Specifying and Constructing Schedulers for Workflows with Autonomous Executions*

P. Jensen† C. Wallace‡ N. Soparkar

Electrical Engineering & Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
USA

{pjensen,wallace,soparkar}@eecs.umich.edu

Abstract

Workflow has become an important paradigm for distributed data and computing systems in a wide range of application areas. In a workflow, tasks executing on autonomous, heterogeneous systems are coordinated through data and control flow constraints. An important challenge in workflow management is the scheduling of actions and operations performed by the concurrently executing tasks. The legal interleavings among the tasks must be specified, and scheduling control mechanisms to ensure correct, efficient executions must be generated. Scheduling workflows is particularly difficult because the dependencies between tasks may be application-specific and task autonomy may place certain actions outside the control or observation of the scheduler.

We use techniques from supervisory control theory of discrete event systems for specifying and generating scheduling controllers in workflow environments. We specify the tasks and the intertask dependencies as finite state automata. To model task autonomy, we characterize some of the event transitions in the task automata as beyond the control or observation of the workflow scheduler. By applying the techniques of supervisory control theory to these specifications, we show how the existence of schedulers may be ascertained and how schedulers may be constructed. In cases where no controller can allow exactly the desired class of schedules, we show how to construct a scheduler that allows the best possible approximation to the desired class. We also address the issues of prioritized tasks and distributed workflow scheduling. Our approach provides an effective means to model several workflow systems and to create scheduling mechanisms to manage them.

Keywords

Interoperability; Autonomous Systems; Distributed Databases; Workflows

*Part of this work appears in the ATMA '96 Workshop.

†Supported in part by IBM grant #302023.

‡Supported in part by NSF grant CCR-92-04742.

1 Introduction

Workflow systems have gained prominence in recent years for applications in business processes, hospital administration, collaboration technologies, and manufacturing control, among other areas. Workflows are characterized by tasks which access and manipulate shared data resources, with application-specific intertask dependencies. These tasks may run on autonomous, heterogeneous platforms, involve interactions outside the computing system, and demand diverse correctness conditions for their interleaved executions. In this paper, we examine a challenging problem in workflow, the correct and efficient scheduling of task actions — some of which may execute autonomously.

As an example, consider a computing system to assist hospital management and administration. A workflow designed for such a system may consist of several tasks such as entering the patient data into a database, obtaining information on previous visits and medical history, ascertaining insurance information, entering diagnostic information by a medical examiner, prescribing treatment medicine, assessing costs and billing the patient. There may be precedence constraints (e.g., initiate prescribing only after diagnostics are entered) and potential data conflicts (e.g., check the prescribed treatment against allergies in the patient's medical history) that proscribe certain interleavings of task actions. Each task itself may consist of several actions which are ordered within the task. Furthermore, these tasks may execute on different platforms and may have autonomous execution characteristics (i.e., some of the actions within each task may be outside the control of the workflow manager once the task is initiated). Data may be concurrently shared among some of the tasks that are permitted to execute at the same time (e.g., the tasks pertaining to medical history and insurance).

The above description indicates that not only are there intratask orderings (many of which may be handled autonomously) but also complex intertask dependencies. Logical dependencies arise from the access to shared data resources, and performance constraints arise from the desire to impose the fewest restrictions on the autonomous executions. To compound the problem, some of the actions within the individual tasks may be uncontrollable, or even unobservable, by an external workflow scheduler. We restrict attention to constraints that can be stated mathematically within the framework adopted by us.

Our approach to the problem of workflow scheduling is developed as follows. First, we model workflow systems as discrete event systems. Each task in a workflow may be regarded as a set of discrete events ordered to execute in a pre-specified manner, and the discrete event systems of the individual tasks is combined to model the system as a whole. Second, we use techniques from supervisory control theory [RW89] to obtain correct and efficient schedulers that manage the workflow system. Our approach, though far from a panacea to all workflow problems, is a step toward understanding and solving several of the difficulties described above. We are able to state unequivocally what is possible, and how to achieve the possible, within the purview of our model. As such, our approach is more formal and mathematical in its treatment of workflows, and appears to better suited to handle aspects of autonomy that arise in workflows.

Note that the scheduling control developed by us could be applied to workflow management "on-line" or "off-line" depending on the manner in which the scheduling *mechanisms* are used. In a sense, our control specifications relate to scheduling mechanisms in a manner similar to the way serializability theory relates to the concurrency control protocols. In the discussions to follow, we use terms such as "supervisor," "controller," and "scheduler," almost interchangeably: the terms

relate to the particular model or formalism in use, and for the purposes of the ideas presented in this paper, their differences may be ignored. Similar comments apply to a few other terms that we use, and their meaning is apparent from the context.

2 Related Work

The increased interest in workflow management (e.g., see [GHS95, KR95, AAA+96]) has resulted in considerable work on scheduling for workflow, and [RS94] provides an overview of various techniques. As in our approach, the results available are tied to the model adopted and the assumptions made.

Much of the research reported for workflow scheduling has developed from multidatabase transaction scheduling (e.g., see [SKS91, GRS91]). In consequence, there are several efforts that deal with issues of commitment of autonomously executed transactions and tasks. Indeed, some of the examples in this paper reflect this trend. While important and interesting, such research touches on just one aspect of several for workflow systems. For instance, issues of controllability and observability (i.e., concepts that arise from the autonomy of constituent systems, as explained in the paper) that are examined in this paper are not handled by such efforts.

Our work is more closely related to [ASSR93, ST94, Kle91, Gün93] in that we model tasks in a workflow system as automata. For example, finite state automata are constructed in [ASSR93] to represent dependencies in a manner similar to ours. However, the concept of supervisory control, including controllability and observability, are not addressed. Similar is the case for [ST94] in which the methods for specifying dependencies and scheduling is based on temporal logic. Our methods, though based on formalisms, are different in that we adapt the well-understood techniques of discrete event systems. In doing so, we are able to leverage several existing results, on-going research, and tools for the study and realization of workflow scheduling.

We believe that our approach to workflow management constitutes a paradigm shift. Other efforts have focused on a workflow scheduler which is submitted various actions, and thereafter, schedules are generated. In contrast, our technique may be likened to a least restrictive approach. The workflow environment is regarded as a set of spontaneously executing tasks some whose actions are controlled or observed by the workflow scheduler. Although ultimately these two views may merge, our technique does provide an alternative, possibly better, approach to modeling and reasoning about autonomous executions in workflow environments.

3 Workflows and Discrete Event Systems

In this section we describe workflows, introduce an illustrative example, and provide the basic concepts from discrete event systems [RW89].

