In *Proceedings of the 6th International Conference on Management of Data (COMAD)*, 1994

Workflow Execution Through Distributed Events

Munindar P. Singh and Christine Tomlinson

Microelectronics and Computer Technology Corporation
Research Division
3500 West Balcones Center Drive
Austin, TX 78759, USA
(512) 338-3431
{msingh,tomlic}@mcc.com

Workflow Execution Through Distributed Events

Munindar P. Singh and Christine Tomlinson

Microelectronics and Computer Technology Corporation

Abstract

Workflows are the semantically appropriate units of activity in heterogeneous environments. Heterogeneous environments, which are extremely common in enterprises of even moderate complexity, involve a number of database systems, each with its own interfaces, applications, and users. It is typically not possible to violate the autonomy of these systems, even though it is essential to have them interoperate properly. Workflows are activities carried out in heterogeneous environments that impinge upon a number of autonomous systems. The semantics of workflows are best represented by different extended transaction models, or combinations thereof. We present an approach by which workflows can be efficiently scheduled in a distributed manner.

1 Introduction

Heterogeneous systems consist of a number of differently constructed applications and information resources that must interoperate coherently. Since they involve a variety of system and human activities, composite tasks in heterogeneous environments are best thought of as *workflows* [DGMH+93]. A number of extended transaction models [DHL91, Elm92] have been proposed recently to overcome the shortcomings of the traditional, ACID, model [GR93] in heterogeneous environments. These models relax the atomicity, isolation, and consistency requirements of the ACID model in various ways.

We have developed a general facility to specify and schedule intertask dependencies. Such dependencies can be used to formalize the scheduling aspects of a large variety of, and combinations of, workflow and transaction models. Our approach can express the primitives of [Kle91], which can capture those of [CR92] and [Gün93]. Our broad approach is similar to [ASSR93, Kle91], although it is more general—e.g., in handling tasks with arbitrary structure, not just loop-free ones.

This paper presents an approach by which workflows can be efficiently scheduled in a *distributed* manner. We have developed an algebra of events for representing and reasoning about intertask dependencies, which are a natural means by which to capture workflow requirements. Our algebra has special features for event scheduling; several technical semantic properties are needed to ensure that the algebra can be used for symbolic reasoning about event schedules. We described these details in a previous paper, and do not discuss them here. Instead we focus on the aspects of designing an automatically distributed event scheduler and highlight the intuitions and principles behind our (implemented) approach.

Current workflow products are centralized with a single server that stores and processes all relevant workflows. They operate in office environments with activities of fairly long time scales, typically involving humans. These products require procedural encoding of workflows and are designed for homogeneous PC environments. All decisions are made in the server. For these reasons, they cannot scale up easily.

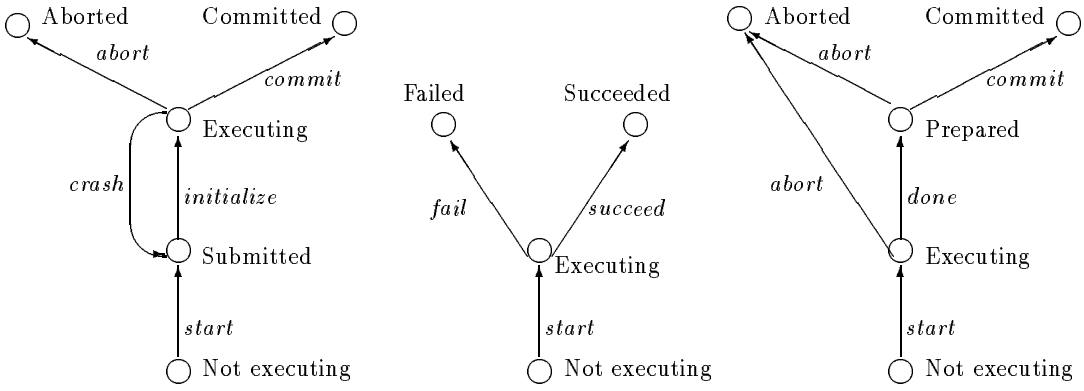


Figure 1: Some Common Task Agents [RS94]

Centralization is clearly undesirable in distributed, heterogeneous environments. Our approach associates an *agent* with each task or transaction. The agent embodies a coarse description of the task, including only states and transitions (or *events*) that are significant for coordination. Our approach applies to arbitrary agents and to any set of significant events. Figure 1 shows some example task agents. An IMS database transaction when submitted enters the executing state, from where it may proceed to abort or commit. If it crashes during processing, it may be resubmitted. A typical application enters its executing state when started, from where it may fail or succeed. Failure or success is usually associated with a return code. An RDA (Relational Data Access protocol) transaction has an explicit prepared state, which enables mutual commit protocols across transactions. Our approach is not limited to these agents.

The agent performs an important function: it interfaces the task with the scheduling system. It informs the system of uncontrollable events like *abort* and requests permission for controllable ones like *commit*. When triggered by the system, it causes appropriate events like *start* in the task. An agent may intercept requests sent or remote procedure calls made to the task (as in our implementation), or be explicitly informed of all transitions by the task (this requires some reprogramming, but is conceptually simpler).

Local autonomy is highly desirable. It is maximized by the above approach, since the task is minimally affected when its agent is inserted into the computational system. The invisible states of the task are not exposed by the agent and its interface is not violated. This approach lends itself to distribution, since the agents can be naturally placed close to their respective tasks.

Our key idea is to instantiate active entities for different events. Like agents, these entities are actors in our actor-based, object-oriented execution environment. This means they are lightweight threads and communicate through symbolic IO ports. At most one copy of the execution environment need be active at each host: this exists as a single process that manages low-level communications and data resources. We have implemented protocols to access various database systems.

