Preserving Trigger-based Coordination In Interorganizational Workflows

Bill Kuechler
DEPARTMENT OF ACCOUNTING AND INFORMATION SYSTEMS
UNIVERSITY OF TEXAS AT SAN ANTONIO

Vijay K. Vaishnavi
DEPARTMENT OF COMPUTER INFORMATION SYSTEMS
GEORGIA STATE UNIVERSITY

December, 1997
GSU CIS Working Paper
CIS-97-11
Preserving Trigger-based Coordination In Interorganizational Workflows

Bill Kuechler
University of Texas at San Antonio, Department of Accounting and Information Systems
Vijay Vaishnavi
Georgia State University, Department of Computer Information Systems
e-mail: bkuechler@utsa.edu, vvaishna@gsu.edu

October 1997

Abstract:

Automated workflow management systems (WFMS) are increasingly used to schedule, track and expedite compound processes involving physically distributed workgroups, each performing a complex set of activities. Many WFMS implemented using existing workflow models work well in centrally controlled environments, but fail when used between autonomous systems. They fail because autonomous workgroups are free to change the activities that constitute their task, disrupting coordination between systems.

This paper presents a novel approach to preserving coordinating ‘triggers’ between automated workflows in autonomous systems. The mechanisms by which coordination is preserved constitute a circumscribed, but useful model of coordination between interorganizational WFMS. We present two views of the model: an architectural level to highlight its fundamental concepts, and a design level to indicate its feasibility.

The model supports three key notions on which coordination preservation is based: (1) activities and processes are modeled as smart objects which carry semantic knowledge about their goal states (2) the association of semantic information with each activity allows process definitions to be polymorphic and highly dynamic, that is, the scheduling of activities, and acceptable substitutes for activities and activity sequences can be reasoned about by the system at run-time, and (3) triggers are modeled as first class objects which carry their own semantics and can be scheduled independently of any other activity.

The Smart Object model which synthesizes object-oriented concepts, sophisticated control modeling and declarative knowledge into an active modeling environment is shown to be useful for the representation of workflows and processes which must carry significant semantic content. A concrete example of the problematic aspects of preserving coordination in autonomous workflows is developed, and the manner in which the model preserves coordination for the example is traced.

1 This work is partially supported by a research release awarded to the second author by the College of Business Administration, Georgia State University
1. Problem Statement and Example

Workflow management systems (WFMS) which work well for centrally controlled work environments frequently fail when attempting to control workflows between heterogeneous groups [Schw96] [Scac96] [Alon96]. By a centrally controlled work environment we mean one in which common definitions for tasks and information are known and respected by all workgroups using the system [Geor94] [wfmc94]. Much of the research on WFMS and nearly all commercial WFMS development has assumed centralized control. This assumption does not apply to an environment in which the workgroups exhibit varying degrees of autonomy, leading to what Schwenkreis [Schw96] has termed “the autonomy problem”. Autonomous groups are free to redefine the sequences of activities in the tasks transferred to them making predefined coordination [Malo94] with other work groups highly problematic.

The remainder of this section develops an example to demonstrate how the interpretation of tasks in different environments can effect coordination in a distributed workflow. In the next section, several current approaches to modeling flexibility in process are briefly discussed, followed by sections in which our coordination model is presented and then illustrated by reference to the example.

At a large aircraft company, a human resources workflow management system (WFMS) is used to schedule new employees efficiently through a multi-stage hiring process. One of the sub-processes of the hiring process is a security clearance. When an engineer is
hired to work in a defense related department, policy dictates that the security clearance is handled *not* by aircraft company personnel, but is routed to a military security agency. 

*The problem cannot be simplified to a tokenized request for service by the aircraft company to the military agency.* What is required is a system which coordinates human and automated activities at both sites, without human intervention, and subject to the following constraints (1) the military system has limited visibility to the aircraft system for security reasons, (2) personnel (resources performing activities) and procedures for the clearance and hiring process at both systems undergo unpredictable modification, (3) a subset of the total employee information will be common to both sites, however each site will have different information it calls Employee, and may assign different meanings (interpretations) even to the commonly held data, (4) the systems must communicate the completion of multiple *semantically equivalent* milestones during processing, despite the fact that the security clearance procedures at the different sites will have many different processing steps. The information gathered on the employee by the human resources department (the employee folder) along with the task: security_clearance, are transferred

![Diagram](image-url)
to the military site. The activities constituting the task at each of the sites are diagrammed in Figures 2 and 3 below.

Figure 2. Simplified security_clearance task definition at *aircraft* site

<table>
<thead>
<tr>
<th>Duration</th>
<th>Activity</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 day (phone)</td>
<td>Verify education</td>
<td>preliminary_clearance_complete</td>
</tr>
<tr>
<td>3 days (fax)</td>
<td>Check for in-state criminal record</td>
<td></td>
</tr>
<tr>
<td>2 weeks</td>
<td>Check FBI for criminal record</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Simplified security_clearance task definition at *military* site

<table>
<thead>
<tr>
<th>Duration</th>
<th>Activity</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 day (phone)</td>
<td>Verify education</td>
<td>Lost coordination !</td>
</tr>
<tr>
<td>3 weeks</td>
<td>Military intelligence investigation for national criminal activity</td>
<td>The activity defining the trigger has been eliminated.</td>
</tr>
<tr>
<td>2 months</td>
<td>Check interpol for international criminal record</td>
<td></td>
</tr>
</tbody>
</table>

Note that the trigger, *preliminary_clearance_complete* occurs after 3 days, since all activities can be started in parallel, *when the task is handled at the aircraft site*. This is valuable since the hiring process is extensive, and beginning other activities in parallel with the FBI check saves 10 days in employee processing time.