3.1 Describing Workflows

A *workflow* is an organized set of *tasks*. Tasks are semantically coherent units of work which may be executed on diverse, heterogeneous platforms. Each task consists of *events* to be executed in a predefined order. An event is an action to be carried out on the system; for instance, a task initiation or termination, or a data access or update. Event notifications are sent to the workflow *scheduler*, which controls task executions by selectively allowing and preventing events from occurring. A

workflow scheduler is therefore passive in nature, allowing tasks to choose the events they execute but limiting their range of choice. Some tasks may involve events that are outside the control of the workflow scheduler. If a scheduler is notified of such an event, it cannot prevent it from occurring.

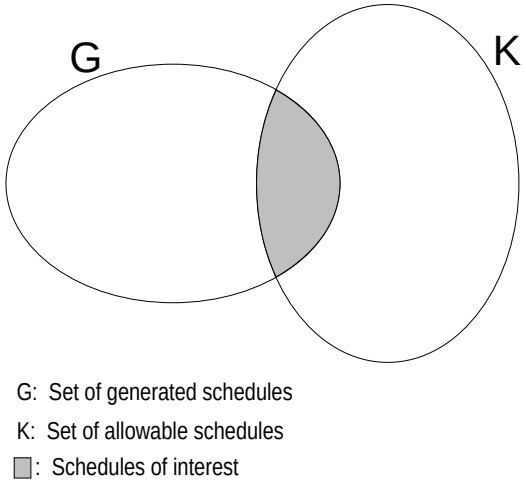

Figure 1: Objective in basic workflow scheduling.

In Figure 1, we depict the relationship between the tasks and the scheduler. A workflow organizes its tasks by establishing relationships between events of different tasks. These *intertask dependencies*, as for the tasks themselves, are defined by the workflow designer. A scheduler must ensure that a *schedule*, or sequence of events, is *legal*, (i.e., a schedule that satisfies both the task specifications and the intertask dependencies). In the figure, the tasks generate the set G of schedules, while K represents the set of schedules allowed by the intertask dependencies. The schedules of interest are the legal schedules, the intersection of G and K.

Consider a simple workflow $TRANS$, illustrated in Figure 2. $TRANS$ transfers funds between bank accounts by debiting one account and crediting the other. The workflow contains a task *debit* which triggers a task *credit*. Both tasks involve a start event, a termination event (either commit or abort), and an intervening precommit event. Each task is required to be *atomic*: it must either execute to completion or not execute at all. We assume that the failure of *credit* is tolerated but not that of *debit*; no work should be committed if *debit* fails. Hence the task *credit* can complete successfully only if *debit* completes successfully.

There are two intertask dependencies in this workflow. First, if *debit* is to start work, *credit* must also start, with *debit* preceding *credit*. This *trigger dependency* involves both a co-occurrence condition (i.e., if *debit* starts, then *credit* starts) and a temporal condition (i.e., if *credit* and *debit* both start, then *debit* starts before *credit*). Second, it must be ensured that *debit* has completed or will complete successfully before *credit* is allowed to do so. This *commit dependency* is a co-occurrence condition (i.e., if *credit* commits, then *debit* must get committed at some point).

Let s_c and c_c represent the start and commit events of *credit*, and let s_d and c_d represent the start and commit events of *debit*. We define the relations *trigger* and *commit* which hold between events. Thereafter, the dependencies may be expressed as $(s_d \; trigger \; s_c) \wedge (c_c \; commit \; c_d)$.

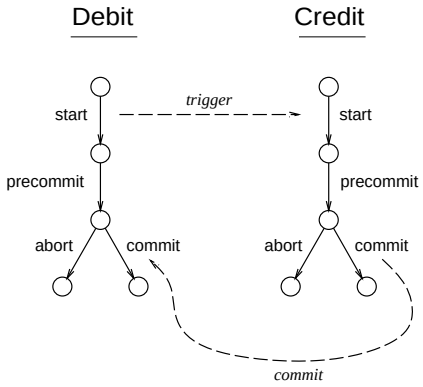

Figure 2: The $TRANS$ workflow.

3.2 Discrete Event Systems

A *discrete event system* (*DES*) is a dynamic systems in which state changes are caused by instantaneous occurrences of events. Workflows are modeled as DESs with the scheduler receiving event notifications in a discrete manner. Sequences of event occurrences are represented as strings, and a set of schedules generated by a scheduler is represented as a language.

We are interested in controlling the sequences of events that the system generates. The *event set* Σ of the system is the set of all event labels, and Σ^* represents the set of all finite strings over Σ including the empty string ϵ. A *language* over Σ is simply a subset of Σ^*. Languages are used to represent the set of all event sequences that can be generated and the set of all legal sequences. If a string w is a legal event sequence, then all event sub-sequences generated before w are also legal. A string u is a *prefix* of a string $v \in \Sigma^*$ if for some $w \in \Sigma^*$, $v = uw$. The *prefix closure* $\overline{L}$ of $L \subseteq \Sigma^*$ is the set of all prefixes of strings in L: $\{u : (\exists v \in \Sigma^*)(uv \in L)\}$. If $\overline{L} = L$ then L is *prefix-closed*. The behavior of a DES is modeled as a prefix-closed language L over an event set Σ.

We represent the behavior of a workflow in terms of a language *generator* G. G is a *finite state automaton* (*FSA*) $(Q, \Sigma, \delta, i, M)$ consisting of a finite state set Q, a finite event set Σ, a partial transition function $\delta : Q \times \Sigma \rightarrow Q$, an initial state $i \in Q$, and a set of marked states $M \subseteq Q$. The system behavior is captured by considering all sequences of transitions that start from the initial state. We extend the function δ to sequences of events: if $q \in Q$, then $\delta(q, \epsilon) = q$, and for all $w \in \Sigma^*, \sigma \in \Sigma$, $\delta(q, w\sigma) = \delta(\delta(q, w), \sigma)$. We define the *behavior of* G to be $L(G) = \{w \in \Sigma^* : \delta(i, w)\ defined\}$.

The generator's marked states M represent states of satisfactory completion. We define the *marked behavior of* G to be $L_M(G) = \{w \in \Sigma^* : \delta(i, w) \in M\}$. While $\overline{L_M(G)} \subseteq L(G)$ always, it is also desirable that $\overline{L_M(G)} = L(G)$ (i.e., every sequence generated by G can be extended to a state of satisfactory completion). When this is the case, we say that G is *non-blocking*.