There is one actor per event. This actor maintains the current *guard* for the event and manages communications. The guard is a temporal logic formula, which defines the condition under which that event may occur. In the simplest case, when a task agent is ready to make a transition, it attempts the corresponding event. Intuitively, an event can happen only when its guard evaluates to true. If the guard for the attempted event is true, it is allowed right away.

Otherwise, it is *parked*. When an event happens, messages announcing its occurrence are sent to actors of other relevant events. These may be at remote sites on the network. When an event announcement arrives, the receiving actor simplifies its guard to incorporate this information. If the guard becomes true, then the appropriate parked event is enabled.

The above is the simplest case. Further complexity is involved in treating events that may be instantiated multiple times (to ensure that all and only the right event instances occur). Other situations that must be handled include those where the given event may have to be proactively triggered, or where the given event is such that it cannot be delayed. Lastly, there are cases where events have mutual (positive or negative) constraints. We discuss these below.

Section 2 shows how workflows can be formally expressed with dependencies and how dependencies can be used to schedule events. Section 3 shows how to compile constraints on events into guards that can be localized on the individual events, and how to execute events using their guards. Section 4 shows how to accommodate event attributes into guard computations. Lastly, section 5 briefly discusses some additional subtleties of workflow execution.

2 Dependencies and Event Scheduling

Event symbols are the atoms of our language. For ease of exposition and to facilitate comparison with the literature, we initially assume that the event symbols denote unique instances of different types. In section 5 we allow the symbols to name event types, which may be multiply instantiated.

We introduce for each event symbol e a symbol $\overline{e}$ corresponding to its complement ($\overline{\overline{e}} = e$). We return to event complementation in section 5.3. Our algebraic language, $\mathcal{E}$, is used to specify acceptable computations or *traces*. All traces considered are *legal* in that

- an event (instance) occurs at most once in any trace, and
- if an event (instance) occurs in a trace, then its complement does not occur in that trace.

$\mathcal{E}$ contains constants 0 and $\top$, and operators for choice ($+$), interleaving ($|$), and sequence ($\cdot$). Roughly, e (resp. $\overline{e}$) means that event e (resp. $\overline{e}$) should occur; $E + F$ (resp. $E|F$) means that one of (resp. both) E and F should hold; and, $E \cdot F$ means that E should hold on the initial part of a trace and F on the remainder. Each expression or dependency identifies a set of event traces, namely, those of which it is a true description. 0 is true of no trace, and $\top$ is true of all.

Σ is the set of significant event symbols, Γ is the *alphabet*, and $\mathcal{E}$ is our formal language of event expressions.

Syntax 1 $e \in \Sigma$ implies that $e, \overline{e} \in \Gamma$

Syntax 2 $\Gamma \subseteq \mathcal{E}$

Syntax 3 $E_1, E_2 \in \mathcal{E}$ implies that $E_1 \cdot E_2$, $E_1 + E_2$, $E_1|E_2 \in \mathcal{E}$

Syntax 4 $0, \top \in \mathcal{E}$

As running examples, we use two dependencies well-known in the literature. In Klein's [Kle91] notation, $e < f$ means that if both events e and f happen, then e precedes f. This is precisely captured by $(\overline{e} + \overline{f} + e \cdot f)$—see Example 1 below. Klein's $e \rightarrow f$ means that if e occurs then f also occurs (before or after e). This is captured by $(\overline{e} + f)$—see Example 2 below.

Example 1 Let $D1 = \overline{e} + \overline{f} + e \cdot f$. Let τ be a legal trace that satisfies $D1$. If τ satisfies both e and f, then e must occur before f on τ. This is because τ satisfies e and satisfies f iff both e and f occur on τ. Since τ is legal, neither $\overline{e}$ nor $\overline{f}$ can occur on τ. Hence, to satisfy $D1$, τ must satisfy $e \cdot f$, which requires that an initial part of τ satisfy e and the remainder satisfy f. In other words, e must precede f on τ. ∎

Example 2 Let $D2 = \overline{e} + f$. Let τ be a legal trace that satisfies $D2$. If τ satisfies e, then it must satisfy f. This is because τ satisfies e iff e occurs on τ. Since τ is legal, $\overline{e}$ cannot occur on τ. Hence, to satisfy $D2$, τ must contain f. There is no constraint as to the relative order of e and f. ∎

Now we have the basic machinery available to define a simple workflow in our formalism. The following example shows that a number of decisions must be taken about the schedules one expects from a given workflow, and that it is possible to capture the various options as dependencies in our approach.

Example 3 Consider a workflow which attempts to *buy* an airline ticket and *book* a car for a traveler. The key semantic requirement is that both or neither task should have an effect. Mutual commit, e.g., two-phase commit, protocols cannot be executed, since the airline and car rental agency are different enterprises and possibly their databases don't have a visible precommit state. We can use the fact that there are several mutually indistinguishable instances of plane seats and rental cars.

Assume that (a) the booking can be canceled: thus *cancel* compensates for *book*, and (b) the ticket is nonrefundable: *buy* cannot be compensated. Assume all subtasks have at least *start*, *commit*, and *abort* events, like RDA transactions, as in Figure 1. For simplicity, assume that *book* and *cancel* always commit. Now the desired workflow may be specified as (1) $\overline{s_{buy}} + s_{book}$ (initiate *book* upon starting *buy*), (2) $\overline{c_{buy}} + c_{book} \cdot c_{buy}$ (if *buy* commits, it commits after *book*—this is reasonable since *buy* cannot be compensated and commitment of *buy* effectively commits the entire workflow), and (3) $\overline{c_{book}} + c_{buy} + s_{cancel}$ (compensate *book* by *cancel*).

