An Event Algebra for Specifying and Scheduling Workflows

Munindar P. Singh, Greg Meredith,* Christine Tomlinson, and Paul C. Attie†
Microelectronics and Computer Technology Corporation
Research & Development Division
3500 West Balcones Center Drive
Austin, TX 78759
USA

msingh@mcc.com
msingh@cs.utexas.edu

Abstract

Heterogeneous systems arise when preexisting or legacy information systems are integrated in such a way as to preserve their autonomy and past applications. *Workflows* are the semantically appropriate units of activity in such environments. They involve a variety of tasks and are best represented by different extended transaction models or combinations thereof. We present an approach by which workflows can be efficiently scheduled. Our novel contribution is an event algebra in which dependencies characterizing workflows can be declaratively expressed. We show how to symbolically process these dependencies to determine which events can or must occur, and when. Our approach can represent dependencies compactly and reason with them efficiently.

1 Introduction

The design and implementation of heterogeneous information systems poses special problems for task scheduling and management. Heterogeneous systems consist of a number of differently constructed applications and databases 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* [1, 6, 13].

The traditional transaction model defines *ACID* transactions [7], so named because they have the properties of Atomicity, Consistency, Isolation, and Durability. These properties are naturally realized through a syntactic correctness criterion, such as serializability. ACID transactions have proved extremely valuable in traditional, homogeneous, single-node databases, but are not well-suited to heterogeneous systems. First, mutual commit protocols, which are required to ensure atomicity, are problematic. They are inefficient to implement because of distribution and often impossible to implement because of autonomous legacy applications. For technical and political reasons, it is usually impossible to violate the autonomy of local systems. Second, the semantic requirements in heterogeneous applications are often quite complicated and need more sophisticated task structuring [5, 3]. A number of extended transaction models have been proposed recently [4]. Typically, these extended models generalize the ACID model in different directions and do not meld well with each other.

We do not propose a transaction model. Instead, we describe a general approach to declaratively specify and schedule intertask dependencies. Such dependencies can be used to formalize the scheduling aspects of a large variety of, and even combinations of, workflow and transaction models, including all known in the literature. Our approach can express the primitives of Klein [10], which can capture those of Chrysanthis & Ramamritham [2] and Günthör [8]. Klein shows how serializability requirements and extended nested transactions can be captured in his approach, so we do not go into such details here.

Our broad approach is similar to our previous temporal logic work on event scheduling [1]. Our main contribution in this paper is an algebra that yields succinct representations of dependencies and leads to more efficient scheduling of events. We emphasize that our contribution is not in inventing a new formalism. Our formalism is loosely based on that of Pratt [11]. We choose our present formalism over the formalisms of past scheduling approaches [1, 10, 8] because it facilitates the derivation of stronger results. How-

*Currently with the Department of Computing, Imperial College, London.

†Currently with the School of Computer Science, Florida International University, Miami.

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

ever, we make crucial enhancements—defined below—to Pratt's syntax (complement events) and semantics (admissibility) to derive our key results. Without these enhancements, those results would not obtain [14]. In our presentation, we carefully isolate admissibility from the basic semantics and motivate it carefully to show exactly where it is necessary.

Each task or transaction is characterized by a skeletal description, which is captured by its *agent*. This description includes the events or state transitions that are significant for purposes of coordination. Our approach applies to any set of significant events. A typical set for database transactions is *start*, *commit*, and *abort*. Precommit or *done* can be added if visible. The agent of a task interfaces 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). Our theory is not sensitive to this detail. Figure 1 shows some example task agents borrowed from Rusinkiewicz & Sheth [12].

These dependencies can also be used as intratask constraints for monitoring whether system or user tasks meet some minimal requirements, e.g., to identify corrupted task instances. Another use is to mediate in the interactions among heterogeneous components, e.g., to ensure that different components do not violate prespecified protocols of interaction.

In this paper, we describe an algebra of events that is designed for representing and reasoning about dependencies. We present the syntax and basic semantics for our language and then motivate the additional features it must have for the above job. We show how we may use the syntactic procedures of residuation for workflow scheduling; and, lastly, how one may actually schedule using our algebra.