Given generators A and B, we represent the concurrent execution of the generators by the *shuffle product* $G = A \parallel B$ of these generators. The states of G consist of pairs of states $A \times B$, the event set is $\Sigma_A \cup \Sigma_B$, the initial state is the pair (i_A, i_B), and the marked states consist of all

pairs $\{(s,t) : s \in M_A \wedge t \in M_B\}$. The transition function is defined as follows:

$$\delta_G((q,r),\sigma) = \begin{cases} (\delta_A(q,\sigma), \delta_B(r,\sigma)) & \text{if } \delta_A(q,\sigma) \text{ and } \delta_B(r,\sigma) \text{ are defined;} \\ (\delta_A(q,\sigma), r) & \text{if } \delta_A(q,\sigma) \text{ defined and } \delta_B(r,\sigma) \text{ undefined;} \\ (q, \delta_B(r,\sigma)) & \text{if } \delta_A(q,\sigma) \text{ undefined and } \delta_B(r,\sigma) \text{ defined} \end{cases}$$

3.3 Modeling Workflows

Each task in a workflow specifies dependencies between its events, and the workflow adds to these with a set of dependencies that hold between events of different tasks. We define a workflow $\mathcal{W}$ to be a pair $(\mathcal{T}, \mathcal{D})$, consisting of a set $\mathcal{T}$ of tasks (each specifying a set of intratask dependencies) and a set $\mathcal{D}$ of intertask dependencies. Both tasks and intertask dependencies are modeled as DESs.

Each task T in $\mathcal{T}$ is an FSA $(Q_T, \Sigma_T, \delta_T, i_T, M_T)$. Q_T is a finite set of states, Σ_T is a finite event set, δ_T is a partial function $Q_T \times \Sigma_T \rightarrow Q_T$, $i_T \in Q_T$, and $M_T \subseteq Q_T$. The event sets of each task are disjoint: $(\forall T, U \in \mathcal{T}, T \neq U)(\Sigma_T \cap \Sigma_U = \emptyset)$. We model the "uncontrolled" workflow by $\|_{T \in \mathcal{T}} T$, the product of all the tasks in the workflow. We assume that the workflow scheduler can distinguish events in a task, so that each event occurrence is unique within a task: $(\forall q, r \in Q_T)(\forall \sigma \in \Sigma_T)[(\delta_T(q,\sigma)\ defined\ \wedge \delta_T(r,\sigma)\ defined) \rightarrow (q = r)]$.

Each task begins with a start event s and terminates with a commit event c or an abort event a. In addition there is a precommit event p that precedes termination. As each task is atomic, it must run to completion or not at all. Therefore the initial states and the states following termination are the marked states. The tasks *credit* and *debit* are modeled as separate automata G_c and G_d and as a single automaton $G_{cd} = G_c \parallel G_d$ in Figure 3. Note that while this may suggest tasks as having only transactional semantics, the model can be extended for non-transactional procedures.

Each dependency D in $\mathcal{D}$ is a DES, and in particular, an FSA. A dependency FSA has no private event set; rather, its event set consists of the union of event sets of the task automata. Let $\Sigma = \bigcup_{T \in \mathcal{T}} \Sigma_T$; then D is an FSA $(Q_D, \Sigma, \delta_D, i_D, M_D)$, where Q_D is finite, δ_D is a partial function $Q_D \times \Sigma \rightarrow Q_D$, $i_D \in Q_D$, and $M_D \subseteq Q_D$. The $TRANS$ workflow intertask dependencies may be represented by the automata D_s and D_c as shown in Figure 4.

4 Workflows under Supervisory Control

In this section, we present the supervisory control theory from [RW89] needed to frame workflow scheduling as a supervisory control problem. We illustrate our technique for developing schedulers with the $TRANS$ example of Section 3.1.

4.1 Supervisory Control Theory

We now discuss how a controller or *supervisor* enforces correctness conditions on DESs by a feedback loop between the generator and the supervisor. At each event occurrence, the supervisor controls the set of possible events by *disabling* certain events. The event set Σ is partitioned into a set Σ_c of *controllable* events (those which can be disabled) and a set Σ_{uc} of *uncontrollable* events (which corresponds to autonomous, uncontrollable task actions). As illustrated in Figure 5, when an event

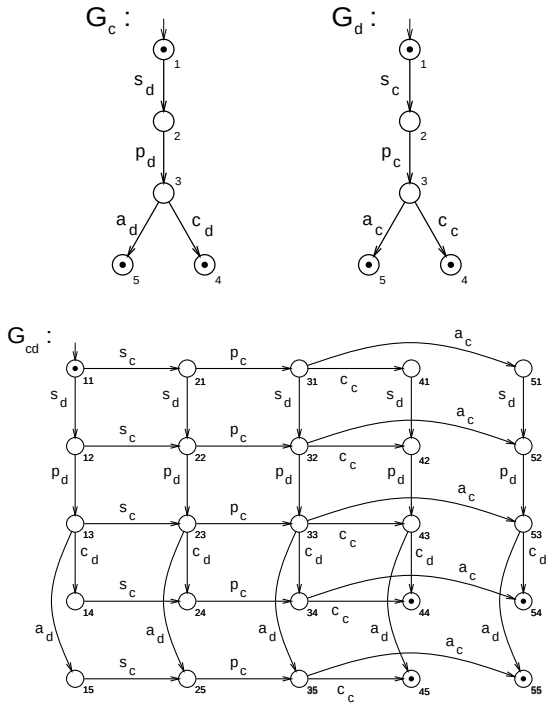

Figure 3: Tasks *credit* and *debit* modeled in terms of FSAs.

σ occurs, the generator G moves from its current state q to state $\delta_G(q,\sigma)$. After each event supplied by G, the supervisor sends a set of events γ, the *control input*, to G. These are the events *enabled* in G's new state $\delta_G(q,\sigma)$. Uncontrollable events are always enabled: $\Sigma_u \subseteq \gamma$. The events generated by G are constrained by the control input; each event σ sent from G to the supervisor is chosen from the current value of γ. We model the supervisor as a DES, and in particular as an FSA S. The supervisor's control input is then determined by the transition structure of S. The supervisor and generator run in parallel. If G is in state q and S is in state r, then event σ is enabled if and only if $\delta_G(q,\sigma)$ and $\delta_S(r,\sigma)$ are defined. If both are defined, then G moves to state $\delta_G(q,\sigma)$ and S moves to state $\delta_S(r,\sigma)$.

It is important to characterize the languages which can be realized by a supervisory controller. If all events are controllable, the control problem is trivial, and any language generated by an FSA (*i.e.*, any regular language) can be realized; at each state, the supervisor can disable the events for which its automaton has no transition defined. For a language K to be *controllable*, each uncontrollable event must lead to a path from which some sequence in K can be attained. In particular, K is controllable if $\overline{K}\Sigma_u \cap L(G) \subseteq \overline{K}$; for any prefix of a sequence in K, if an uncontrollable event is added, the result is still a prefix of a sequence in K. Below, we use these concepts to develop schedulers which generate only legal sequences.

