR95] Ellis, C. A.; Keddara, K.; Rozenberg, G.: "Dynamic Change Within Workflow Systems", in Comstock, N.; Ellis, C. (ed.): *Proc. of the Conf. on Organizational Computing Systems COOCS '95*. New York (ACM Press) 1995; pp. 10-21.

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

[GeKe94] Genesereth, M.R.; Ketchpel, S.P.: "Software Agents". *Communications of the ACM*, 37(7), 1994; pp. 48-53.

[GHS95] 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.

[Gro+99] Gronemann, B; Joeris, G.; Scheil, S.; Steinfort, M.; Wache, H.: "Supporting Cross-Organizational Engineering Processes by Distributed Collaborative Workflow Management - The MOKASSIN approach", in *Proc. of the 2nd Intl. Conf. on Concurrent Multidisciplinary Engineering (CME'99)* Bremen, Germany 1999.

[HaGe96] 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.

[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.

[Jen+00] Jennings, N.R.; Norman, T.J.; Faratin, P.; O'Brien, P.; Odgers, B.: "Autonomous Agents for Business Process Management" *International Journal of Applied Artificial Intelligence*, to appear, 2000.

[Jen+96] Jennings, N.R.; Faratin, P.; Johnson, M.J.; Norman, T.J.; O'Brien, P.; Wiegand, M.E.: "Agent-based Business Process Management" *International Journal of Cooperative Information Systems*, 5(2&3) (1996), pp. 105-130.

[Joe99] Joeris, G.: "Defining Flexible Workflow Execution Behaviors" in P. Dadam, M. Reichert (ed.) *Enterprise-wide and Cross-enterprise Workflow Management - Concepts, Systems, Applications*, GI Workshop Proceedings - Informatik'99, Ulmer Informatik Berichte Nr. 99-07, University of Ulm, 1999, S. 49-55.

[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; pp. 310-319.

[JoHe99] 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, 1999; pp. 88-102.

[Kap+95] 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.

[Kir96] Kirn, St.: "Kooperativ-Intelligente Softwareagenten" *IM - Information Management*, 11(1), 1996; pp. 18-28.

[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.

[Mer+96] Merz, M.; Liberman, B.; Müller-Jones, K.; Lamersdorf, W.: "Inter-organizational workflow management with mobile agents in COSM", in Proc. of the 1st Int. Conf. on Practical Applications of Intelligent Agents and Multi-Agent Technology, 1996; pp. 405-420.

[Mil+96] Miller, J. A.; Sheth, A. P.; Kochut, K. J. ; Wang, X.: "CORBA-Based Run-Time Architectures for Workflow Management Systems" *Journal of Database Management, Special Issue on Multidatabases*, 7(1) (1996), pp. 16-27.

[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), Kluwer Academic Publishers, March 1998; pp. 93-129.

[ScMi96] Schill, A.; Mittasch, C.: "Workflow Management Systems on Top of OSF DCE and OMG CORBA" *Distributed Systems Engineering*, 3(4) (1996), pp. 250-262.

[SiHu98] Singh, M. P.; Huhns, M. N.: "Multiagent Systems for Workflow". *Int. Journal of Intelligent Systems in Accouting, Finance and Management*, Vol. 8, 1999, pp. 105-117.

[Sin97] Singh, M. P.: "A Customizable Coordination Service for Autonomous Agents", in Singh, M. P.; Rao, A.; Wooldridge, M. J. (eds..): *Intelligent Agents IV. Agent Theories, Architectures, and*

Languages. Rhode Islad, USA (Springer Verlag, LNCS. 1365) 1997; pp. 93-106.

[SuOs97] 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.

[TGD97] 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.

[WoJe95] Wooldridge, M.; Jennings, N.R.: "Intelligent Agents: Theory and Practice", in *Knowledge Engineering Review*, 10(2), 1995; pp. 115-152.

[YuSc99] Yu, L.; Schmid, B.F.: "A Conceptual Framework For Agent Oriented and Role Based Workflow Modeling", in *Proc. of the 1st Int. Bi-Conference Workshop on Agent-oriented Information Systems (AOIS'99)*, Seattle, USA & Heidelberg, Germany, 1999.

Appendix

***Definition A1**: Relevant ECA rules of a task agent*

Denote TD = t.instance_of the task definition of a task instance t, s = t.step the corresponding process step of t within a task graph, and ST = TD.behaviorDefinition.state.stateTransition the set of state transitions defined for t by TD.