2 Action Algebra

Our formal language is based on an algebra of actions due to Pratt [11]. A user of our system—a DBA or a sophisticated end-user—would typically be supplied with some graphical notation for specifying workflows, which would be translated into our formal language. Event symbols are the atoms of our language. Unlike Pratt, we also introduce for each original event symbol another symbol corresponding to its complement. We return to event complementation in section 5. Throughout, $\triangleq$ means *is defined as*.

2.1 Syntax

Action expressions are formed by the following syntactic definitions. Σ is the set of significant event symbols, Γ is the *alphabet*, and $\mathcal{E}$ is our language of event expressions.

Syntax 1 $\Sigma \subseteq \Gamma$

Syntax 2 $e \in \Sigma$ implies that $\overline{e} \in \Gamma$

Syntax 3 $\Gamma \subseteq \mathcal{E}$

Syntax 4 $E_1, E_2 \in \mathcal{E}$ implies that $E_1 \cdot E_2 \in \mathcal{E}$

Syntax 5 $E_1, E_2 \in \mathcal{E}$ implies that $E_1 + E_2 \in \mathcal{E}$

Syntax 6 $E_1, E_2 \in \mathcal{E}$ implies that $E_1 | E_2 \in \mathcal{E}$

Syntax 7 $0, \top \in \mathcal{E}$

A *dependency*, D, is an expression of $\mathcal{E}$. A *dependency system*, $\mathcal{D}$, is a set of dependencies.

We require an *alphabet* to be closed under event complementation.

Definition 1 Θ is an alphabet $\triangleq \Theta \subseteq \Gamma$ and ($\forall e \in \Gamma$: $e \in \Theta$ iff $\overline{e} \in \Theta$)

And, Γ_E, the alphabet of an expression E, is defined as the set of events mentioned in E, and their complements. Specifically,

Definition 2
$\Gamma_0 \triangleq \emptyset$; $\Gamma_\top \triangleq \emptyset$; $\Gamma_e \triangleq \{e, \overline{e}\}$; $\Gamma_{\overline{e}} \triangleq \{e, \overline{e}\}$
$\Gamma_{E \circ F} \triangleq \Gamma_E \cup \Gamma_F$, where $\circ$ is one of $+$, $\cdot$, and $|$

2.2 Semantics

The semantics of $\mathcal{E}$ is given in terms of computations or *traces*. Each trace is a sequence of events in the given system. It is important to associate expressions with possible computations, because they are used

- to specify desirable computations, and
- to determine event schedules to realize good computations.

For convenience, we overload event symbols with the events they denote and use concatenation variously for sequence expressions and traces. Our usage is always unambiguous. Thus in a semantic context, $e\overline{f}$ means the trace in which event e occurs followed by the event $\overline{f}$.

Let $\mathbf{U}_\Gamma \triangleq \Gamma^*$. Thus $\mathbf{U}_\Gamma$ is the set of all possible traces over Γ. For a trace, $\sigma \in \Gamma^*$, and an expression $E \in \mathcal{E}$, $\sigma \models E$ means that trace σ satisfies expression E. $[\![\,]\!]$ gives the *intension* or *denotation* of an expression: $[\![E]\!] \triangleq \{\sigma : \sigma \models E\}$.

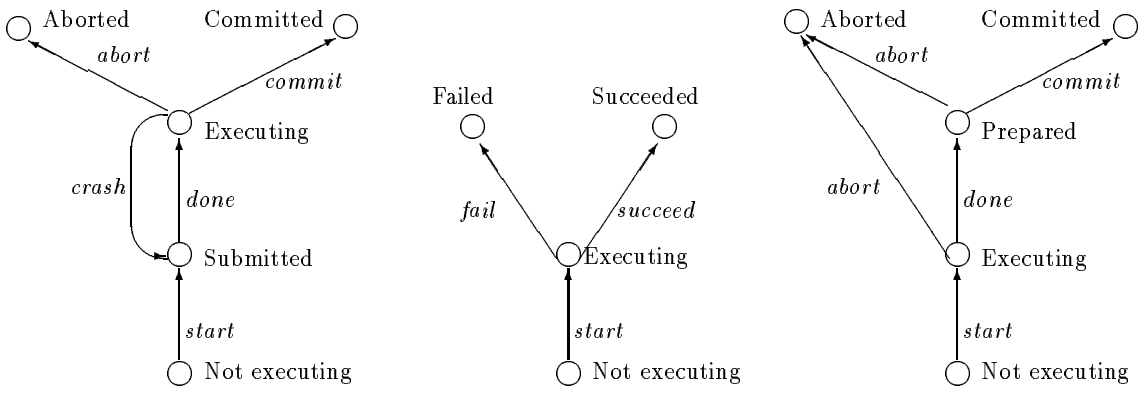
Semantics 1 $[\![f]\!] = \{\sigma : f \text{ is mentioned in } \sigma\}$, if $f \in \Gamma$