For workflow scheduling, we seek a solution that is not only controllable but also non-blocking To characterize the conditions under which a non-blocking solution is possible, we introduce the notion of $L_M(G)$ *closure*. A language $K \subseteq L_M(G)$ is $L_M(G)$-closed if $\overline{K} \cap L_M(G) = K$. To attain $L_M(G)$ closure, all the strings in K must be in $L_M(G)$, and that any marked prefix of a string in

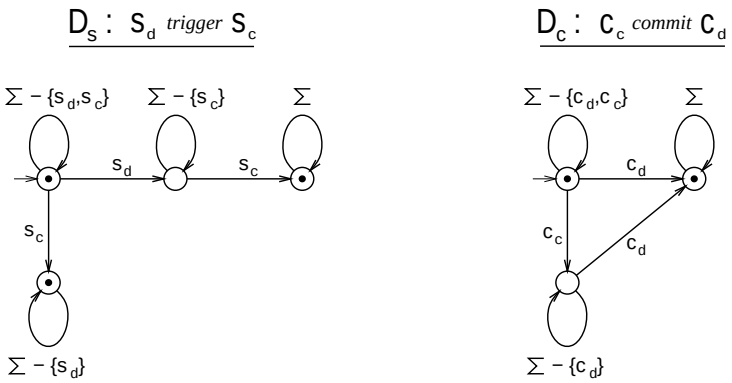

Figure 4: Automata representing the intertask dependencies of $TRANS$.

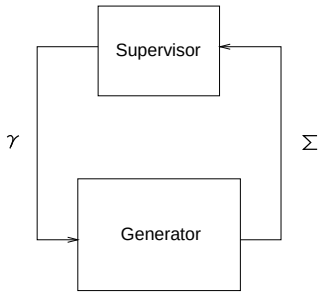

Figure 5: Feedback loop of control.

K is itself in K. To avoid blocking, we must not exclude any marked prefix that leads to a desired string in K, else there will be no way to attain the desired string.

The results for controllability, taken from [RW87b], are summarized as follows. Fix a generator G that generates language $L(G)$ and marked language $L_M(G)$.

- For nonempty $K \subseteq L(G)$ there exists a supervisor S realizing K if and only if K is prefix-closed and controllable.
- For nonempty $K \subseteq L_M(G)$ there exists a supervisor S realizing K as its marked language, and the system is non-blocking if and only if K is controllable and $L_M(G)$-closed.

4.2 Pragmatism in Obtaining Controllers

If a given language K is found to be uncontrollable, we would like to be able to find a controllable language K that is as close an approximation to K as possible. In workflow scheduling, this implies finding a controller that allows as many legal schedules as possible while not allowing any illegal schedules. The class of controllable sublanguages of K is closed under set union and has a unique supremal element $K^{\uparrow}$ under set inclusion [WR88b]. We call $K^{\uparrow}$ the *supremal controllable sublanguage* of K, and we characterize a controller realizing $K^{\uparrow}$ as a *supremal controller* for K. If, as in workflow scheduling, dependencies implicit in K are never violated, then $K^{\uparrow}$ is the best solution possible. Furthermore, if we model the generator and supervisor as FSAs, an efficient algorithm to compute $K^{\uparrow}$ is available.

Scheduling workflows involves enforcing a set of intertask dependencies simultaneously. If each dependency can be expressed in terms of an FSA, a supervisor can be achieved by taking the product of all the dependency automata. However, the result may suffer from an exponential state space increase. To counter this, additional structure could be introduced into the model, and the supervisor can be modularized into a set of independent dependency enforcers. The dependency automata in D are run in parallel, and the control input is the intersection of the control inputs of all the automata. Whether this can be done depends on whether the languages defined by the dependencies are *non-conflicting*. Languages K_1 and K_2 are non-conflicting if $\overline{K_1 \cap K_2} = \overline{K_1} \cap \overline{K_2}$; for every prefix that K_1 and K_2 have in common, there is a word they have in common that contains this prefix. Consider $K_1^{\uparrow}$ and $K_2^{\uparrow}$, which in the best case are equal to K_1 and K_2. For non-conflicting languages, the $\uparrow$ operator commutes with the intersection operator. That is, a supremal supervisor generating $(K_1 \cap K_2)^{\uparrow}$ can be obtained efficiently, effecting $(K_1^{\uparrow} \cap K_2^{\uparrow})$.

In short [RW87a, WR88a]:

- If K_1 and K_2 are non-conflicting, $L_M(G)$-closed and controllable, then $K_1 \cap K_2$ is $L_M(G)$-closed and controllable.
- If $K_1^{\uparrow}$ and $K_2^{\uparrow}$ are non-conflicting, then $K_1^{\uparrow} \cap K_2^{\uparrow} = (K_1 \cap K_2)^{\uparrow}$.

4.3 Illustrating Supervisory Control

We now illustrate our approach by applying supervisory control theory to the $TRANS$ workflow. We define our specification language of correct schedules and test it for controllability and blocking. We then determine the controllable, non-blocking language that includes the greatest number of legal schedules and construct a generator for this language. Finally, we test whether the supremal controllable sublanguage can be attained using a modular control approach.

From the specifications of the generator G and the dependencies D_s and D_c we define the legal language K of the workflow system. K must be a subset of each of the languages $L(D_s)$ and $L(D_c)$ generated by the dependency automata, and it should be a subset of the language $L(G)$. The desired language is $K = L(G) \cap L(D_s) \cap L(D_c)$, and the desired marked language is $K_M = L_M(G) \cap L_M(D_s) \cap L_M(D_c)$. The shuffle product $G \parallel D_s \parallel D_c$, generating K with marked language K_M, is illustrated in Figure 6.

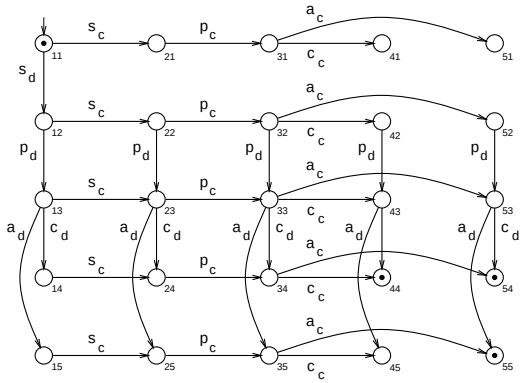

Figure 6: Generator for specification language K.