When the task and data folder are transferred to the military site, the task will necessarily be redefined. The military site is *autonomous*, that is, *local process definitions take precedence over external process definitions*. Note that following redefinition the trigger [Joos94] (preliminary_clearance_completed) has been eliminated due to task substitution, extending the hiring process by over two months.

As an isolated occurrence, the disruption of coordination in the example is simply a nuisance. However if repeated several hundred times a year the cost to the organization is substantial. More generally, considering the increasingly global nature of the workplace, the *automated* reconciliation of intergroup task coordination has significant economic
implications [Olea97]. *Virtual corporations*, substantial increases in the *outsourcing* of non-strategic corporate function, *concurrent* and *round-the-clock engineering* and *workgroup empowerment* are some of the current organizational trends which increase the number of (semi)autonomous groups in a business workflow, and result in rapid shifts of work from one group to another. These organizational forms are precisely those most in need of the increased efficiency and communication that derives from automated workflow management systems, yet as noted, the performance of many current WFMS degrades when used between autonomous workgroups.

2. **Current Research on Flexible Representation of Process**

Surveys of coordination literature [Malo94][KuecIP] and workflow modeling research [Alon96] indicate that formal models of process coordination are still an open research question. To the best of our knowledge, the model presented here is the first attempt at an implementable model of coordination between autonomous WFMS. The research contributions below are therefore not alternative approaches to our specific research problem. Rather the models surveyed here highlight both the importance of flexibility in process modeling, and various approaches to the many problems inherent in modeling processes in open environments.

Coordination problems are ubiquitous, and we have the advantage of being able to draw from multiple fields. In all, we find the recognition of the inadequacy of fixed representations for modeling real-world work situations.
Cao and Sanderson [Cao95] find fixed representations of process inadequate even for the constrained field of robotic workcells. Traditional deterministic planning representations are simply not robust enough under real world conditions. Their approach is to expand traditional Petri net models of process by allowing the transitions between states to be expressed as fuzzy variables, interpreted by production rules. While suggesting interesting techniques, their research addresses a sufficiently different problem and thus their solution is not immediately applicable to workflows. Specifically, their domain requires fixed sequencing of activities, and many workflows, as noted in the OIS and organizational sociology literature, are dynamically resequenced in response to changes in the environment.

The HFBP model [Mori96] of business workflows is specifically intended to describe work processes between interoperating workflows in autonomous organizations. Drawing from research in fault tolerant software systems, they model processes in terms of an attribute grammar. While successful in addressing certain types of error recovery in interoperating WFMS (due to inherent capabilities of attribute grammars) they note that the more general problem of dynamic change in process definitions for interoperating workflows is not sufficiently handled by the attribute grammar approach alone.

Clarence Ellis and his research collaborators [Ellis94][Ellis95] have been among the longest standing exponents of the disfunctionality of rigidly defined work processes. Quoting earlier researchers, they term automated workflow using fixed process
definitions “the automation of an illusion”. Ellis and Wainer [Elli94] sketch an approach to flexible modeling of workflows in which high level business goals are mapped to an information control net (ICN) representation of workflow. This model however, is informal and descriptive (with regard to intentional information), and though supportive of both our motivation and technique (incorporation of intention into WFMS models) it is not sufficiently rigorous to allow automated reasoning about the effects of dynamic changes to process definitions.

Much research in office information systems (OIS) describes the complexity of actual work environments, and is acutely aware of the inadequacy of fixed process models for support of real-world work situations [Gers86][Hewi86][Hewi91][Bola92]. The following two models come from that area.

The Partially Shared View scheme [Lee90] is a technique for enhancing the utility of computer supported collaborative work systems used between autonomous groups with differing worldviews and correspondingly different functional requirements. The scheme has been implemented in the Object Lens collaborative work support system, and attaches a type hierarchy to each work object the system supports. The type hierarchy is assumed at least partially shared (especially at its higher levels) by all users of the system. By determining correspondences with a known hierarchy, an object otherwise unknown to a user group can be at least partially recognized and usefully processed. We have adopted this technique by analogy: the knowledge we use for inferences is different (as detailed in
Section 4), but our partial matching to a hierarchical structure to make inferences about an unknown entity is very similar.

The AMS formalism [Ang94] is both a model and a collaborative work support system implementation. The formalism can be characterized as “loosely grammatical” in that the activity sequences that make up a workflow are sequentially expanded from a root “abstract activity” by a series of productions. Stressing flexibility, however, it does not attempt to formalize the grammar in the manner of HSBP. The allowed productions by which non-atomic activities can be expanded are: (1) an elementary activity, (2) an activity network, and (3) a memory organization packet for activities (MOPA). The MOPA draws from the work of Schank and others in natural language understanding and is responsible for much of the flexibility of the system. A greatly simplified description of a MOPA is as an abstract script that represents an activity sequence in general terms, and is instantiated to actual activities by the run-time context. Though a substantial contribution to flexible office work process models, AMS models work in a single environment, and have no mechanisms for preserving coordination between interoperating systems.

Table 1 summarizes the adaptive / flexible work process models discussed in this section.