Semantics 2 $[\![E_1 + E_2]\!] = [\![E_1]\!] \cup [\![E_2]\!]$

Semantics 3 $[\![E_1 \cdot E_2]\!] = \{\sigma\tau : \sigma \in [\![E_1]\!] \text{ and } \tau \in [\![E_2]\!]\}$

Semantics 4 $[\![E_1 | E_2]\!] = [\![E_1]\!] \cap [\![E_2]\!]$

Semantics 5 $[\![0]\!] = \emptyset$



IMS Transaction **Typical Application** **RDA Transaction**

Figure 1: Some example task agents

Semantics 6 $[\![\top]\!] = \mathbf{U}_\Gamma$

Thus the atom e denotes the set of traces in which event e occurs. $E_1 \cdot E_2$ denotes memberwise concatenation of the traces in the denotation E_1 with those for E_2. $E_1 + E_2$ denotes the union of the sets for E_1 and E_2. Lastly, $E_1|E_2$ denotes the intersection of the sets for E_1 and E_2. This semantics validates various useful properties of the given operators, e.g., associativity of $+$, $\cdot$, and $|$, and distributivity of $\cdot$ over $+$ and over $|$.

Example 1 Let $\Gamma = \{e, \overline{e}, f, \overline{f}\}$. Then $[\![e]\!] = \{e, e\overline{e}, ef, fe, e\overline{f}, ee\overline{e}f\overline{f}, \ldots\}$ and $[\![ef]\!] = \{ef, e\overline{e}f, eff\overline{e}, fee\overline{e}f\overline{f}, \ldots\}$. One can verify that $[\![e + \overline{e}]\!] \neq \top$ and $[\![e|\overline{e}]\!] \neq 0$. ∎

The above semantics gives the formal meaning of algebraic expressions in our language. This semantics seeks to associate with each expression the set of traces over which it is realized. However, the version given above is not the whole story. It does not capture two important aspects of our execution framework. These constraints, given below, both apply to event instances and are quite intuitive.

- An event excludes its complementary event from any computation.
- An event occurs at most once in any computation.

Traces that satisfy these constraints are termed *legal*. Many of the traces in Example 1 are clearly illegal (when e and f are specific instances). An event instance only precludes its very own complement, not the complements of other instances of its type. Similarly, different event instances of the same type can occur in the same computation.

The above restriction enables the interpretation of complement events (formally independent symbols) as semantically complementary. Thus, the event *abort* in an RDA task agent is the complement of the event *commit*. In formal reasoning, therefore, we need an event symbol c interpreted as *commit*; we automatically obtain another event symbol $\overline{c}$, which can be interpreted as *abort*. This observation is used in the examples below.

Example 2 Let $\Gamma = \{e, \overline{e}, f, \overline{f}\}$. Then restricting to legal traces as informally defined above, $[\![e]\!] = \{e, ef, fe, e\overline{f}, \overline{f}e\}$ and $[\![ef]\!] = \{ef\}$. One can verify that $[\![e + \overline{e}]\!] \neq \top$ and $[\![e|\overline{e}]\!] = 0$. ∎

As running examples, we use two dependencies due to Klein [10] that are well-known in the literature. In Klein's notation, $e \rightarrow f$ means that if e occurs then f also occurs (before or after e). Klein's $e < f$ means that if both events e and f happen, then e precedes f. Restricting to legal traces, $e \rightarrow f$ is precisely captured by $(\overline{e} + f)$—see Example 3. Similarly, $e < f$ is precisely captured by $(\overline{e} + \overline{f} + e \cdot f)$—see Example 4.