We use the results presented above to determine whether K_M is controllable. We can show that K_M is not controllable (*i.e.*, $\overline{K_M}\Sigma_u \cap L(G) \not\subseteq \overline{K_M}$) by the following counterexample. Consider the string $k = s_1 s_2 p_1 p_2 c_2 \in \overline{K_M}$. If we extend this string by adding the event $a_1 \in \Sigma_u$, we get the string $s_1 s_2 p_1 p_2 c_2 a_2 \notin \overline{K_M}$. Intuitively, the string k represents a schedule in which the *credit* task is allowed to commit without the *debit* task having committed earlier. For the commit dependency to hold between *credit* and *debit*, the *debit* task must then commit. However, this cannot be guaranteed, as *debit* may abort instead (*e.g.*, due to failure of the site where *debit* is running), an event outside the control of the workflow scheduler.

Since $K_M^{\uparrow} \neq K_M$, a supremal supervisor will allow only a subset of K_M. Applying an algorithm in [WR88b], we construct a generator for $\overline{K_M^{\uparrow}}$ with marked language $K_M^{\uparrow}$. We start with an FSA H_0, a sub-automaton of G that generates $\overline{K}$ with marked language K_M. H_0 is the shuffle product $G \parallel D_s \parallel D_c$ shown in Figure 6. We remove all states that are unreachable to obtain H_1 as shown in Figure 7. We define an *uncontrollable transition* to be a pair (q, σ) where $\sigma \in \Sigma_u$ and $\delta(q, \sigma)$ is defined. Comparing the uncontrollable transitions of H_1 with those of G, if there is an uncontrollable transition (q, σ) in G which leads from a state q in H_1 to a state r not in H_1, we must remove q from H_1. One such transition, $(43, a_d)$, requires removal of state 43 and all transitions to and from it. We then remove state 42 to obtain H_2 as shown in Figure 8. Comparing H_2 with G, we find no uncontrollable transitions in G that lead from a state in H_2 to a state not in H_2. As this iteration leaves H_2 unchanged, a fixed point is reached, so the algorithm terminates. H_2 generates $\overline{K_M^{\uparrow}}$ with marked language $K_M^{\uparrow}$.

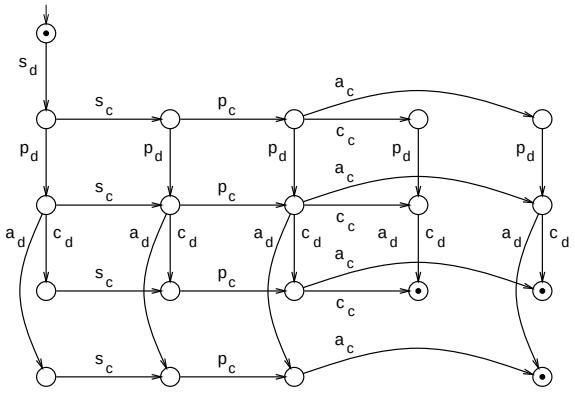

Figure 7: Generator H_1

Though the controller solution described above suffers little from a large state space, we test whether a modular solution exists. Let K_s and K_c be the languages generated by the dependency automata D_s and D_c. We can construct generators for $K_s^{\uparrow}$ and $K_d^{\uparrow}$, using the algorithm of [WR88b]; the generators are shown in Figure 9. The language resulting from the parallel use of the generators for $K_s^{\uparrow}$ and $K_c^{\uparrow}$ is $K_s^{\uparrow} \cap K_c^{\uparrow}$. Next, by inspection we see that $K_s^{\uparrow}$ and $K_c^{\uparrow}$ are non-conflicting, *i.e.*, $\overline{K_s \cap K_c} = \overline{K_s} \cap \overline{K_c}$. This implies that $K_s^{\uparrow} \cap K_c^{\uparrow} = (K_s \cap K_c)^{\uparrow}$, and that the modular approach produces the desired language K.

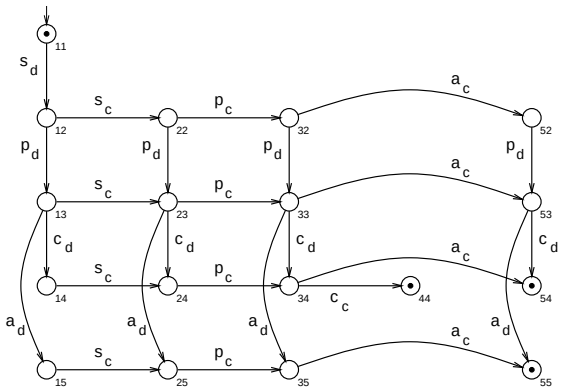

Figure 8: Generator for $\overline{K_M^{\uparrow}}$.

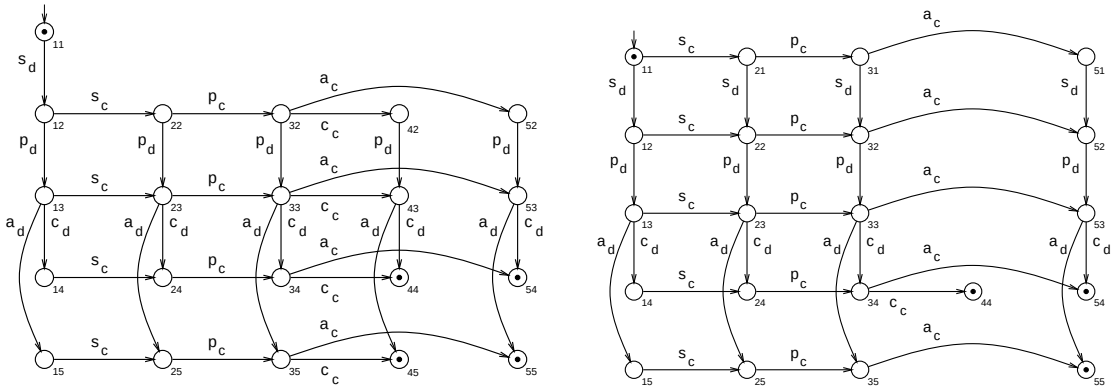

Figure 9: Generators for $K_s^{\uparrow}$ and $K_c^{\uparrow}$.

5 Extensions

To account for more complex aspects of workflow systems, we describe several extensions together with motivating examples. In particular, to account for autonomous distributed executions, we consider the issues of observability, incorporating priorities, distributed control, and limited relaxation of dependencies.

5.1 Observability

Not only are certain events in a workflow outside the control of the workflow scheduler, but also, certain events may occur without the scheduler's knowledge. For example, an application may have a limited interface with the scheduler, notifying it of only some of its actions. In such situations, a scheduler must be able to anticipate the occurrence of unseen events, using its control to generate legal schedules regardless of whether the unseen events occur. In this regard we incorporate the concept of *observability* into the supervisory control model [CDFV88].