<table>
<thead>
<tr>
<th>Model</th>
<th>Authors</th>
<th>Flexibility Enhancement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuzzy Petri Nets</td>
<td>Cao and Sanderson, 1995</td>
<td>fuzzy transitions to augment Petri net models</td>
</tr>
<tr>
<td>HSBP</td>
<td>Morita, et. al., 1996</td>
<td>Attribute grammar description of process</td>
</tr>
<tr>
<td>Goal based ICN</td>
<td>Ellis and Wainer, 1994</td>
<td>Goals informally enhance information control nets</td>
</tr>
<tr>
<td>PSV</td>
<td>Lee &amp; Malone, 1990</td>
<td>Documents carry type/behavior hierarchy allowing partial interpretation</td>
</tr>
<tr>
<td>AMS</td>
<td>Ang &amp; Hong, 1994</td>
<td>Generalized scripts, which are situationally instantiated</td>
</tr>
</tbody>
</table>

Table 1: Flexible Workflow Models
3. The Process Architecture

We have divided our conceptual model into two major components, a process architecture and a knowledge architecture. This division is typical of knowledge engineering, where a Process and an Expertise analysis model are defined for every system [Tans93] [Chan86]. We use the term architecture, to denote the significance of the structure possessed by the two descriptions, in addition to their content. The knowledge architecture, which articulates the intelligence used in maintaining coordination, is described in Section 4. The process architecture, which highlights the major functions required for dynamically maintaining coordination between loosely coupled (autonomous) workflows, is shown in Figure 4. At this level of abstraction it is apparent that its basic structure is similar to well known

![Diagram of process architecture]

Figure 4: Highest Level Process Architecture for Dynamic Coordination in WFMS

models for analogous tasks: planning, understanding from plans, and constraint based scheduling.
Process Overview

Moving from left to right in Figure 4: the original workflow definition (activities, their priorities, etc. as described in Section 4) and a modification of that original definition are available to the model. The comparison module determines differences and similarities between the definitions. In our earlier model, comparison and interpretation were combined in a single functional module [Kuec96]. We discovered the comparison functions to be both complex enough and interesting enough (from the standpoint of an explanatory model) to warrant separate consideration.

After comparison, the difference data (an entity that will figure prominently in the design model of Section 5) is passed to the interpretation module. The interpretation module, working from a goal oriented description of process, and a domain ‘common sense’ knowledge base, attempts to ‘make sense’ of the differences, and reconcile them to known elements in the original process definition. If this activity is successful, as judged by control and evaluation heuristics, the ‘recognized’ activity set is passed to the scheduler. That module attempts to reschedule coordinating triggers in a manner that satisfies both ‘hard’ constraints, such as fixed times for events, and fixed predecessor / successor relations and the semantics attached to the coordinating triggers.

If the scheduler is unsuccessful, the control module has two strategies available to it, derived from observation of workgroups and general problem solving behavior.

Typically, the initial strategy would be to request alternatives, first from the scheduler,
and should this fail, from the interpreter. If alternative generation fails, the second strategy is that of progressive constraint relaxation. If neither strategy yields results within an acceptable time, the process has failed. Note that failure to achieve an acceptable result is not a failure of the model to accurately capture the essential elements of the coordination process as it occurs in actual workgroups.

**Detailed module descriptions**

The principal functions of the *comparison module* are (1) identify changed *individual activities*, i.e. single activities bracketed by known activity sequences, (2) identify changed *sequences*, i.e. strings of activities specified by explicit predecessor and successor reference, which are common to both work definitions, and (3) identify altered *goals*. Though we wish to stay at an implementation independent level for this discussion, it is appropriate to note that the functions of this module are non-trivial. They are addressed further in the design description of Section 5.

The principal function of the *interpretation module* is to attach semantics to the workflow definition differences that have been identified by the comparison module. For explanatory and computational purposes, the main function has been divided into (1) activity substitution recognition and (2) goal difference interpretation. Activity substitution recognition determines that an activity of a different name, or trivially different attributes has been substituted for another. Note that the *triviality* of changes is relative, and must be determined from domain heuristics, which can in turn be overridden
by site-specific considerations. Goal difference interpretation is more complex. Guided by a heuristic to 'seek similarity', this activity identifies the degree of overlap in functionality and activity between altered goals in two work descriptions, and attempts to affix a degree of similarity. A heuristic ranking of similarity is used by the scheduling module.

The scheduler, in addition to performing constraint based scheduling of activities, applies domain specific and domain independent heuristics to the scheduling of coordinating communications. This is required, for example, when changes to the workflow definition have resulted in the elaboration of a single task in the original definition into multiple activities in the changed definition. Heuristics may be used to determine a point within the elaborated activity at which to communicate state completion to a cooperating workflow. The scheduler, like the interpretation module, is capable of generating multiple alternative solutions.

The principal functions of the control module are (1) evaluation of the results of the three computational modules, and (2) control of the process based on result evaluations. Control is assumed by the evaluation module in two ways: resource driven interrupts, principally time, may cause evaluation to begin, however the evaluation process is usually called, by either the completion of a modules function, or the modules failure, which results in a supervisory call for meta-level evaluation. As discussed in Section 5, the Smart Object Language chosen for our formal model [Vais96] naturally and effectively handles such shifts in reasoning level.
4. The Knowledge Architecture

We now consider the knowledge content of the model, describing and classifying the knowledge the model contains, and relating it to the Process Architecture of the prior section. The major components of the Knowledge Architecture, and their relationships are diagrammed in Figure 5.

![Figure 5. Knowledge Architecture of the WFMS Coordination Model](image)

Description of the components logically begins with the lexicon, since all other components must be defined with reference to it. A necessary basis for understanding (or reconstructing) intentions from activities is a shared minimum set of beliefs for establishing context for the activities [Loch90]. In our model we call this belief set the lexicon, a term widely used for similar structures for semantic reconciliation across

<table>
<thead>
<tr>
<th>verify</th>
<th>the activity of confirming the accuracy of information</th>
</tr>
</thead>
<tbody>
<tr>
<td>information</td>
<td>data which is used as the basis of decisions in a human activity system</td>
</tr>
<tr>
<td>activity</td>
<td>a set of actions required on the part of an agent to satisfy a service request</td>
</tr>
<tr>
<td>accurate</td>
<td>having the state of true</td>
</tr>
</tbody>
</table>

**Figure 6. Dictionary and Concept Structure Entries in the Lexicon**
multiple databases [Pito95] [Weig95]. The lexicon is the set of entities and relationships most fundamental to the specific domain of the coordinating workflows, usually explicated through domain or ontological analysis [Kuec98][Frei88].