Example 3 Let $D_\rightarrow = \overline{e} + f$. Let τ be a legal trace that satisfies $D_\rightarrow$—i.e., $\tau \in [\![D_\rightarrow]\!]$. 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 $D_\rightarrow$, τ must contain f. There is no constraint as to the relative order of e and f. ∎

Example 4 Let $D_< = \overline{e} + \overline{f} + e \cdot f$. Let τ be a legal trace such that $\tau \in [\![D_<]\!]$. 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 $D_<$, 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. ∎

One may be tempted to define the universe set to contain only legal traces. However, this prevents proving the soundness of the equations for residuation (introduced in section 3) that are at the core of our approach. We ultimately do restrict the set of possible traces to properly capture the above aspects, but in a more subtle manner—through the definition of *admissibility* (see section 3.2). To highlight the main reasons for admissibility and to convince the reader that we are not modifying the formalism gratuitously, we introduce the needed restrictions later in the technical development. However, the next motivating example of a workflow specification also assumes the restriction to legal traces.

Example 5 Consider a workflow which attempts to *buy* an airline ticket and *book* a car for a traveler. Both or neither task should have an effect. 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 are RDA transactions (as in Figure 1). For simplicity, assume that *book* and *cancel* always commit. Now the desired workflow may be specified as the conjunction of the following three dependencies:

1. $\overline{s_{buy}} + s_{book}$ (if *buy* starts, then *book* must also start),
2. $\overline{c_{book}} + \overline{c_{buy}} + c_{book} \cdot c_{buy}$ (*book* commits before *buy* if both commit), and
3. $\overline{c_{book}} + c_{buy} + s_{cancel}$ (compensate *book* by *cancel*—i.e., if *book* commits and *buy* does not, then start *cancel*).

∎

Thus a workflow may be formally defined as a set of significant events (in different task agents), and a set of dependencies over those events. Indeed, the set of dependencies component is the crucial one of the two.

2.3 Scheduling Events to Enforce Dependencies

Each expression or dependency identifies a set of event traces, namely, those of which it is a true description. Our scheduler, howsoever implemented, must ensure that any trace that is realized satisfies all stated dependencies. Further, the scheduler should meet the converse requirement: roughly, if there is a trace that satisfies all stated dependencies and that may be generated given the events and their order of submission, then the scheduler should be able to realize some trace that satisfies all dependencies.

Example 6 Acceptable traces for the workflow of Example 5 include the following. The reader can readily verify that these traces satisfy each of the dependencies given in Example 5.

1. $s_{buy} s_{book} c_{book} c_{buy}$
2. $s_{buy} s_{book} \overline{c_{buy} c_{book}}$
3. $s_{buy} s_{book} c_{book} \overline{c_{buy}} s_{cancel} c_{cancel}$
4. $s_{book} s_{buy} c_{book} c_{buy}$

∎

An important component of the state of the scheduler is determined by the dependencies it is enforcing, because they specify the traces it is supposed to allow. As events occur, the possible traces get narrowed down. An event e occurs when one of the following three conditions occurs:

1. the scheduler accepts that event if requested by the task agent in which that event arises,
2. triggers that event in the task agent on its own accord, or
3. rejects the complement of that event if the complement is requested by the task agent.

For each dependency, in order to guarantee its satisfaction, the scheduler must keep track of the current state and attain a state labeled $\top$. It is fruitful to consider how the state of the scheduler evolves when it is trying to enforce a dependency.

Example 7 Figure 2 shows the state changes for dependencies $D_< = (\overline{e} + \overline{f} + e \cdot f)$ and $D\rightarrow = (\overline{e} + f)$. For $D_<$, if the complement of events e or f happens, then $D_<$ 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 $D\rightarrow$, 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 $\overline{e}$. ∎

Requirements of the kind as given in the above example, when formalized, lead to a solution to the scheduling problem that is intimately related to the operation of *residuation*. In section 3, we explore the intuition further and motivate the definition of residuation. We present a set of equations that capture the desired properties of residuation, and develop their proofs of soundness, which entails adding admissibility to the models of section 2.2.

3 Residuation

The event scheduler must decide for each event whether or not it must occur. For expository ease, let us consider the case where an event has been attempted by a task agent, and the scheduler must decide whether or not to accept it. Other interesting cases are covered in section 4 below. The key factors on which a decision to accept an event e must be based are as given next. Here *proper* informally refers to traces that meet the legality requirements of section 2 and satisfy the stated dependencies.

1. Whether e can occur in the initial part of the currently remaining part of any of the traces for which the given dependency is true; and
2. Whether accepting e would leave the scheduler in a state where it *may*
   - prevent some proper traces, or
   - generate some improper traces.