The set of events Σ in an uncontrolled DES is partitioned into a set of *observable* events Σ_o and a set of *unobservable* events Σ_{uo}. For any schedule s generated by the concurrently executing tasks, the supervisor observes only the subsequence $P(s)$ (with all event occurrences in Σ_{uo} removed from

s) from the *projection*P. A projection is a mapping of event sequences of an alphabet (e.g., Σ) to event sequences of an alphabet that is a subset of the original (e.g., $\Sigma_o \subseteq \Sigma$), the mapping essentially removes events which do not exist in the range alphabet (Σ_o) from a given sequence. A model of the control loop, as shown in Figure 10, has a module P between the generator and supervisor, and P is a projection which sends only observable generated events to the supervisor.

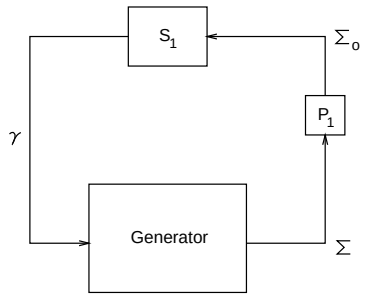

Figure 10: Feedback loop of control, incorporating observability.

We first determine whether a given specification language K is achievable by supervisory control in the presence of unobservable events. Clearly, if the occurrence of an unobservable event requires a change in the control input (*i.e.*, the enabling or disabling of events) in order to achieve K, then the supervisory control cannot achieve the language. This concept of *observability* suggests that a language K is observable if for all $s, s' \in \Sigma^*$, $(P(s) = P(s')) \rightarrow (\forall \sigma \in \Sigma_c)((s, \sigma, s') \in NextAct)$. The set $NextAct$ is defined as follows: for $s, s' \in Sigma^*, \sigma \in \Sigma$, $(s, \sigma, s') \in NextAct\ if\ (s\sigma \in \overline{K} \wedge s' \in \overline{K} \wedge s'\sigma \in L(G)) \rightarrow (s'\sigma \in \overline{K})$.

As an illustration, consider an example workflow consisting of two tasks New and Old. The former may be regarded as a new application that has a full interface with the workflow scheduler, while the latter is an older one with only a rudimentary interface. Though both applications update data items, only New informs the scheduler of its updates, though both inform the scheduler of termination. The interaction of these tasks involves accessing a shared variable x. New updates x and terminates, whereas Old either updates x or an unshared variable y, and terminates. The intertask dependency requires that a task updating x should terminate before the other updates x.

We model the tasks New and Old as FSAs, as shown in Figure 11. The event set of New consists of the update event w_x^N and the termination event t^N, and the event set of Old consists of the update events w_x^O and w_y^O and the termination events t_x^O and t_y^O. Assume that all events are controllable, but that w_x^N, w_x^O and w_y^O are unobservable. Our dependency is modeled by the FSA shown in Figure 12. If w_x^N occurs, then t^N must occur before w_x^O is allowed to occur. Likewise, if w_x^O occurs, then t^O must occur before w_x^N is allowed to occur.

Let K be the specification language arising from these generators, then K is shown to be unobservable by the following counterexample. Let $s = \epsilon$, $s' = w_x^O$, and $sigma = w_x^N$. $P(s) = P(s') = \epsilon$, and since $\sigma \in \Sigma_c$, for K to be observable it must be the case that $(s, \sigma, s') \in NextAct$. However, this is not the case; $s\sigma = w_x^N \in \overline{K}$, $s' = w_x^O \in \overline{K}$, and $s'\sigma = w_x^O w_x^N \in L(G)$, but $s'\sigma \notin \overline{K}$. The intuition behind this result is that the schedules s and s' are indistinguishable by the scheduler, yet require different control inputs. If no event has occurred (as in s), then no event need be disabled, but if w_x^O has occurred (as in s'), the event w_x^N must be disabled.

If a specification language happens to be unobservable, the options for computing adequate

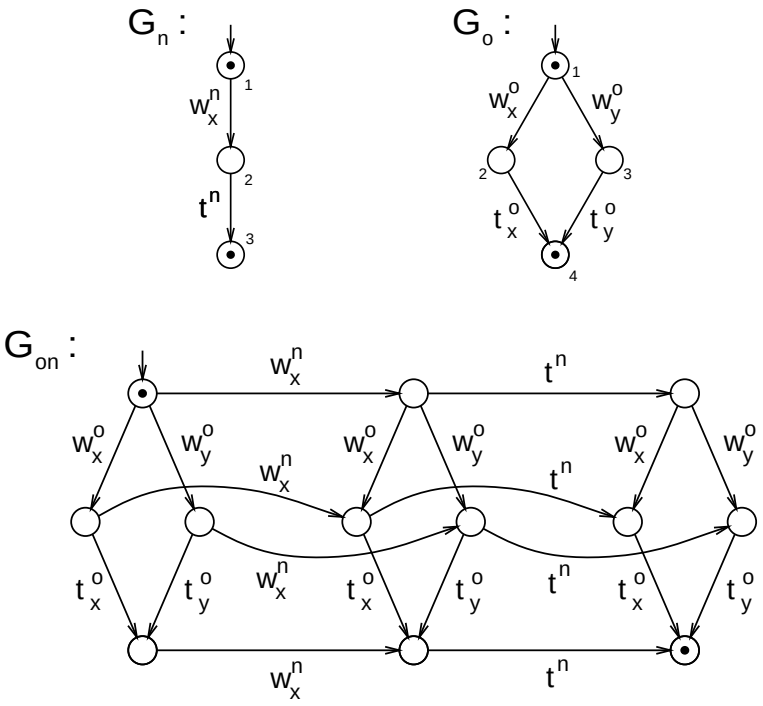

Figure 11: *Old* and *New* tasks modeled in terms of FSAs.

schedulers are fewer since the class of observable sublanguages of a language K is not closed under union. One way around this is to use a stronger property than observability that is closed under union which leads to a single solution. As another approach, special characteristics of a model can be exploited to determine the supremal observable sublanguage. A situation in which this is possible is when $\Sigma_c \subseteq \Sigma_o$, (i.e., when the controllable events are all observable). In such cases, any algorithm which computes a maximal solution is sufficient to determine the supremal observable sublanguage. Finally, it is possible to compute a single maximal sublanguage by assigning *priorities* to events during the computation, and by varying the priority assignment, different maximals may be obtained. In our example, giving priority to w_x^N over w_x^O will lead to a maximal solution involving the initial disabling of w_x^O.