The lexicon itself contains only information about the domain that is shared by all systems in the WFMS network and minimizes the need for inter-site communications. Local constraints are expressed in the work definition unique to a site. However even though different expressions may be encountered at different sites, the elements making up the expressions must be found in the lexicon, and the relations may not logically contradict those found in the lexicon. Figure 6 gives an example of lexicon entries.

The second primary knowledge structure of the model is the process grammar, an attributed grammar for which the attributes are semantic actions (also known as a semantic grammar). Succinctly, the process grammar is a set of rules that describe allowable ways of constructing activity sequences from goals in much the same way language grammars describe the construction of allowable sentences in a language [Vili90]. The formal properties of grammars give the model much of its computational power. The result of applying a grammar yields the third knowledge structure of the model, which we term the work description. A work description expresses a set of activities that give the specifics of work to be performed, along with the goals, constraints, and the decisions made to pursue the goals using specific activities.
Figure 7: Knowledge levels in the STM work description

Though much of the plan recognition literature works solely from intentional (goal) information. We find it useful to introduce a concept of generalized tasks or abstracted functionality, in addition to goals, and activities. The utility of this type of information derives from an analysis of production WFMS and greatly assists in recognition of the common types of elementary process changes discussed in [KuecIP]. Further, it more closely models the manner in which individuals reason about and discuss their work environments.

An abstraction of a complete work description is diagrammed in Figure 7.

<table>
<thead>
<tr>
<th>Goal Nodes</th>
<th>Generalized Function Nodes</th>
<th>Activity Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name (from Lexicon)</td>
<td>Name (from Lexicon)</td>
<td>Name</td>
</tr>
<tr>
<td>Formal Description (semantic net)</td>
<td>Formal Description (semantic net)</td>
<td>Formal Description (semantic net)</td>
</tr>
<tr>
<td>predicate relations from Lexicon</td>
<td>predicate relations from Lexicon</td>
<td>predicate relations from Lexicon</td>
</tr>
<tr>
<td>Description (natural language)</td>
<td>Description (natural language)</td>
<td>Description (natural language)</td>
</tr>
<tr>
<td>Parent Node</td>
<td>Parent Node</td>
<td>Parent Node (generalized functionality)</td>
</tr>
<tr>
<td>list of struct:</td>
<td>list of struct:</td>
<td>list of:</td>
</tr>
<tr>
<td>Attribute</td>
<td>Attribute</td>
<td>Constraint on activity</td>
</tr>
<tr>
<td>.Constraint on Attribute</td>
<td>.Constraint on Attribute</td>
<td>Hard constraint data (predecessor activities, successor activities, temporal)</td>
</tr>
<tr>
<td>Contribution (to goal achievement) of Attribute</td>
<td>Contribution (to achievement) of Parent goal</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Knowledge for each Work Description Node Type

Nodes at different knowledge levels have different structures, as shown in Table 2. At the root level a new design starts from an bundle of process desiderata which can be
decomposed into progressively more specific attributes which the process must possess, and constraints on these.

At the transition from subgoals to functions, the attributes are sufficiently specified that functions can be linked to them as production means. Based on environmental factors, specific activities are selected to perform the selected functions. When sequenced according to precedence, timing and other constraints, the set of activities that are the leaf nodes of the work description constitute a traditional workflow description, as might be processed by the *workflow enactment engine* [wfmc94] of a WFMS.

All allowable work descriptions (work processes, with goal information) can be *generated* from top down application of the design grammar. The *interpretation* module uses this capability to generate alternative sequences of activities. All allowable activity sequences can be *recognized* in terms of the goals from which they were derived by bottom up parsing of the final activity sets, which are taken as sentences in the language defined by the grammar. The *interpretation* module uses this capability to attach semantics to the information passed to it by the comparison module.

The fourth knowledge structure is the *active rule base*, sets of logical rules which are invoked by the process modules to derive actions based on the lexicon, process grammar, and process work descriptions. The rule base is broadly partitioned into domain knowledge, unique to each work domain, general knowledge which does not change, such as general rules about coordination and workflow, action level knowledge which
derives inferences directly from knowledge structures, and meta-level knowledge which controls the activity level inferences and evaluates their results.

The rule base divides naturally into finer, non-interacting partitions along functional lines as set forth in the Smart Object [Vais97] design model in the next section.

5. Implementation considerations: A design model

A conceptual model, such as the architectural model, operates deliberately at a high level of abstraction to communicate significant ideas. If implementation is an issue as it is for us (since the value of the concepts depend partly on the degree to which they may be included in working systems), then it is desirable to augment the description with a more detailed model. This section provides an object oriented design level model, based on Smart Object concepts. Figure 8 is a graphical and narrative overview of the Smart Object Model. A brief discussion of the most significant ideas that Smart Objects add to conventional object oriented modeling is given prior to presenting the design.

An overview of the Smart Object Model

Smart Objects are active objects that contain knowledge in various forms, which describe the behavior of the objects, both individually and when working together as a system. Separate sections of each Smart Object contain knowledge of different aspects of the objects behavior. The Methods section contains knowledge of the behavior of the
modeled element itself. An Interface section captures the description of the interaction of the object with other Smart Objects in the system. Data about the state of the object is contained in the Attributes section, along with rules which describe allowable values for the data, and actions to be taken automatically when the data is added, changed, or accessed. The most unique portion of a Smart Object is the Monitor section, which contains the control descriptions for the system as a whole and for each individual object. The monitor is a higher level of control that determines how to interpret each rule in light of the current state of the system.

![Smart Object Model Framework Diagram]

• Real world systems are conceptualized as a network of minimally coupled active entities called Smart Objects (SO's). The network of SO's fully partitions all knowledge contained in the model of the system. Different knowledge representations can be used to optimally support different tasks (procedural vs. declarative knowledge, for example). SO's can inherit knowledge and data structures analogously to facilities available in most OOP languages. SO's are themselves partitioned into: domain knowledge, state information, control knowledge and interface information.