Note that (2) explicitly orders c_{book} before c_{buy}—as in Example 1 above. However, (1) and (3) do not order any events—see Example 2 above. When the events s_{book} and s_{cancel} are to be triggered only by the scheduler, then the above specification suffices. However, to be triggered, the events should have the attribute *triggerable*—see section 4. The scheduler causes the events to occur when necessary, and may order them before or after other events as it sees fit. However, if the two events may occur independently, then the specification must be strengthened further. For instance, if s_{book} may be directly attempted by a user application independently of s_{buy}, then we must ensure that either (i) s_{book} is accepted only if s_{buy} also occurs—e.g., by adding $\overline{s_{book}} + s_{buy}$, or (ii) s_{book} is accepted but s_{cancel} occurs unless s_{buy} occurs—e.g., by adding $\overline{c_{book}} + \overline{s_{buy}} + s_{cancel}$. ∎

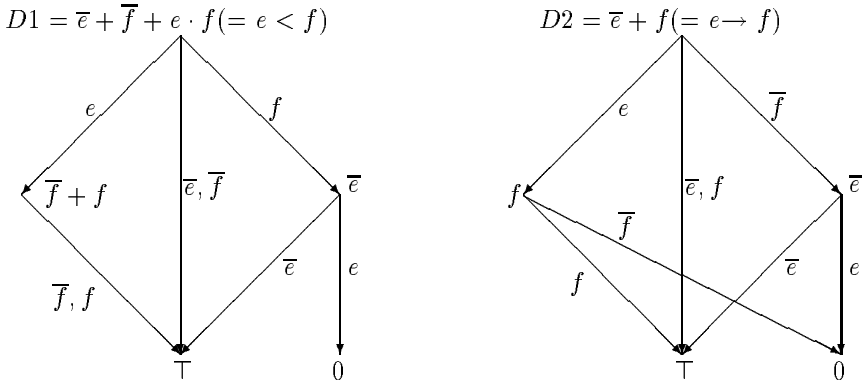


Figure 2: Scheduler States and Transitions Represented Symbolically

The scheduler must ensure that any trace that is realized satisfies all stated dependencies. An important component of the state of the scheduler is determined by the dependencies it is

enforcing, because they specify the traces it must allow. As events occur, the possible traces get narrowed down. An event e occurs when the scheduler (a) accepts that event if requested by the task agent in which that event arises, (b) triggers that event in the task agent on its own accord, or (c) rejects the complement of that event if the complement is requested by the task agent.

Consider how the state of the scheduler evolves when it enforces a dependency. After each event, the state equals the remnant of the dependency yet to be enforced.

Example 4 Figure 2 shows the state changes for dependencies $D1 = (\overline{e} + \overline{f} + e \cdot f)$ and $D2 = (\overline{e} + f)$. For $D1$, if the complement of events e or f happens, then $D1$ is necessarily satisfied. If e happens, then either f or $\overline{f}$ can happen later. But if f happens, then only $\overline{e}$ must happen afterwards (e cannot be permitted any more, since that would mean f precedes e). Similarly, for $D2$, the scheduler can permit $\overline{e}$ or f to happen right away, but if e happens first, it must be followed by f and if f happens first, it must be followed by $\overline{e}$. ∎

3 Guards on Events

The above is a *dependency-centric* view of the scheduling procedure. This procedure is most naturally centralized, although the dependency representations could be distributed appropriately. However, in many practical cases, it is a good idea to take the dual, *event-centric*, view. This too can be either centralized or distributed; in either case, it preprocesses the dependencies so that the information pertinent to an event is kept locally. As events prepare to happen and happen, messages are sent to ensure that other events learn of their recent or imminent occurrence. Thus only essential messages need be transmitted. This is the approach that we have taken.

In order to properly convert from the dependency representations to the event representations, we consider all possible computations relevant to each dependency to determine the various conditions in which a given event can occur: what should have happened already and what should be guaranteed to happen. The sum of these conditions denotes the *guard* of the event, i.e., the weakest condition that must hold for the event to occur, so that correctness is preserved. It turns out that in the dependencies that are most often of interest, the guards of the participating events are succinct temporal formulae.

Taking this approach requires the resolution of the following issues: (a) how does one come up with the right guards on different events, (b) how does one evaluate the guard on an event, and (c) how does one communicate information about the occurrence or expected occurrence of an event to events in other agents? Briefly, the guard on an event is derived from the applicable dependencies mentioning that event and from the attributes of that and other events. Before we specify how guards are obtained, we must augment the underlying language so as to have a sufficiently expressive formalism.

3.1 Temporal Logic

The guards on events are temporal formulae. This is necessary so that decisions made on different events can be sensitive to the state of the system, particularly with regard to which events have occurred, which have not occurred but are expected to occur, and which will never occur. These conditions are essential in order to properly enforce various dependencies.

We now present the formal language, $\mathcal{T}$, in which the guards are expressed. This language enables events that have occurred to be explicitly distinguished from those that have not yet occurred as well as from those that have not yet occurred, but will eventually occur.

Syntax 5 $\mathcal{E} \subseteq \mathcal{T}$

Syntax 6 $E_1, E_2 \in \mathcal{T}$ implies that $\Box E_1$, $\Diamond E_1$, $\neg E_1$, $E_1 + E_2$, $E_1|E_2$, $E_1 \cdot E_2 \in \mathcal{T}$

The semantics of $\mathcal{T}$ is given with respect to a trace *and* an index into that trace. The trace must be *maximal*, i.e., for each event, contain that event or its complement. Intuitively, one can think of the index as identifying a prefix of the given trace. Thus in effect the semantics is given with respect to pairs of traces, where the first is maximal, and the second trace is a prefix of the first. The second trace describes what has happened, and the first trace what will eventually have happened.