For further motivation, let s be the set of traces that satisfy a dependency D. Thus, s characterizes the state of the scheduler. If event e is executed now, let y denote the set of traces that would be permissible at the end of e. In other words, we must replace s by y to reflect the fact that e has occurred. Therefore, y is the set such that

- $e \cdot y \subseteq s$ (relative to s, y contains no improper traces); and
- $(\forall z : e \cdot z \subseteq s \Rightarrow z \subseteq y)$ (relative to s, y contains all proper traces after e)

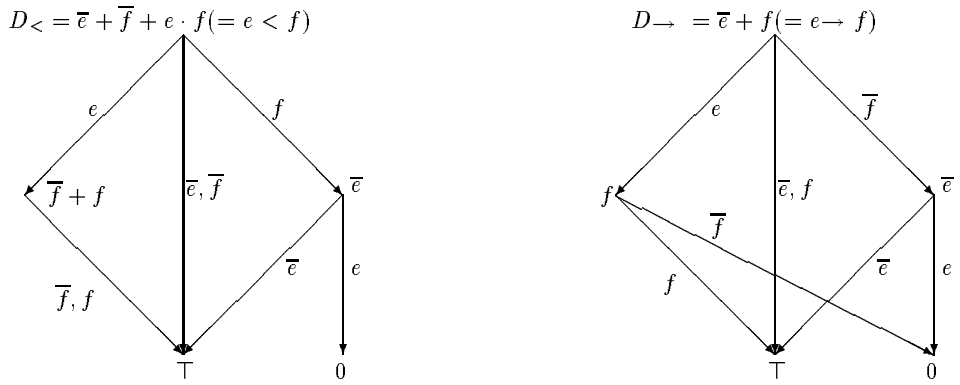


Figure 2: Scheduler states and transitions represented symbolically

3.1 Equations for Residuation

What makes the above extremely exciting is that when s is represented by an expression in $\mathcal{E}$, then y is also represented by an expression in $\mathcal{E}$, which can be efficiently computed from s. Thus, the states of the scheduler can be represented symbolically and its transitions processed algebraically. The operation of computing the resulting set of traces corresponds to residuation in our algebra. We define residuation as an operator /. The residuation operator is not formally in $\mathcal{E}$. Consequently, it cannot be used to specify workflows—e.g., e/f is not a well-formed dependency. It can only be used in reasoning about workflows.

Semantics 7 $[\![E_1/E_2]\!] = \{v : (\forall u : u \in [\![E_2]\!] \Rightarrow uv \in [\![E_1]\!])\}$

In our usage—i.e., in an algorithm for scheduling events—E_1 is a general expression and E_2 is an event. The above is a model-theoretic definition. This definition clearly meets the criteria motivated above, whereby the proper and only the proper traces may be generated. It is possible to characterize residuation symbolically by a set of equations or rewrite rules. The equations below assume that the given expression is in a form where there is no | or + in the scope of the · operator. This holds for conjunctive normal form (CNF) and disjunctive normal form (DNF), each of which can be obtained by repeated application of the distribution laws. Thus, E in equations 3, 9, and 10 must be an atom or a sequence expression. Below, E, etc. are expressions and e, etc. are event symbols. For simplicity, we identify $\overline{\overline{e}}$ with e.

Equation 1 $0/E = 0$

Equation 2 $\top/E = \top$

Equation 3 $(e \cdot E)/e = E$

Equation 4 $(E_1 + E_2)/e = (E_1/e + E_2/e)$

Equation 5 $(E_1|E_2)/E = ((E_1/E)|(E_2/E))$

Equation 6 $E/(E_1 \cdot E_2) = (E/E_1)/E_2$

Equation 7 $E/(E_1 + E_2) = (E/E_1)|(E/E_2)$

Equation 8 $E/e = E$, if $e, \overline{e} \notin \Gamma_E$

Equation 9 $(e' \cdot E)/e = 0$, if $e \in \Gamma_E$ and $e \neq e'$

Equation 10 $(e' \cdot E)/e = 0$, if $\overline{e} \in \Gamma_E$

Example 8 In Figure 2, we can verify that residuating each state label by any of its out-edge labels yields the label of the next state. For instance, $(\overline{e} + \overline{f} + e \cdot f)/e = ((\overline{e}/e) + (\overline{f}/e) + (e \cdot f/e)) = (0 + \overline{f} + f) = (\overline{f} + f)$. Similarly, $(\overline{e} + f)/e = ((\overline{e}/e) + f/e) = (0 + f) = f$ and $(\overline{e} + f)/f = ((\overline{e}/f) + f/f) = (\overline{e} + \top) = \top$. Thus, we need not represent the automaton for any dependency explicitly, but can efficiently compute its transition function as needed. ∎