• Control is abstracted by explicit control knowledge and an inter-object control architecture that defines the operationalization of the control knowledge. The object structure together with inheritance give support to reusable domain frameworks.

• A multi-modal, logic engine interprets the modeled knowledge. Continuous inference cycles derive new knowledge and system states from the existing state, and information from the system environment.

Figure 8. The Smart Object Model Framework

A completed Smart Object model is an active system that is “run” by an inference engine, in the same way a computer runs a program written in an conventional programming language. Systems of Smart Objects are extremely dynamic, and are capable of creating new objects of predefined types, new objects of new types and new rules, based on the ability to observe their own past activity (reflection). Such systems are sometimes called adaptive or ‘learning’ systems.
The manner in which the semantics of high level designs are actively interpreted by a Smart Object system is diagrammed in Figure 9.

![Logical View and Architectural View](image)

**Figure 9. Logical and Architectural View of a Smart Object Model**

The *logical view* portion of Figure 9 depicts a typical object-oriented system model, as would be generated by the systems analysis phase of many object-oriented analysis methodologies. The *architectural view* portion of Figure 9 depicts the *active interpretation* of the same model, after population of the Smart Objects that compose it with declarative control and domain knowledge.

In [Vais96] we demonstrated the general utility of the Smart Object model to WFMS modeling. When instantiated, the Smart Object paradigm [Vais97] results in active, object-oriented models in which relationships between objects and control flows are explicitly expressed with declarative knowledge representations (rules).
In the Smart Object representation of workflows, activities, goals and functions are Smart Object instances, as detailed in the design description. Constraints on activities are represented as rules in the Attributes section of the objects. A portion of the Attributes section of our example trigger, preliminary_clearance_complete, is shown in Figure 10.

```plaintext
attributes {activity: preliminary_clearance_complete}
state
  scheduled
  range : boolean
  derivedFrom : ConstraintsMet and OnAgenda
<other state definitions>
data
  name
  actor
  type
  range : (activity, trigger)
  must_follow
  concurrent_with
  range : array of activity
  must_complete_within
  range : days
  comp_time_measured_from
<other data definitions>
endAttributes

Figure 10. Attributes of task objects (partial)

There are substantial differences between Smart Object methods and the methods of most object oriented programming languages. A conventional method is a short procedural function definition. A method call in a Smart Object model makes a set of related production rules (the Method) the focus of the inference engine. Through calls within a given Method to Methods in other Smart Objects, other rules may be added to the active inference set. When all rules are assembled, the inference engine and rule set behave similarly to other production rule based programming systems, such as OPS5.
When a rule determines a process has finished, or no more rules may fire, control passes to a separate inference cycle in the Monitor of the Smart Object (which in this case contains a portion of the control knowledge from the architectural model). One way of viewing the operation of a Smart Object model is that each message to a Smart Object method dynamically assembles a rule based expert system for a specific function. That system then examines the data pertinent to that function, draws inferences from it, and takes actions based on those inferences. The key points are (1) that Smart Object methods generally perform far more actions than the methods in a conventional OO language design and (2) most Smart Object methods represent actions declaratively using rules rather than procedurally, and (3) a separate set of high level rules, the Monitor, exercise control over method rule firings.

A Smart Object Design for Coordination Preservation

In an object-oriented design, each object contains data and related functionality. Using Smart Object Language, passive data is contained in the Attributes section, and functionality is captured as rules. Mapping from our conceptual model to a Smart Object design becomes a matter of deciding which knowledge elements are most critical, establishing them as classes, and distributing the functionality of the process architecture and the remaining knowledge from the knowledge architecture over them. A knowledge structure taken directly from the knowledge architecture for the design is Work Description. A second key structure, Difference Data, is implicit in the architectural description; it contains the results of the comparison of two Work Descriptions. Both
**Difference Data** and **Work Description** are modeled as aggregate objects [Rumb94], that is, conceptual entities composed of multiple, tightly coupled object classes. In both cases many of the objects that make up each structure share an inheritance hierarchy as well as close structural coupling.

The **Work Description** begins with a **WD_root** class, which is the identity node of a workflow description. Its attributes are **name**, **source**, and a number of 'bookkeeping' or scratch variables, used in processing, such as **Alternate_number**, **status**, etc. The **WD-Root** is linked to node classes, all specializations of (descended from) an abstract class, **WD_node**: **Activity_node**, **Goal_node**, **Function_node**. Most of the non-trivial attributes for the **WD_node** specializations are taken directly from Table 2.

The **Process grammar** from the knowledge architecture, is mapped to a set of Smart Object methods (production rule sets) distributed over **Work Description** nodes. In this way, **Work Descriptions** can 'recognize themselves' and 'generate themselves' in keeping with the OO metaphor. Note that production rules have a long standing as a formal definition means for language grammars [Clea77] and for augmented grammars, such as our process grammar [Fire88]. When the rules are active, as are the methods of a Smart Object, the definition is also the implementation. That is, the grammar rules have consequent actions that create new nodes or modify the attributes of nodes, etc.

Within the design model triggers are **privileged activities**, derived from a separate **Coordination_activity** class hierarchy. Though they must be must be **scheduled** along
with WD derived work activities, they have additional parameters not required for work activities. General heuristics on process coordination and specific rules on what constitutes coordinated behavior in a given work domain are specified in the Coordination activity hierarchy methods, along with attributes needed to store coordination related state values for processing. The knowledge contained in this class structure is distributed over the Interpretation and Scheduling modules in the process architecture model. The privileged status of Trigger activities is conceptually valid (as well as from a design standpoint), since it highlights the fact that Triggers are the only activities concerned with the overall process occurring across the multiple sites of the linked WFMS’. Virtually all the functionality for Interpretation functions of the architectural model are contained in the methods of the Coordination activity classes.