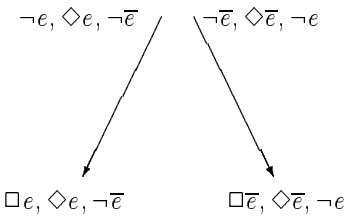


Figure 3: Temporal Operators Related to Events

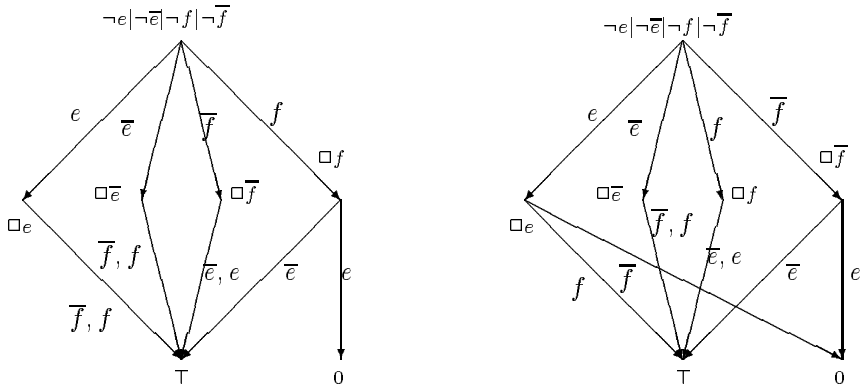
The general technical definitions are given in [ST94]. Informally, a pair satisfies an event e, if e occurs on its second trace, i.e., if e has happened. This definition validates the *stability* of events, meaning that if e is true for a trace, u, then it is also true for all traces of which u is a prefix. A pair satisfies $\Box E$, if E will always be true. A pair satisfies $\Diamond E$ if E will eventually hold (thus, $\Box e \Rightarrow \Diamond e$) A pair satisfies $\neg E$, if E does not yet hold. Because of stability, we have $\Box e = e$, where $e \in \Gamma$. The corresponding assertion for all expressions of $\mathcal{T}$ is false, e.g., $\Box\neg e \neq \neg e$. The formula $\neg e$ is thus the boolean complement of $\Box e$ ($\neg e + \Box e = \top$ and $\neg e|\Box e = 0$). Also, $\neg e + \Box\overline{e} = \neg e$.

Figure 3 shows the possible traces with respect to event e. On different possible traces, the event or its complement may occur. Initially, neither e nor $\overline{e}$ has happened, so $\neg e$ and $\neg\overline{e}$ hold on both traces. But, the trace on which e will occur satisfies $\Diamond e$, and the other $\Diamond\overline{e}$. After e occurs, $\Box e$ becomes true, $\neg e$ becomes false, and $\Diamond e$ and $\neg\overline{e}$ remain true. Figure 3 illustrates the following results: (a) $\Box e + \Box\overline{e} \neq \top$—neither e nor $\overline{e}$ may have occurred at certain states, e.g., initially, (b) $\Diamond e + \Diamond\overline{e} = \top$—eventually either e or $\overline{e}$ will occur, (c) $\Diamond e|\Diamond\overline{e} = 0$—both e and $\overline{e}$ will not occur, and (d) $\Diamond e + \Box\overline{e} \neq \top$—initially, $\overline{e}$ has not happened, but e may not be guaranteed. Günthör's approach validates $\Diamond e + \Box\overline{e} = \top$, which we believe is a major shortcoming. The above and allied results were our main motivation in designing the formal semantics of $\mathcal{T}$—its details are not necessary for this paper.

3.2 Computing Guards

For expository ease, we begin with the simplest case; section 4 discusses additional features. Figure 4 illustrates our procedure for the dependencies introduced previously. The initial node in the dependency representation is labeled $\neg e|\neg\overline{e}|\neg f|\neg\overline{f}$ to indicate that no event has occurred yet. The nodes in the middle layer are labeled $\Box e$, etc., to indicate that the corresponding event has occurred. To avoid clutter, labels like $\Diamond e$ and $\neg e$ are not shown after the initial state.

To compute the guard on an event, we sum the contributions of different paths on which it occurs. The contribution of a path ending in 0 is 0. For a path ending in $\top$, the contribution is the label of the pre-state for the event conjoined with the label of the future post-state. In each case, we delete the event and its complement from the label. $\mathsf{G}(D, e)$ denotes the guard on e due to dependency D.



$D1 = \overline{e} + \overline{f} + e \cdot f (= e < f)$ $\qquad$ $D2 = \overline{e} + f (= e \rightarrow f)$

Figure 4: Computing Guards on Events With Respect to Different Dependencies

Example 5 $\mathsf{G}(D1, e) = (\neg f|\neg\overline{f}|\Diamond(\overline{f} + f)) + (\Box\overline{f}|\top)$. But $\Diamond(\overline{f} + f) = \top$. Hence, $\mathsf{G}(D1, e) = (\neg f|\neg\overline{f}) + \Box\overline{f}$, which reduces to $\neg f + \Box\overline{f}$, which equals $\neg f$. $\mathsf{G}(D1, \overline{e}) = (\neg f|\neg\overline{f}|\Diamond(\overline{f} + f)) + (\Box f|\top) + (\Box\overline{f}|\top)$, which reduces to $\top$. $\mathsf{G}(D1, \overline{f}) = \top$. $\mathsf{G}(D1, f) = (\neg e|\neg\overline{e}|\Diamond\overline{e}) + \Box e + \Box\overline{e}$, which simplifies to $\Diamond\overline{e} + \Box e$.