Observation 1 $D/e = D$, if $f \notin \Gamma_D$ ∎

The above means that events can be scheduled *independently* of dependencies that do not mention them. Combined with Equation 5, it entails the modularity of dependencies. In other words, we can compute the result of residuating a workflow—a set of dependencies—by an event by residuating each dependency separately and essentially ignoring dependencies that do not mention the given event.

Observation 2 $\Gamma_{D/f} \subseteq \Gamma_D - \{f, \overline{f}\}$ ∎

Theorem 3 Equations 1 through 10 are sound. ∎

The soundness of equations 1 to 7 follows directly. However, equations 8, 9, and 10 require admissibility, which we formalize next. The formal proofs are developed in [14].

3.2 Admissibility

Admissible traces are those in which

- no event is repeated,
- no event and its complement both occur, and
- an event or its complement occur.

Intuitively, these traces characterize the eventual behavior of the "environment," i.e., of the collection of tasks whose significant events are scheduled. Let $\mathbf{A}_\Theta$ be the set of all admissible traces on the alphabet Θ. As before, identify $\overline{\overline{e}}$ with e.

Definition 3 σ is maximal over Θ $\triangleq$ ($e \in \Theta$ iff $\overline{e} \in \Theta$) and ($\forall e \in \Theta : \sigma \models e$ or $\sigma \models \overline{e}$)

Definition 4 $\mathbf{A}_\Theta \triangleq \{\sigma : \sigma$ is maximal over Θ and $(\forall e \in \Theta : (\forall \pi, \rho, \tau \in \Theta^* : \sigma \neq \pi e \rho e \tau$ and $\sigma \neq \pi e \rho \overline{e} \tau))\}$

The key idea in using admissibility is as follows. Recall that, roughly, residuation determines whether the residuated expression allows immediate execution of the event with which it is residuated. (It also determines what would remain to be done afterwards, but let us ignore that aspect for the time being.) Suppose that the given event, e, occurs after some other event e' in the sequence expression, E. We know that e' has not occurred yet, or it would have been residuated out already. Therefore, if we let e happen right away, then either (a) we must prevent e' altogether or (b) we must eventually let e' happen followed by another instance of e. Option (a) clearly violates the requirement imposed by E, which is to do e' and then e. Option (b) violates our assumption that events are not repeated. Option (a) is precluded by our semantic definitions, since they require that a sequence containing e' is satisfied only by traces on which e' occurs. Option (b) is precluded by the set of admissible traces, since they exclude repeated occurrences of events. When appropriately generalized and formalized, this argument becomes the proof of soundness of Equation 9. A similar proof can be constructed for Equation 10.

Each dependency is associated with the state of the environment in which it is evaluated. The state of the environment changes as events occur. The scheduler must permit only those events whose immediate execution will not violate any of the asserted dependencies. We use admissibility to capture the key property of the environment that (a) instances of events are not repeated and (b) an event and its complement do not both occur in any trace. Another way of understanding this is as follows. Initially, all the admissible traces are possible. Whenever an event occurs, it *shrinks* the set of admissible traces to those in which neither that event nor its complement occur.

Thus, in effect, we attach to each expression the current context of evaluation. Given our understanding of the meanings of expressions and of the role of residuation, two expressions are interchangeable with respect to a set of admissible traces if they allow exactly the same subset of that set of admissible traces. As a result, two expressions that have different denotations may end up being interchangeable with each other in certain evaluation contexts. The proofs of the soundness of the remaining equations are given using this notion of interchangeability or equivalence.

A formalization of the above argument involves formalizing admissible sets, characterizing the shrinkage of admissible sets as events occur, and defining a notion of equivalence derived from admissibility. Using these we can show that the desired equations are indeed sound when equality is replaced by equivalence. It is also required to show that the above notion of equivalence is well-defined in that equivalent expressions can be interchanged without loss in satisfiability. The technical development is deferred to [14].