The second major object subsystem is the Difference Data (DD) structure. The functions of the Comparison module of the architectural model, are wholly distributed over the classes of the structure. The general and domain specific knowledge required for the comparison process is contained in Smart Object methods within the classes. The knowledge is in the form of procedural functions, and rules for comparing two workflow definitions (work descriptions).

A (DD) generates and stores the results of the comparison of two WD’s. It is structurally isomorphic to the superposition of the compared work descriptions, that is to the overlaying of both WD tree structures. Where two nodes are identical in both WD’s, the DD has a single node so indicating. Where the WD’s have comparable nodes with
different parameters, the DD has also a single node, storing the parameter differences.

When the WD's differ structurally, the DD node just above where the difference begins
‘sprouts’ the subtrees of both WD's. Such heuristic information as can be gained from a
comparison of the subtrees is stored in the highest common node. For example, two
different lowest level goal nodes may result in extended sequences of common activities
in two work processes. Under some circumstances, this information might allow the valid
inference that one goal statement is simply a rephrasing in unfamiliar terms of the
understood goal node. (The inference is a parsing of the ‘activity sentence’ using the
design grammar).

A DD begins with a \textit{DD\_root node} which identifies the comparison (which two WD's are
being compared\footnote{Since a grammar can generate multiple full WD's from a single partially understood WD, there can be multiple DD's, each comparing a generated WD to an 'original' WD.}) and stores aggregate comparison information. The root node always
exists to ‘anchor’ comparisons, since the root nodes of all compared WD's are considered
to be identical, that is, to represent the intention to perform the identical highest level
function. By aggregate information we mean states which are not dependent on direct
comparison of two WD nodes, for example the state flag indicating the two WD's differ
only at the activity level. This information is used primarily by control rules to direct
inference strategies. WD's which differ only at lower levels are more easily reconciled
(interpreted) than those that differ at goal levels. Likewise, WD's which differ at a single
goal are more easily interpreted (and invoke different rule sets) than those which differ in
multiple goals, since in the latter case goal conflicts \cite{Wile84} and interactions \cite{Crow94}
\cite{Crow92} must be considered.
Analogous to the specialization of nodes in the WD, DD nodes below the root are descended from a common \textit{DD\_node} class, and specialized for the type of node to node comparison data stored there: \textit{DD\_goal\_node}, \textit{DD\_function\_node}, \textit{DD\_activity\_node}. Comparison functions (Methods) specific to each type of node are associated with the appropriate class. Comparison functions which act on the entire WD structure are implemented in the \textit{DD\_root}\textsuperscript{3}.

Several additional entities were required for the design which were implicit in the architectural model(s). A Smart Object class, \textit{Work\_request} handles all interface activities with the host operating system; in practice, we envision WFMS implementations calling an implementation of our model as a utility subsystem. \textit{Work\_request} attributes include \textit{source}, \textit{date}, \textit{associated data} on which the activities of the work are to be performed (e.g. ‘Smith’s insurance claim’, a ‘folder’ in workflow terminology), and ‘bookkeeping’ variables associated with the minutia of working, computerized processing. \textit{Work\_Request} is also contains much of the knowledge associated with the control module in the architectural model, since it is logically associated with the process as a whole rather than an individual WD. It is also, logically, the class that initiates processing.

A database, the \textit{Process Repository} is implicit in the functioning of the architectural model. It is the source of local process definitions (if they exist) to be substituted for and

\footnote{3 Since many of the comparison functions involve procedural algorithms, such as depth and breadth first tree structure traverses, for clarity and efficiency they are implemented procedurally using calls to host language routines, rather than declaratively.}
compared with external definitions for goal-equivalent processes. Note that the run-time incorporation of previously defined active structures (such as WD's in this design) into a running system is most easily handled by an interpretive implementation language. In our implementation for example, using KAPPA-PC as the host language, a traditional database is queried by keyword. Each record in the database references the name of a file defining an instance of a *Work Description*. Simply reading the work description file with a KAPPA command makes the full definition, data and rules, active in the already functioning model. Though more complex, requiring multiple indices and multiple record types to implement, the *lexicon* also becomes an external database structure in the design, interrogated by Smart Object methods.

Recall from the description of the architectural model that the scheduling module has two primary functions: (1) the detection of constraint violations on such as time and predecessor/successor relations in the overall activity set, and (2) the rescheduling and parameter generation for Trigger activities. Most of the intelligence for normal activity scheduling is contained in the *Activity_node* class mentioned the discussion of the *Work Description*. Most of the methods implementing trigger scheduling are contained in the *Coordination_activity* class hierarchy. This division allows objects to ‘maintain themselves’ and results in an efficient and comprehensible partitioning of the knowledge required for scheduling.

The mappings from the conceptual, architectural model to the detailed design model are summarized in Table 3. Notice that in the design model, the process knowledge for the
architectural modules is in most cases, distributed over a number of classes. While the
distribution is well motivated by design considerations, the dispersion makes it more
difficult to visualize key concepts of the overall model. As an attack on this problem
some OO modeling techniques introduce ‘Subjects’ to model clusters of classes as single
conceptual entities [Coad91], however we feel the dual model presentation is clearer in
this case.

<table>
<thead>
<tr>
<th>Architectural Component</th>
<th>(maps to) Design Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work Description</td>
<td>Object structure: WD_root, Activity_node, Goal_node, Function_node, decended from parent WD_node class</td>
</tr>
<tr>
<td>Process grammar</td>
<td>Rule sets (Smart Object methods) distributed over design rational node classes</td>
</tr>
<tr>
<td>Lexicon</td>