Consequently, $\overline{e}$ can occur at any point in the computation, and e can occur if f has not yet happened (possibly because it will never happen). Similarly, $\overline{f}$ can occur anywhere, but f can occur only if e has occurred or $\overline{e}$ is guaranteed. ▌

Example 6 $\mathsf{G}(D2, e) = (\neg f|\neg\overline{f}|\Diamond f) + (\Box f|\top)$. Since $\Diamond f$ entails $\neg\overline{f}$, this reduces to $(\neg f|\Diamond f) + \Box f$. Since $\Box f$ and $\neg f$ are boolean complements of each other, we obtain $\mathsf{G}(D2, e) = \Diamond f + \Box f$, which equals $\Diamond f$. $\mathsf{G}(D2, \overline{f}) = (\neg e|\neg\overline{e}|\Diamond\overline{e}) + \Box\overline{e}$, which using the same reasoning as above simplifies to $\Diamond\overline{e}$. $\mathsf{G}(D2, \overline{e}) = (\neg f|\neg\overline{f}|\Diamond\top) + (\Box f|\top) + (\Box\overline{f}|\top)$, which reduces to $\top$. $\mathsf{G}(D2, f) = \top$.

Consequently, $\overline{e}$ and f can occur at any time; e can occur if f has happened or will happen; and $\overline{f}$ can occur if $\overline{e}$ has happened or will happen. ▌

The guards on different events can be calculated at compile-time: they can be kept precompiled for frequently occurring dependency expressions.

3.3 Execution Through Guard Evaluation

Execution with guards is conceptually straightforward. The guard on an event is evaluated when it is submitted. Since guards are simplified whenever an event mentioned in them occurs, evaluation usually means checking if the guard is $\top$ (subexpressions of the form $\neg f$ cannot always be simplified). However, preprocessing is necessary to ensure that temporal expressions are consistently evaluated and that tasks are not unnecessarily delayed.

3.3.1 Prohibitory Relationships

Let e be the given event, whose guard is being evaluated. Roughly, a subexpression of the form $\neg f$ evaluates to $\top$, since it means that f has not yet happened: if f had happened, $\neg f$ would have been reduced to 0. But such subexpressions are potentially problematic, since the message announcing f could be in transit when $\neg f$ is evaluated, leading to an inconsistent evaluation. Thus, a message exchange with f's actor is essential to ensure that f has not happened and is not happening. This is an example of a *prohibitory* relationship between events, since f's

occurrence can possibly disable e (depending on the rest of the guard of e). The left part of Figure 5 diagrams the message flow in this case.

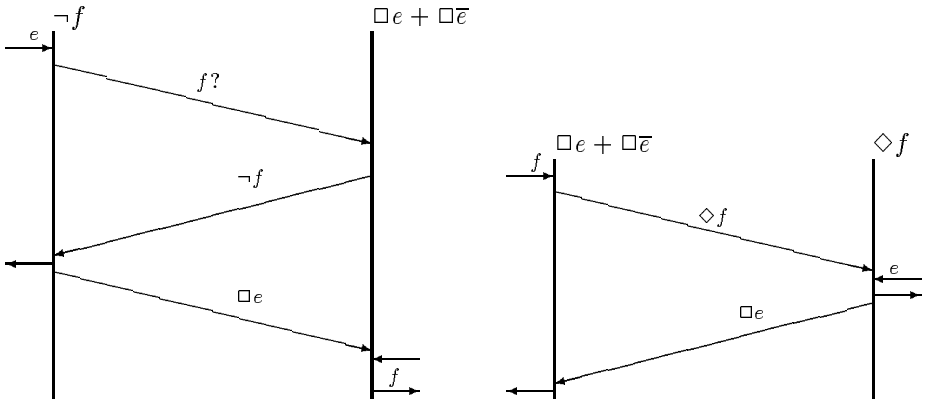


Figure 5: Prohibitory and Promissory Message Flows

3.3.2 Promissory Relationships

If the guard on an event is neither $\top$ nor 0, then the decision on it can be deferred. The execution scheme must be enhanced to prevent mutual waits in consistent situations:

Example 7 Consider dependencies $D1$ and $D2$ of Figure 2. These result in e's guard being $\Diamond f | \neg f$ and f's guard being $\Box e + \Diamond \overline{e}$. Roughly, this means that e waits for $\Diamond f$, while f waits for $\Box e$. ∎

The guards given in Example 7 do not reflect an inconsistency, since f is allowed to occur after e. One way in which to resolve this kind of wait is through *promises* (this appears related to Klein's approach [Kle91]). This relationship is recognized at compile-time. The events are setup so that when f is submitted, it promises to happen if e occurs. Since e's guard only requires that f occur sometimes, before or after e, it is then enabled and happens upon submission. When news of e's occurrence reaches f, f discharges its promise by occurring—events always keep their promises.

Example 8 Consider dependency $D2$ of Figure 2 and its "transpose" $D2^T = (\overline{f} + e)$. These result in e's guard being $\Diamond f$ and f's guard being $\Diamond e$. Thus, e waits for $\Diamond f$, while f waits for $\Diamond e$. This kind of a cyclic wait can also be handled through promising; either temporal order of e and f is acceptable. ∎

The message $\Box e$ means e has occurred; $\Diamond e$ means that e is promised. Thus, when e occurs, $\Box e$ is asserted to all events whose guards mention e. Similarly, when e is promised, $\Diamond e$ is asserted. $\Box e \Rightarrow \Diamond e$ means that occurrence is stronger than promising (see Figure 3). We use a limited set of proof rules to reduce guards when events occur or are promised. $\Box e$ reduces subexpressions $\Box e$ or $\Diamond e$ reduce to $\top$, and $\neg e$ to 0. However, $\Box e$ and $\neg e$ are unaffected when $\Diamond e$ is received. And, $\Box e$ and $\Diamond e$ reduce to 0 and $\neg e$ to $\top$, when $\Box \overline{e}$ or $\Diamond \overline{e}$ is received. The right part of Figure 5 diagrams the message flow in this case.