5.2 Prioritized Events in Workflow

While allowing as many schedules as possible is a good general rule for obtaining satisfactory performance, it may be desirable to avoid certain legal schedules that incur a high performance cost. The schedules obtained by the minimally restrictive non-blocking scheduler should be further restricted, disallowing the excessively costly schedules (see Figure 13). A similar issue is the assignment of priorities to certain tasks or actions in the workflow. If the actions of concurrent tasks are assigned different priorities, a scheduler should attempt to schedule the actions of higher priority first.

We represent the performance criteria or priorities as cost functions on the set of events. We augment the workflow model with two cost functions $c_e : \Sigma \rightarrow \Re^+ \cup \{0, \infty\}$ and $c_c : \Sigma \rightarrow \Re^+ \cup \{0, \infty\}$, mapping events to the non-negative real numbers or infinity. The function c_e represents the cost

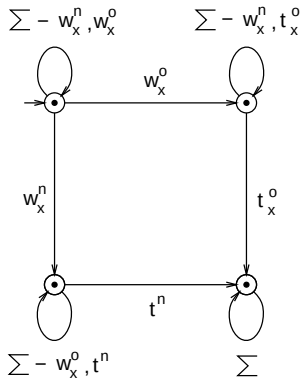

Figure 12: Automaton representing the intertask dependency between Old and New.

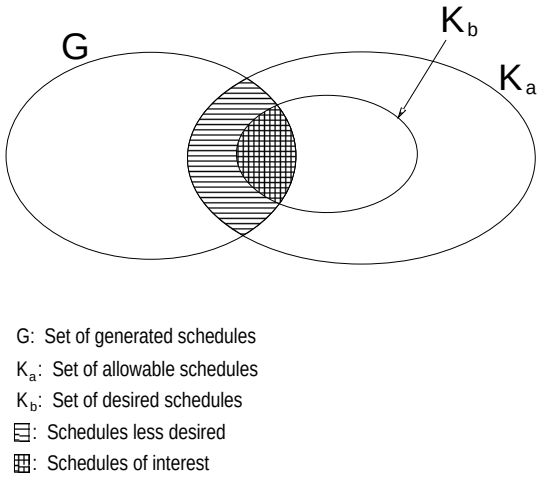

Figure 13: Objective in workflow scheduling with priorities.

of executing an event in the system, while the function c_c represents the cost of taking a control action on an event (*i.e.*, preventing it from occurring). We assign uncontrollable events a control cost of infinity. The cost functions are extended to strings and languages as follows. For a language A_m realized by a controller S, the cost for $s \in A_m$, the cost of string s is the sum of the costs of the events that comprise s together with the sum of the costs of control actions taken by S in achieving s. The cost of a language A_m is equal to the highest cost string in A_m: $c(A_m) = max_{s \in A_m} c(s)$.

We formulate the control problem as follows. A language $K_{om} \subseteq K_m$ is an optimal language if for all $A_m \subseteq K_m, c(K_m) \leq c(A_m)$. For a specification language $K_m \subset L_m(G)$, we consider a controller S such that the behavior under control $L_m(S/G)$ is non-blocking and optimal. Note that an optimal language with finite cost is always controllable. Therefore, an optimal language is a controllable sublanguage which allows the least costly worst case behavior. An efficient algorithm exists to realize an optimal scheduler (e.g., see [SL93a, SL93b]).

As an example, consider a case where a workflow has priorities assigned to certain tasks. One possible approach to model this situation is to simply assign low execution and high control costs for higher priority tasks, and high execution and low control costs for lower priority tasks. As illustrated in Figure 14, consider two tasks, H and L, accessing a common data item. Both H and

L read the data item, make some modifications, and update the data item. Task H is regarded to be of high priority, and task L of low priority. We assign all event costs to be 0, and control costs $c_c(r_x^H) = c_c(w_x^H) = 5$ and $c_c(r_x^L) = c_c(w_x^L) = 1$. In this case, the specification would require both tasks not to read data item x without one of them having updated the data item. The only optimal sublanguage of this specification, given the costs, is the language only allowing the schedule $r_x^H w_x^H r_x^L w_x^L$, which favors the high priority task.

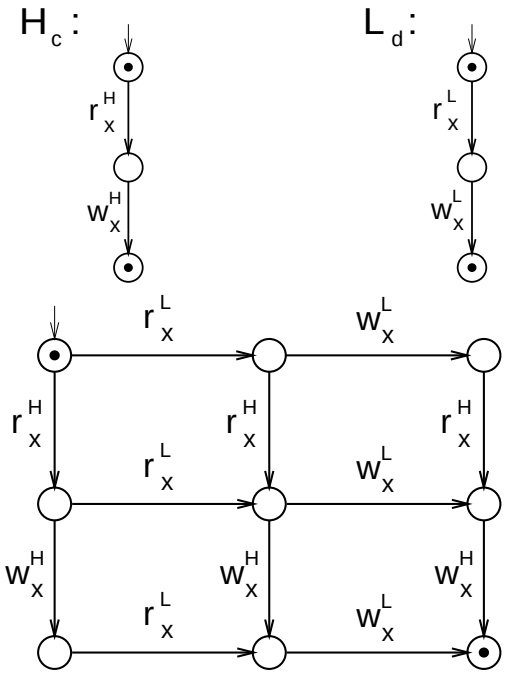

Figure 14: Prioritized tasks.

5.3 Distributed Control

Most current workflow schedulers are centralized, whereas the environments for workflow systems are often distributed. For such situations, efforts in distributed supervisory control of discrete event systems may apply to distributed workflow. As an example, consider n sites participating in a workflow with a local supervisor at each site responsible for scheduling local events. If each supervisor sees only a subset of the possible events in the system (e.g., some sites may be oblivious of events occurring remotely), then some events are not controllable by remote schedulers.

Figure 15 shows a model for distributed supervisory control [LW88] where, associated with each site i is a supervisor S_i and a projection P_i. Each supervisor S_i sees only a subset Σ_{oi} of the events in the system due to the projection P_i and has control over only the events Σ_{ci}. The control input γ_i is a subset of Σ such that $\Sigma - \Sigma_{ci} \subseteq \gamma_i \subseteq \Sigma$. In this model, we allow specifications to be made over the entire event set Σ, to account for intertask dependencies.