<td>Multi-index, multi-record type external database</td>
</tr>
<tr>
<td>Control module functionality</td>
<td>Distributed in the Monitor rule sets of WD-root, and node objects, and the Methods of Work-request</td>
</tr>
<tr>
<td>Interpretation module functionality</td>
<td>Rule sets (Smart Object methods) in the Coordination_activity class hierarchy</td>
</tr>
<tr>
<td>Scheduling module functionality</td>
<td>Rule sets (Smart Object methods) in the Activity_node components of the Work Description, and in the Coordination_activity class hierarchy</td>
</tr>
<tr>
<td>Comparison module functionality</td>
<td>Distributed over method rule sets, and calls to procedural functions in Difference Data nodes</td>
</tr>
<tr>
<td>Implied in overall system</td>
<td>Work_request class: control and operating system (WFMS) interface functions</td>
</tr>
<tr>
<td>Implied in functional description</td>
<td>Difference Data</td>
</tr>
<tr>
<td>Implied in functional description</td>
<td>Process repository: an external database; persistent storage for Work Descriptions</td>
</tr>
</tbody>
</table>

Table 3: Conceptual model to Design model mapping

The design model does have a distinct advantage over the architectural model in
describing interactions between model components at more granular level. The *Object
Interaction Diagram* [Jaco95] [Rumb94] has been proposed by numerous authors in
various forms to show the sequence of messages between model objects which are
required to perform a higher level process function. Figure 11 shows the major objects in
the design model along the topmost row and the main inter-object messages as heavy
arrows. The diagram is read top to bottom, and the ordered sequence of messages enacts
the functionality, sometimes termed a ‘transaction’, shown in bold print, on the left side
of the diagram. The curved arrow notation in Figure 11 indicates when a complex
*Smart Object Method* consisting of a large number of rules, and performing multiple sub-functions has been invoked.

Figure 11: Object Interaction Diagram for Principal Model Activities

6. Application of the model to the example problem scenario

To demonstrate the behavior of the design model in a real-world situation, we will walk through the process of maintaining trigger based coordination between the systems of the example developed in Section 1. We assume the lexicon and process descriptions have
been populated in a prior analysis [Kuec98]. At each step, the interaction between model entities will be noted, with reference to Figure 11.

1. A request to perform a security_clearance process, and the definition of that process at the aircraft manufacturer (the external process definition), is received at the military site, along with the applicant data (folder). The operating system (WFMS) external to the model sends a create() message to the Work_Request class causing a work request object to be created and initialized. This initiates operation of the model. As in all Smart Object method calls, the Monitor (control knowledge section of a Smart Object) of Work_request intercepts the call to determine if it should override or modify the request. There is no need to do so in this instance. Henceforward in this discussion, we will ignore Monitor processing unless it takes some exceptional action.

In Figure 11, action (1) above is represented by arrow 1 in the upper right corner from WFMS to Work_request. As part of its create() logic, Work_request sends a message to the class Work_description to create an instance for the external process (External WD) sent from the manufacturing site (arrow 2 in Figure 11). As part of its creation logic, External_WD, detects a trigger activity, and sends a create() message to class Coordinating_activity to establish the instance Preliminary Clearance Complete (arrow 3 in Figure 11).
Proceeding further with initialization processing, \textbf{Work\_request} performs the following actions:

2. A search is made of the Lexicon for a \textit{local} process equivalent for the same task.

3. A local process is found, and, it is taken as the base (overriding) definition. The semantic information in the \textit{external} process is examined to determine whether a full substitution (\textit{local for external}) can be accomplished, or whether non-deletable activities from the external process definition will need to be scheduled with the activities of the local definition.

4. The only non-deletable task in the external definition is the trigger activity, \texttt{preliminary\_clearance\_completed}. This activity must be meaningfully scheduled with the local (military site) activities in order for coordination between the two systems to be preserved.

Activities 1 through 4 above are generated by the \textbf{Work\_request Method Section} fragment shown in Figure 12. Note that the \textbf{Work\_request} methods do not explicitly invoke the internal \textbf{Work\_request} method \texttt{repository\_read()}. Rather, \texttt{repository\_read()} is an attached method that has been defined in the \texttt{Attributes Section} of the \textbf{Work\_request} Smart Object, and is invoked when the state attribute \texttt{local\_definition\_for\_process} is first accessed. That attribute is used in rules R1 and R3, however \texttt{repository\_read()} is invoked only on the \textit{initial} access of the attribute.
repository_read() is shown as arrow 4 in Figure 11. On successfully retrieving a local definition for a security_clearance, repository_read() issues a message to class Work_definition to create() the instance Local_WD (arrow 5 in Figure 11).

**Method:** Initialize

R1: If external_process_request
    and local_definition_for_process
    and external_WD.non_deletable_activities
    then DD.node.compare(external_WD, local_WD)

R2: If reconcile_local_process_changes
    then DD.node.compare(external_WD, local_WD)

R3: If external_process_request
    and not local_definition_for_process
    then DD.node.lex_compare(external_WD)

R4: If not external_process_request
    and not reconcile_local_process
    then suspend

**Figure 12:** Partial Methods Section for Work_request

Comparing the definitions for Local_WD and External_WD at a high level (Figures 13 and 14) it is apparent that the example corresponds to one of a small number of elementary process change scenarios that repeat across work cases⁴. In this particular

---

⁴ Five other elementary change scenarios and the behavior of the model in reaction to them is given in [KuecIP]
instance, multiple activities in one work definition have been substituted for others in a modified work definition. The substitute activities have been derived from different generalized functionalities than the activities they replace. However, the goal structure is identical for both definitions, as is common in a subcontracting situation, that is, both parties and WFMS sites agree on what is to be accomplished, though the subcontractor may unilaterally modify how it is done.

The coordinating trigger preliminary_clearance_complete had been scheduled to follow one of the activities that no longer exists in the current work definition. This requires preserving the semantics of the communication, which requires that it be scheduled to indicate an 'equivalent' process state. The first step in determining equivalence is to compare the two work definitions, Local_WD and External_WD. The consequent of R1 in Figure 12 sends the message DD_node_compare() to the Difference_data object, effectively asking it to "generate itself" from the two WDs passed to it (arrow 6 in Figure 11). DD_node_compare is another complex method, which uses procedural algorithms controlled by Method rules. When finished, DD_node_compare() has populated Difference_data with the following information:

- a subgoal node from which several functions and activities were derived now links to different generalized activities and functions
- that goal node (and the entire goal structure) is unchanged
- the immediate successor activity for the coordinating trigger is missing in the changed definition
• there are no other changes to the definition

**Difference_data** completes its activity and control returns to the Monitor (arrow 10 in Figure 11). Normally returns from activity completion are not shown in Object Interaction Diagrams, but we do so here for clarity, using dotted lines. Recall that the Monitor section of a Smart Object is where much of the control knowledge from the architectural model resides. The Monitor examines process state data in *Work_request* (message not shown in Figure 11) and determines that interpretation of differences between work definitions can now proceed. To accomplish this it sends the message *interpret()* to **Preliminary Clearance Complete** (arrow 11 in Figure 11).

The information that *new activities derived from a single unchanged goal constitute the only change* is used for control within the *interpretation/recognition* methods. One of several sets of heuristics is used for further inferencing, depending on *high level patterns* identified in the difference data. For example, if changes derived from *multiple* goals had been found, heuristics examining the changes for interactions between them would have been invoked. Recall from the design model discussion that most of the knowledge required for interpretation resides in the class *Coordinating_activity*, for which **Preliminary Clearance Complete** is the only instance in this example. Inferencing within the complex method *interpret()* proceeds as follows:

• **If** the trigger was attached to an internal activity in a sequence of activities that is now represented by a new activity sequence
- and if the goal for both sequences is the same,
- and if there is no domain-specific information (from the lexicon or domain
  specific rules) to the contrary,
- then the original trigger point (state) must correspond to some state of the new
  activity sequence

Since the goal of the trigger, indicated by an attribute, is temporal efficiency, control
rules direct inferencing to a rescheduling heuristic that gives priority to time. One such
heuristic is:

- change the predecessor based trigger to a timed trigger,
- set the start of the timer to the start of the activity sequence in Local_WD that is
  goal-equivalent to the activity sequence it replaces in External_WD, and
- set the duration of the timer to the duration of the task(s) that preceded the trigger
  in the original activity sequence (so that it fires at the same time after the start of
  the new activity sequence as it did after the start of the original sequence)

The activity above is represented in Figure 11 as the series of reads() and writes() to the
objects Difference_data, Local_WD and External_WD (arrow(s) 12). Once it has
introduced the timed trigger into the Local_WD activity sequence, and set its parameters,
control passes from Preliminary_clearance_complete to the Monitor. Monitor logic
again uses Work_request state information to determine the next activity to initiate. It
invokes scheduling of non-coordination work activities first, by sending the message
schedule_constraints() to Local_WD (arrow 14 in Figure 11) and then causes the
coordinating activity to be scheduled by sending the message \texttt{schedule\_triggers()} to \texttt{Preliminary\_Clearance\_Complete}.

Note that trigger scheduling was constrained by attachment to an activity sequence by \texttt{interpret()}). The dependence of scheduling on interpretation is inevitable; determining what events mean in many cases significantly effects their placement in a workflow. We will assume parameters for the activities in \texttt{Local\_WD} were such that no scheduling conflicts were identified. When both scheduling activities are complete, control returns to the \texttt{Monitor}, which then sends the message \texttt{evaluate\_schedule()} to \texttt{Work\_request} (arrow 17 in Figure 11). \texttt{evaluate\_schedule()} determines that no further action is necessary, and returns control to the WFMS which can now use the activity portion of \texttt{Local\_WD} as its workflow definition for the actual dispatching of activities to appropriate resources.

The example just completed is actually more complex than many common change situations in actual work environments, such as elementary activity name changes or activity parameter changes within specified limits, that are also easily handled by the model. [KuecIP] contains detailed descriptions of coordination maintenance for a basis set of elementary change scenarios, and the composition of these into more complex situations.

\section{Conclusions and future work}

This paper has presented an approach to preserving coordinating triggers between
automated workflows in distributed, autonomous systems. We have also presented the
model of coordination in interorganizational workflows on which the approach was
based, and have mapped the concepts to a more detailed design model to demonstrate
feasibility. By using Smart Objects to represent semantic knowledge, we are able to:

(1) construct a process model that includes work domain semantics, and that spans the
abstract problem domain and the concrete activity domain. This permits high level
reasoning about activity sequences.

(2) model triggers as first class entities independent of other events which allows them
to be reasoned about and temporally shifted in a “reasonable” manner when
required.

A WFMS able to incorporate semantics into the work definitions it processes, and reason
using this information is more robust in dynamic situations than one that uses closed
specifications. Additionally, work specifications which include goal information are more
easily understood and changed during manual redesigns of workflow, since the purposes
behind the activities are explicitly encoded with the activities [Yu95].

Our current work is focused on completing the model prototype using SOL (Smart Object
Language) as a design language, and KAPPA-PC for the implementation language

We envision expanding the model in two areas to enhance its ability to accurately capture
more complex environments and process changes:
• investigation of more sophisticated knowledge structures, such as multi-leveled function abstractions [Tene86] and interleaved goal hierarchies

• use of a more sophisticated lexicon that includes fundamental shared-world-view predicates for expanded common sense reasoning about coordination semantics (similar to the CYC project)

In the longer term, we hope the conceptual model will provide grounding for empirical exploration of the use of WFMS in new and evolving organizational forms, such as global software development organizations, and virtual manufacturing corporations [Olea97].
References