4 Incorporating Event Attributes

The definition of guards given in section 3.2 ignores the attributes of different events. However, the following attributes are crucial to scheduling [ASSR93].

- *Triggerable*: events that the system can initiate;
- *Rejectable*: events that the system can prevent; and
- *Delayable*: events that the system can delay.

A nondelayable event must also be nonrejectable, because it happens before the system learns of it. Such an event is not attempted: the scheduler is notified of its occurrence after the fact. It is possible to have nonrejectable, but delayable, events.

Example 9 The scheduler may trigger a *start* or an *abort*, but not a *commit* (see Figure 1). However, it can reject or delay a *commit*, but can neither delay nor reject an *abort* (a task may unilaterally abort). The scheduler can delay but not reject a *forget* (not shown), in which a task clears its bookkeeping data and releases its locks. Timer events (not shown) are not delayable, rejectable, or triggerable. ∎

The correctness and executability of the schedules generated depends crucially on the attributes of the involved events. We now show how to compute guards in a manner that takes attributes into account. These guards can then be executed as before, but ensure that no attribute constraint is violated.

The key intuition here is that traces that violate some event attribute must be eliminated. A dependency excludes certain computations as illegal; additional knowledge of event attributes further restricts the set of allowed computations. Thus event attributes can be seen as means of strengthening a given dependency. We eliminate from a dependency all the risky paths. The updated guard computations, as given next, effectively sets the contributions of all risky paths to 0.

It turns out that the computations are most natural when the attributes are paired as nondelayable and nonrejectable, and delayable and nonrejectable, respectively. The default of section 3.2 applies to delayable and rejectable events; we argued above that nondelayable and rejectable events are impossible.

4.1 Delayable and Nonrejectable Events

An event that is delayable but nonrejectable must be permissible on every computation, possibly after some other events have transpired. Thus the scheduler must realize only those computations on which the given event can occur later (if it has not occurred already). The basic approach is as described in section 3.2, except that it is applied to a dependency representation from which certain paths are removed.

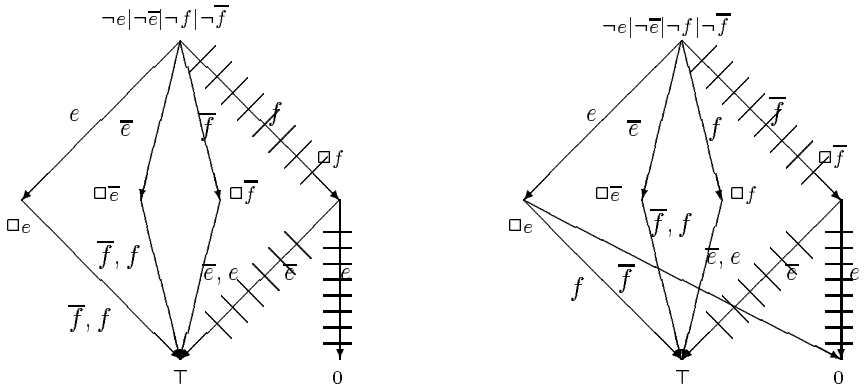
Figure 6 shows the dependencies of Figure 4 with paths deleted to reflect the fact that event e is delayable and nonrejectable. Using this representation, we can readily determine the guards for the various events.

Example 10 $\mathsf{G}(D1, e) = \neg f | \neg \overline{f} + \Box \overline{f} = \neg f + \Box \overline{f} = \neg f$. Similarly, $\mathsf{G}(D1, f) = \Box e + \Box \overline{e}$. Thus, $\mathsf{G}(D1, e)$ is unchanged but $\mathsf{G}(D1, f)$ is stronger: since e cannot be rejected, we cannot let f happen unless e or $\overline{e}$ has already happened. ∎

Example 11 Again referring to Figure 6, we can compute the following: $\mathsf{G}(D2, e) = \Diamond f$; $\mathsf{G}(D2, \overline{f}) = \Box \overline{e}$; $\mathsf{G}(D2, \overline{e}) = \neg f | \neg \overline{f} + \Box f = \neg \overline{f} + \Box f$; and $\mathsf{G}(D2, f) = \top$. ∎

4.2 Nondelayable and Nonrejectable Events

An event that is nondelayable (and, therefore, also nonrejectable) must be permissible in every state of the scheduler, i.e., of each dependency. Thus the scheduling system must never take an action that leaves it in a state where the given event cannot occur immediately. Thus, in essence, the guard of any such event is set to $\top$ by fiat. This category corresponds to *uncontrollable* events of Klein and Günthör.