The *context-free ECA rule* of a transition $\tau \in$ ST is defined by the following OCL-expression

$eca_\tau^{cf} = \tau$.eCA->select(eca | eca.controlFlowType->isEmpty) (A1.1)

The *relevant context-dependent ECA rules* ECA^{cd} of a transition $\tau \in$ ST are defined by $ECA_\tau^{cd} = ECA_\tau^{dep}$->union($ECA_\tau^{group}$) with

- $ECA_\tau^{dep} = ECA^{src}$->union($ECA^{trg}$)->select(eca | eca.defined_for = τ) (A1.2)
- ECA_τ^{group} = s.group.refers_to.eCA->union(s.part_of.eCA)->select(eca | eca.stateTransition = τ) (A1.3)
- ECA^{src} = s.dependency->select(d | d.src = s).refers_to.eCA->select(eca | eca.for_source)
- ECA^{trg} = s.dependency->select(d | d.trg = s).refers_to.eCA->select(eca | **not** eca.for_source) ■

***Definition A2**: Fusion of ECA rules and derived execution behavior of a task agent*

Let TD, ST, eca_τ^{cf} and ECA_τ^{cd} be defined as above for a task t and transition $\tau \in$ ST. We partition ECA_τ^{cd} as follows:

$eca_\tau^{overwrite} = ECA_\tau^{cd}$->select(eca | eca.fusion_type = overwrite) (| $eca_\tau^{overwrite}$ | ≤ 1)

$ECA_\tau^{and} = ECA^{dep}$->select(fusion_type = and), $ECA_\tau^{comp} = ECA^{dep}$->select(fusion_type = of_component),

$ECA_\tau^{or} = ECA^{dep}$->select(fusion_type = or), $ECA_\tau^{inv} = ECA^{dep}$->select(fusion_type = inverted).

The *set* Π_τ *of triggers that are defined for a transition* τ *on schema level* is given by

Π_τ := eca^{cf}->union(ECA^{cd}).eCATrigger

Π_τ is partitioned into a set of equivalence classes $\mathit{\Pi}_\tau = \{ [\phi]_\sim \mid \phi \in \Pi_\tau \}$ by the (equivalence) relationship $\sim$ on triggers:

$\phi \sim \gamma : \Leftrightarrow \quad \phi$.event_name = γ.event_name $\wedge$ ϕ.trigger_condition = γ.trigger_condition

Then, the *execution behavior* Γ *of a task agent t* is derived from the workflow schema as follows:

$\Gamma = \{ \delta$: TransitionExecBehavior | $\tau \in$ ST $\wedge$ δ.name = τ.name $\wedge$ δ.controllable = τ.controllable $\wedge$ (A2.1)
δ.src_states = τ.source.sub*.name $\wedge$ δ.trg_state = τ.target $\wedge$ δ.generated_event = τ.generates.event_name
$\wedge$ δ.trigger = π $\wedge$ δ.application_condition = χ $\wedge$ δ.receivers = Ψ }

with

- $\pi = \{$ tt : Trigger | $[\phi]_\sim \in \mathit{\Pi}_\tau \wedge$ tt.event_name = ϕ.triggered_on.event_name $\wedge$ tt.trigger_condition = ϕ.trigger_condition $\wedge$ tt.producers = $\bigcup_{\gamma \in [\phi]_\sim}$ t.evaluatePathExpr(γ.event_producer) } (A2.2)

$$\blacklozenge\ \chi = \begin{cases} eca^{cf}.condition \wedge \tau.target.entry\_condition \wedge eca^{overwrite}.condition & \text{if } eca_\tau^{overwrite} \neq \varnothing \\ eca^{cf}.condition \wedge \tau.target.entry\_condition & \text{if } ECA_\tau^{cd} = \varnothing \wedge eca_\tau^{overwrite} = \varnothing \\ eca^{cf}.condition \wedge \tau.target.entry\_condition \wedge \chi^{cd} & \text{otherwise} \end{cases} \quad \text{(A2.3)}$$

$$\text{with } \chi^{cd} = \Big(\bigwedge_{e \in ECA_\tau^{and}} e.condition \Big) \wedge \Big(\bigvee_{e \in ECA_\tau^{or}} e.condition \Big) \wedge \Big(\Theta_{e \in ECA_\tau^{comp}} e.condition \Big) \wedge \Big(\Xi_{e \in ECA_\tau^{inv}} e.condition \Big)$$

$$\text{and } \Theta = \begin{cases} \wedge & \text{if } t.step.join\_type \in \{ \underline{none}, \underline{and} \} \\ \vee & \text{if } t.step.join\_type \in \{ \underline{xor}, \underline{or} \} \end{cases}$$

$$\Xi \in \{ \wedge, \vee \} \setminus \{ \Theta \}$$

$$\blacklozenge\ \Psi = t.evaluatePathExpr(eca^{cf}.receiver\_expr) \cup \bigcup_{e \in ECA_\tau^{cd}} t.evaluatePathExpr(e.receiver\_expr) \quad \text{(A2.4)}$$

■