4 Back to Scheduling

We have shown how to algebraically represent dependencies and compute the possible transitions of a scheduler. Now we briefly discuss how events may actually be scheduled. There are several ways to apply the algebra. The relationship between the scheduling algorithm and the algebra is similar to that between proof search strategies for a logic and the logic itself. In the case of scheduling, the system has to determine a trace that satisfies all dependencies. It can assign different values to the event literals by accepting or rejecting them.

Example 9 In satisfying dependency $D_<$ of Figure 2, the scheduler may choose any path from the initial state to $\top$. The scheduler could always reject e (i.e., do $\overline{e}$), or reject whatever event is submitted first. Similarly, the scheduler could, in satisfying $D\rightarrow$, always reject e unless f had already occurred. Indeed, given the sole dependency $\top$, the scheduler could decide to reject every event. These may not be acceptable behaviors from the user's point of view, but they will satisfy $D_<$. ∎

Thus, we face a problem of finding good heuristics, e.g., to ensure that events are not unnecessarily rejected. However, it might sometimes be useful to risk rejecting events unnecessarily so as to avoid unbounded delays, which is why it is only a heuristic.

Example 10 For $D_<$, if f is attempted first, the scheduler may either accept f immediately, or delay deciding on it. In the former case, it will be forced to reject e if it is ever attempted; in the latter case, it will delay f until e or $\overline{e}$ is submitted and then allow f to occur after e, or in any order with $\overline{e}$. ∎

Event scheduling depends on two main aspects:

- resolution of dependencies that apply to the same event, and
- the attributes or semantic properties of the given events in the underlying workflow.

The scheduler can take a decision to accept, reject, or trigger an event only if no dependency is violated by that decision. Thus the constraint on an event is the conjunction of the constraints due to different dependencies. As argued in section 3.1, only dependencies mentioning an event are *directly* relevant in scheduling it. In general, we also need to consider events that are caused by events that are caused by the given one, and so on.

The semantic event attributes we use were introduced by Attie *et al.* [1]. Three primitive attributes were identified in [1]:

- *Triggerable* events—called forcible events in [1]—are those that the system can initiate.
- *Rejectable* events are those that the system can prevent.
- *Delayable* events are those that the system can delay.

Nondelayable and nonrejectable events are not really attempted: the scheduler is notified of their occurrence after the fact.

Example 11 Consider tasks such as shown in Figure 1. The scheduler may trigger a *start*, but cannot unilaterally trigger a *commit*. However, it can reject or delay a *commit*, but can neither delay nor reject an *abort*. ∎

The attributes of events constrain the options available to the scheduler. Certain schedules that are otherwise all right may be unacceptable in that they leave the scheduler in a state where it may be forced to violate a dependency. We define a *secure* state as one where this will not be the case. The scheduler can make a transition if the target state is secure.

Example 12 In dependency $D_<$ of Figure 2, if e is *abort*, then the state labeled $\overline{e}$ is not secure. Thus, the transition labeled f from the initial state must be disabled. However, $\overline{f} + f$ is secure. ∎

Example 13 In dependency $D_\rightarrow$ of Figure 2, if f is *start*, then the state labeled f is secure. However, the scheduler must eventually cause f to fire. ∎

For brevity and ease of exposition, we describe how our algebra fits in with the approach of [1], which is based on *pathsets*. Our implemented scheduler is somewhat different, however. It distributes events and arranges appropriate message flows among them. Further details of these messages (but not the algebra) are presented in [15]. The pathset approach maintains the current state of each dependency. It attempts to find a *viable* set of paths, i.e., one for which the following conditions hold:

- each path begins in the current state of some dependency and ends in a secure state;
- events mentioned on the paths occur in the same relative order;
- if an event occurs on a path, then its complement occurs on no path;
- if an event occurs on a path, then the pathset contains a path from each dependency involving that event;
- for every event e mentioned in a path, (a) e is either triggerable or was attempted by its agent or $\overline{e}$ is rejectable and (b) if e was attempted and is not the first event, then e is delayable.

Consequently, if a viable pathset can be found, then all events in it can be scheduled in an appropriate partial order. A viable pathset is tried to be computed whenever an event is attempted or when the scheduler learns that an event occurred somewhere in the system.