$D1 = \overline{e} + \overline{f} + e \cdot f (= e < f)$ $\qquad$ $D2 = \overline{e} + f (= e \rightarrow f)$

Figure 6: Guards Assuming e is Nonrejectable, but Delayable

Example 12 Figure 6 still holds with the assumption that e is not delayable or rejectable. Using this representation, we can readily determine that $\mathsf{G}(D1, f) = \Box e + \Box\overline{e}$. It is necessarily the case that $\mathsf{G}(D1, e) = \top$. Thus, $\mathsf{G}(D1, e)$ is weakened, but $\mathsf{G}(D1, f)$ is strengthened: since e cannot be rejected, we cannot risk letting f happen unless e or $\overline{e}$ has already happened. ∎

Example 13 Again referring to Figure 6, we can compute the following: $\mathsf{G}(D2, e) = \top$; $\mathsf{G}(D2, \overline{f}) = \Box\overline{e}$; $\mathsf{G}(D2, \overline{e}) = \neg f | \neg\overline{f} + \Box f = \neg\overline{f} + \Box f$; and $\mathsf{G}(D2, f) = \top$. ∎

4.3 Enforceability of Dependencies

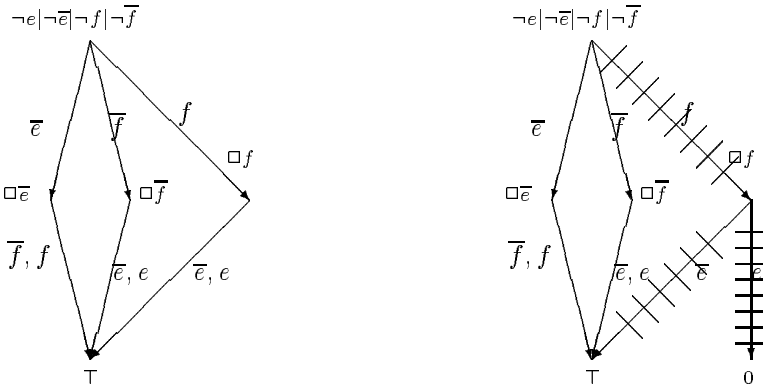
Dependencies can be enforced during workflow execution as long as appropriate guards for events can be found. Enforceability must be checked at compile-time, when dependencies are added. Individual dependencies may become unenforceable if the asserted attributes as overconstraining.

Example 14 Dependency $D1$ of Figure 6 is enforceable if e is nonrejectable and f is nonrejectable and delayable, because f can be delayed until e or $\overline{e}$ occurs. However, $D1$ is not enforceable if e is nonrejectable and f is nondelayable, because the nonrejectability of e removes the path beginning with f from the initial state. ∎

Example 15 Figure 7 diagrams a dependency, $D3 = \overline{e} + \overline{f} \cdot e + f \cdot e$, which is enforceable when e is nonrejectable and delayable (because e can occur after f or $\overline{f}$), but not enforceable when e is nondelayable (because e cannot occur right away). $D3$ continues to be enforceable when both e and f are nonrejectable and and e is delayable. By contrast, $D4 = \overline{e} + \overline{f} \cdot e$ is not enforceable even when e and f are both nonrejectable and delayable, because there is no path on which they both occur. ∎

However, it is possible that dependencies are individually, but not jointly, enforceable. This can be detected when the guards on different events as contributed by different dependencies are conjoined and clash with the attributes of those events.

Example 16 Let $D5 = \overline{f} + \overline{e}$. The dependencies $D1 = \overline{e} + f$ and $D5$ are jointly enforceable, although $\mathsf{G}(D1, e) | \mathsf{G}(D5, e) = \Diamond f | \Diamond\overline{f}$, which reduces to 0. This just means that e must always be rejected. However, if we are given the fact that e is nonrejectable, then the dependencies are jointly unenforceable, since the guard of a nonrejectable event must be an expression that will eventually evaluate to $\top$. ∎



$D3 = \overline{e} + \overline{f} \cdot e + f \cdot e$ $\qquad$ $D4 = \overline{e} + \overline{f} \cdot e$

Figure 7: Nondelayable is Stronger than Nonrejectable and Delayable

4.4 Triggerable Events

The computational consequence of a dependency can vary based on whether some event is triggerable.

Example 17 Consider dependency $D2$ of Figure 2, which says that if e happens, then f must also happen. The guards due to $D2$ on the various events can be readily checked: $\mathsf{G}(D2, e) = \Diamond f$; $\mathsf{G}(D2, \overline{e}) = \top$; $\mathsf{G}(D2, f) = \top$; and $\mathsf{G}(D2, \overline{f}) = \overline{e}$. ∎

Ordinarily, these guards capture the desired behavior. In our usual execution scheme, if e is attempted, it will wait for f to be promised. When f is triggerable, this is unsatisfactory, since f would never happen by itself. An indefinite wait can result.

We must setup the information flow so that f is triggered. A principled way to do this is as follows. Recall that an event or its complement must eventually occur, and that they both cannot occur. When event e occurs (or is promised), it causes the guard of $\overline{f}$ to reduce to 0. This, in effect, prevents $\overline{f}$ from ever occurring. In other words, it forces f to occur, sooner or later. Consequently, we can set $\mathsf{G}(D2, e) = \top$, provided our execution scheme guarantees that f will be triggered when the guard on $\overline{f}$ goes to 0. The contributions of other dependencies to the guards of both events are unchanged. The above relationship is recognized when the guard on an event includes the promise or occurrence of a triggerable event.

5 Further Subtleties

Sections 3 and 4 gave the core ideas about computing and executing guards on events. We now show how some further complexities are handled in our approach.

5.1 Multiply Instantiated Events

Multiply instantiated events are handled in the expected manner by introducing unique event identifiers. An event is uniquely determined by (a) the identity of the agent in which it occurs and (b) the count of the event. Thus each agent can maintain a counter for each event (or a single counter for all events) and increment it whenever it attempts an event or an event is triggered in it. This guarantees uniqueness. For practical reasons, we consider events in different agents to be different types and have a separate actor for each of them. Different instances of an event are different tokens of the same type.