For generator G, prefix-closed $K \subseteq L_M(G)$, and alphabets $\Sigma_{o1}, \ldots \Sigma_{on} \subseteq \Sigma$ and $\Sigma_{c1}, \ldots \Sigma_{cn} \subseteq \Sigma$, we are to design non-blocking controllers $S_1, \ldots S_n$ such that $L(S_1 \wedge \ldots \wedge S_n/G) \subseteq K$. Here each Σ_{oi} and Σ_{ci} are the sets of events observable and controllable by supervisor S_i. K is the specification

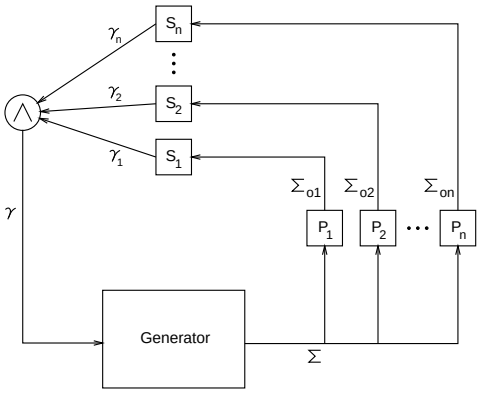

Figure 15: Feedback loop for distributed control.

language defining the constraints on the behavior of the system, and G is the generator of the uncontrolled language. The control input (*i.e.*, the events enabled under control of the supervisors) is the intersection of the control inputs of each of the separate supervisors.

For distributed control, existence results are available, with the concepts of controllability and co-observability, similar to the centralized case. There exist supervisors $S_1, \ldots S_n$ such that $L(S_1 \wedge \ldots \wedge S_2/G) = K$ if and only if K is controllable and co-observable [RW92]. An efficient algorithm presented in [RW93] determines whether controllers $S_1, \ldots S_n$ exist which satisfy the above conditions. When the specification K is either not controllable or not co-observable, this typically gives rise to computationally expensive problems.

5.4 Limited Relaxation of Constraints

Often, it may not be possible to obtain the exact set of schedules of a particular specification K. One way around this problem is a scheduler for the supremal controllable sublanguage $K^{\uparrow}$ which may adversely impact the efficiency of the system by prohibiting many legal schedules. As an alternative, we may consider a larger class of schedules that not only includes all legal schedules, but also includes some illegal schedules. In our context, this would correspond to a relaxation of the constraints, and constitute superlanguages in terms of set inclusion. Clearly, one would need to restrict the relaxation while allowing controllability.

To this end, we use a complementary operation to the supremal controllable sublanguage operation. The infimal prefix-closed and controllable superlanguage of K, $K^{\downarrow}$, is the unique controllable language which allows the fewest illegal schedules. Since the prefix-closed regular languages are closed under intersection, $K^{\downarrow}$ exists in general, and there is an efficient algorithm which can compute $K^{\downarrow}$. Interestingly, the class of prefix-closed controllable and observable languages is closed under intersection. Therefore, $K^{\downarrow}$ exists in general, and can be computed efficiently [CDFV88].

6 Conclusions

We have addressed the issues in scheduling autonomous tasks in a workflow environment. We provided a framework adapted from the well-understood domain of discrete event control systems theory. We are able to account for autonomous executions that might occur in integrated workflow

environments by allowing for the possibility of some task actions being uncontrollable or unobservable to the workflow management system. Furthermore, we have facilitated the imposition of application-specific intertask dependencies.

Our approach is based on discrete event automata, and it provides practical mechanisms for specifying dependencies among the tasks. Also, we addressed some of the scheduling issues that arise in distributed workflow systems. We are able to reason about schedulability given the dependencies, and to realize the requisite schedulers. With regard to efficiency, in the most relevant cases, we provide the provably best possible schedulers that use the degree of concurrency as a figure of merit.

There are several issues for our approach that need to be resolved. These include failure recovery, handling computational intractability problems that are often unavoidable with *any* similar approach, implementation techniques, etc. Several simple toolkits exist for the specification and manipulation of discrete event systems control, and we expect to use them in conjunction with our approach to assist in the effective development of workflow management systems.

Acknowledgments: We gratefully acknowledge the discussions we had with the several individuals. Most notably, Umesh Dayal, C. Mohan, Amit Sheth, and Dimitrios Georgakopolous explained to us some of the issues involved in workflow systems. Also, Stephane Lafortune and Raja Sengupta clarified the concepts in the supervisory control of discrete event systems.

References

[AAA+96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proceedings of the Twelfth International Conference on Data Engineering*, February 1996.

[ASSR93] P.C. Attie, M.P. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 134–145, 1993.

[CDFV88] R. Cieslak, C. Desclaux, A.S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, March 1988.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[GRS91] D. Georgakopolous, M. Rusinkiewicz, and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Seventh International Conference on Data Engineering*, 1991.

[Gün93] R. Günthör. Extended transaction processing based on dependency rules. In *Proceedings of RIDE-IMS '93*, pages 207–214, April 1993.

[Kle91] J. Klein. Advanced rule driven transaction management. In *Proceedings of the Thirty-sixth IEEE Computer Society International Conference*, pages 562–567, March 1991.

[KR95] M. Kamath and K. Ramamritham. Modeling, correctness and systems issues in supporting advanced database applications using workflow management systems. Technical Report 95-50, University of Massachusetts, 1995.

[LW88] F. Lin and W.M. Wonham. Decentralized supervisory control of discrete-event systems. *Information Sciences*, 44:199–224, 1988.

[RS94] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1994.

[RW87a] P.J. Ramadge and W.M. Wonham. Modular feedback logic for discrete event systems. *SIAM Journal of Control and Optimization*, 25(5):1202–1218, May 1987.

[RW87b] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, January 1987.

[RW89] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[RW92] K. Rudie and W.M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, November 1992.

[RW93] K. Rudie and J.C. Willems. The computational complexity of decentralized discrete-event control problems. Technical Report IMA Preprint Series 1105, University of Minnesota, 1993.

[SKS91] N. Soparkar, H.F. Korth, and A. Silberschatz. Failure-resilient transaction management in multidatabases. *IEEE Computer*, 24(12):28–36, December 1991.

[SL93a] R. Sengupta and S. Lafortune. A deterministic optimal control theory for discrete event systems: Formulation and existence theory. Technical Report CGR-93-7, University of Michigan, December 1993.

[SL93b] R. Sengupta and S. Lafortune. A deterministic optimal control theory for discrete event systems: Computational results. Technical Report CGR-93-16, University of Michigan, December 1993.

[ST94] M.P. Singh and C. Tomlinson. Workflow execution through distributed events. In *Proceedings of the Sixth International Conference on Management of Data*, 1994.

[WR88a] W.M. Wonham and P.J. Ramadge. Modular supervisor control of discrete event systems. *Mathematical Control, Signals, and Systems*, 1(1):13–30, January 1988.

[WR88b] W.M. Wonham and P.J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, May 1988.