Example 14 Using the dependencies of Figure 2, if e is *abort* and f is *start*, then the pathset $\{\langle e\rangle, \langle e\rangle\}$ beginning in the initial states is viable. The scheduler can execute e, but must trigger f later. We can also compute a *trigger-closed* viable pathset, which in this case would be $\{\langle e, f\rangle, \langle e, f\rangle\}$. ∎

5 Discussion and Comparisons

The idea of defining admissible traces and using them as in the above appears novel to our approach. By using admissibility, we are able to obtain a notion of residuation that is more intimately related to eager scheduling of events under dependencies. Our notion of residuation, while closely related to the standard one—e.g., the one of Pratt [11]—is specialized for use in scheduling. The standard notion of residuation does not in any obvious manner yield the kinds of equations we give here.

Since events are not repeated in admissible traces, it might appear that we cannot handle task agents with loops. However, we only require that a given event *instance* not be repeated. New event instances must be presented to the scheduler, e.g., by generating unique event IDs each time an agent executes an event. We restart dependencies appropriately for each relevant instance. We lack the space here to discuss details.

Our treatment of event complementation requires a formal complement for each significant event. An event and its complement are both atoms in the formal language, $\mathcal{E}$; corresponding model-theoretic entities occur in the traces. However, sometimes events are not intuitively thought of as having complements. For example, whereas *abort* and *commit* usually are complements of each other [10], *start* and *forget* usually have no complements at all. A superfluous—in the above sense—formal complement of an event causes no harm, because it is never instantiated. However, a complement is required for every event that is optional. For optional events, there is typically just a two-way branch in their associated task agent, e.g., between *abort* and *commit*. However, when the task agent has a multiway split, then the complement of an event is, in effect, the join of all events that are its alternatives. This is quite rare in practice, since our agents include only *significant* events. However, it can be captured by adding additional dependencies in the obvious manner.

Our approach is syntactically simple, which enables easy requirements capture. Other approaches rely on fine syntactic variations: dependencies with nested $<$ operators, which they require, are often confusing [10, 8]. Our approach involves no unintuitive semantic assumptions, but makes use of every available aspect of the problem to gain expressiveness and efficiency. It generates smaller representations than [1]: e.g., simple compensate dependencies are of size 3 here, but were over size 40 there.

6 Conclusions

We have demonstrated a working prototype based on our theory [16]. Our implemented scheduler is distributed and somewhat different from the pathset approach described above [15]. Our approach has the following merits. It is provably correct. It yields succinct representations for many interesting dependencies that arise in practice, including those representing compensations. The simplicity of our algebra facilitates capturing various requirements. It yields a distributed implementation as easily as a centralized one, an important factor in many heterogeneous, distributed environments.s

Our formal theory and, in particular, our equations for residuation constitute an algebraic approach for scheduling events to satisfy all stated dependencies, assuming those dependencies are mutually consistent and enforceable. The dependencies or specifications are lazy in that they characterize traces as acceptable or unacceptable based on entire computations. However, our approach to scheduling is eager in that it attempts to execute events as soon as possible and accounts for both the conditions under which an event may be executed and the change of state that its execution must bring about. The latter is required so that the resulting computations satisfy the initially specified dependencies. Thus, for the purposes of workflow scheduling, our approach goes well beyond other action algebra approaches.

References

[1] 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.

[2] P. Chrysanthis and K. Ramamritham. ACTA: The SAGA continues. In *[4]*. 1992. Chapter 10.

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

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

[5] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987.

[6] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, September 1994.

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

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

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

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

[11] Vaughan R. Pratt. Action logic and pure induction. In J. van Eijck, editor, *Logics in AI: European Workshop JELIA '90, LNCS 478*, pages 97–120. Springer-Verlag, September 1990.

[12] Marek Rusinkiewicz and Amit Sheth. Specification and execution of transactional workflows. In *[9]*. 1994.

[13] Munindar P. Singh and Michael N. Huhns. Automating workflows for service provisioning: Integrating AI and database technologies. *IEEE Expert*, 9(5), October 1994. Special issue on *The Best of CAIA'94* with selected papers from Proceedings of the 10th IEEE Conference on Artificial Intelligence for Applications, March 1994.

[14] 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.

[15] Munindar P. Singh and Christine Tomlinson. Workflow execution through distributed events. In *Proceedings of the 6th International Conference on Management of Data*, December 1994.

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