5.2 Deadlocks and Race Conditions

The underlying execution mechanism should avoid provide a consistent view of the temporal order of events. It should avoid potential race conditions and deadlocks. The compilation phase can detect these conditions and add messages to ensure that there are no problems. It is also possible to show that if the guards are automatically generated from a fixed set of dependencies, then certain problematic conditions won't occur. This set of dependencies is quite adequate for most workflow specifications, but we do not discuss any details here.

5.3 Event Complements Revisited

We require that $\overline{e}$ exist for each significant event e. In practice, some events may not have a complement. For example, the *start* event of a task may have no complement (see Figure 1). But our assumption simplifies the required reasoning. A user may state a dependency $\overline{s_1}+\overline{s_2}+s_1s_2$. By assigning a meaning to $\overline{s_1}$ and $\overline{s_2}$, even if it is the empty set, we can proceed without regard to the fact that $\overline{s_1}$ and $\overline{s_2}$ never.

Also, task agents may easily be defined in which there is a three-way split, meaning that the first event along each split is complemented by the first events of the other two splits. However, these can be captured in our approach by defining three separate events, each of which triggers the complements of the other two.

5.4 Control Flow and Data Flow

Depending on the attributes of the events, dependencies can model both control flow and data flow requirements. Control flow involves any kind of conditional triggering, whereas data flow involves waiting for another computation to complete.

Example 18 Consider dependency $D1$ of Figure 2 again. For simplicity, assume there are no other dependencies. If f is triggerable, then e's occurrence causes f to fire. However, if f is not triggerable, then e must be delayed until f is guaranteed to occur. If in addition, we had required that f precede e, then in essence, we can model the flow of information from f (or its associated task) to e (or its associated task). That is, f could be a write event and e a read event. ∎

6 Conclusions

A prototype of our system has been implemented [STW94]. Our approach is provably correct, and applies to many useful workflows in heterogeneous, distributed environments. Much of the required symbolic reasoning can be precompiled, leading to efficiency at run-time. Although we begin with lazy specifications, which characterize entire traces as acceptable or unacceptable, we setup our computations so that information flows as soon as it is available, and activities are not unnecessarily delayed. We believe this will lead to good scalability. Although we focused on the intuitive, implementational aspects of our approach, a theoretical basis exists for it.

Klein's approach, which is only sketchily described, is the closest to our approach in that it is event-centric and distributed [Kle91]. However, it is limited to tasks without loops, and doesn't handle event attributes. Also, he tried to obtain the effect of prohibitory message flows through more complex promises, which seems unintuitive. Günthör's approach is based on temporal logic, but is centralized [Gün93]. These approaches are somewhat *ad hoc* in their details and do not properly handle complementary events. Also, they do not consider all the attribute combinations that we motivated above. Lastly, our previous approach, which constructs finite automata for dependencies, is centralized [ASSR93]. It uses pathset search to avoid generating

product automata, but the individual automata can be quite large. The event attributes used here were introduced there. Neither of the above approaches can express or process complex dependencies as easily as the described approach.

In our system, events variously wait (at their actors), send messages to each other, and thereby enable or trigger each other. This appears intuitively similar to Petri nets in some respects. Petri nets are indeed an important formalism. However, our goal in this paper was to find a way to characterize workflows that may be weaker than Petri nets, but which has just enough power to do what we need and is declaratively specified. Indeed, in a sense we synthesize Petri nets automatically by setting up the appropriate message flows. At compile-time, we also ensure that the "net" will operate correctly, e.g., by not deadlocking at mutual waits, but generating appropriate promissory messages instead.

Our approach enables certain useful kinds of reasoning to be formalized and to some extent automated. These aspects of the approach follow from the formal semantics of our language that we carefully designed. These are outside the scope of this paper. Also outside the scope of this paper is a discussion of how one may actually come up with the necessary intertask dependencies to capture some desired workflow. This is highly important and a problem that has been attacked by others, especially in the formulation of various extended transaction models [Elm92].

References

[ASSR93] Paul C. Attie, Munindar P. Singh, Amit P. Sheth, and Marek Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th VLDB Conference*, August 1993.

[CR92] P. Chrysanthis and K. Ramamritham. ACTA: The SAGA continues. In *[Elm92]*. 1992. Chapter 10.

[DGMH+93] Umesh Dayal, Hector Garcia-Molina, Mei Hsu, Ben Kao, and Ming-Chien Shan. Third generation TP monitors: A database challenge. In *SIGMOD*, May 1993.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Proceedings of the 17th VLDB Conference*, September 1991.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[Gün93] Roger Günthör. Extended transaction processing based on dependency rules. In *Proceedings of the RIDE-IMS Workshop*, 1993.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Kim94] Won Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1994.

[Kle91] Johannes Klein. Advanced rule driven transaction management. In *Proceedings of the IEEE COMPCON*, 1991.

[RS94] Marek Rusinkiewicz and Amit Sheth. Specification and execution of transactional workflows. In *[Kim94]*. 1994.

[SMTA94] Munindar P. Singh, L. Greg Meredith, Christine Tomlinson, and Paul C. Attie. An algebraic approach to workflow scheduling. Technical Report Carnot-049-94, Microelectronics and Computer Technology Corporation, Austin, TX, July 1994.

[ST94] Munindar P. Singh and Christine Tomlinson. Temporal logic constraints for workflows. Technical Report Carnot-087-94, Microelectronics and Computer Technology Corporation, Austin, TX, October 1994.

[STW94] Munindar P. Singh, Christine Tomlinson, and Darrell Woelk. Relaxed transaction processing. In *Proceedings of the ACM SIGMOD*, May 1994. Research Prototype Demonstration.