Semantical Considerations on Workflows: An Algebra for Intertask Dependencies

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA

singh@ncsu.edu

April 15, 1996

Abstract

Workflows are composite multitransaction activities occurring in heterogeneous environments. They relax the semantic properties of traditional transactions to accommodate the demands of such environments. It is important that workflows be specified declaratively, reasoned about formally, and scheduled automatically. Declarative approaches based on intertask dependencies are prominent in the literature. However, extant approaches often lack a formal semantics, or fail to meet other important criteria. Also, they do not carefully distinguish event types from instances, a distinction that is crucial when the constraint that tasks are loop-free is relaxed. We propose an approach that gives a rigorous formal semantics for dependencies and meets the above conditions. Our approach uses algebraic expressions to represent dependencies and uses symbolic reasoning to take scheduling decisions. It can form the basis of a programming language for workflows.

1 Introduction

Workflows are composite activities that typically involve a variety of computational and human activities, and span multiple systems. They arise naturally in heterogeneous environments. The importance of workflows is increasing rapidly because of market forces. Much research has recently been focused on them [Hsu, 1993]; they are listed as a major database challenge in [Dayal *et al.*, 1993].

The traditional transaction model defines ACID transactions, which have the properties of atomicity, consistency, isolation, and durability. ACID transactions have proven remarkably effective in a number of data processing applications [Gray & Reuter, 1993]. Unfortunately, they are not well-suited to heterogeneous systems. First, atomic commit protocols are inefficient in a distributed environment, and perhaps even impossible if legacy systems cannot be modified. Autonomy of legacy systems is often sacrosanct for technical and political reasons. Legacy systems are also frequently technically closed in that they have no visible precommit state. Second, the semantic requirements in heterogeneous applications are often quite complex and need more sophisticated task structuring than the traditional model offers [Garcia-Molina & Salem, 1987]. A number of extended transaction models have been proposed recently [Elmagarmid, 1992]. Typically, these generalize the ACID model in different ways and do not meld well with each other.

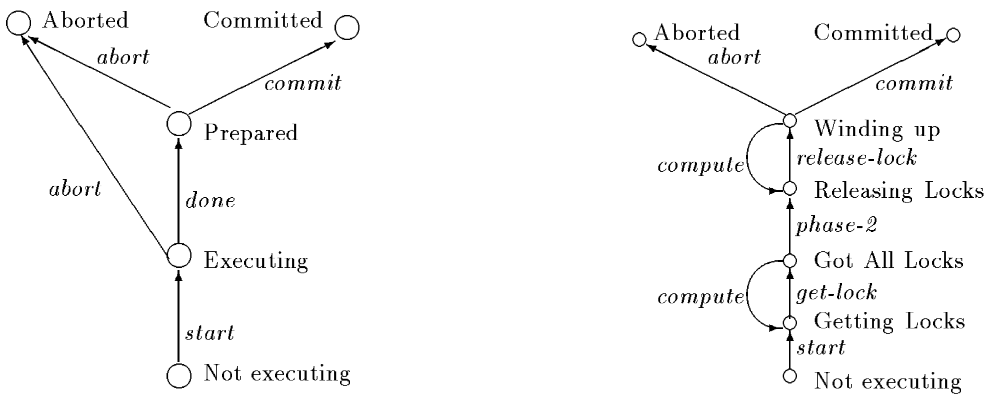
The sheer variety of extended transaction models has led to generic "RISC" approaches that enable the specification of different transaction models in terms of a small number of primitives [Klein, 1991a; Chrysanthis & Ramamritham, 1992; Attie *et al.*, 1993]. These approaches do not offer new transaction models *per se*, but instead provide declarative *intertask dependencies* to specify workflows. Major issues in this paradigm are (a) how to express dependencies, and (b) how to schedule events to satisfy them.

The present work is in this paradigm: we too provide a formal language for specifying intertask dependencies. However, we give a formal *Tarskian semantics* for our language. Our approach addresses both issues (a) and (b) above. It is superior to previous approaches in each respect. First, our semantics meets certain criteria that are crucial to a specification language—it is compositional, provides a notion of correctness, associates a notion of strength with different specifications, and carefully distinguishes between event types and instances. Second, our semantics has key features crucial to scheduling—it can encode the knowledge of the scheduling system, and make decisions on events through efficient *symbolic* reasoning. Because we have a rigorous semantics based on event traces, we are able to derive stronger results than were previously obtained. Further, whereas previous approaches are limited to tasks that never loop over their significant events (see section 2), our approach applies to arbitrary tasks.

This paper seeks to put the specification and scheduling of workflows on a sound footing, in a model-theoretic sense. Although several transaction models have been proposed, there has not been sufficient theoretical work on the nature of the computations involved. This paper provides rigorous definitions of workflow computations, and shows how those definitions may be used to specify workflows and to formally reason about their properties. The proposed approach makes a clear distinction between event types and instances, which is not only crucial in formulating correct workflow specifications, but is also key to relaxing the requirement that tasks be loop-free. Some implementational aspects were reported in [Singh & Tomlinson, 1994]; however, the present paper gives the first description of the underlying theory.

Section 2 gives an overview of the recent research in the area. Section 3 presents our event algebra for representing and reasoning about dependencies, taking care to highlight our key motivations and assumptions. Section 3 also exhibits a carefully engineered set of equations by which a scheduler can symbolically reason about dependencies and shows how these can be used in scheduling. Section 4 proves the soundness of our equations in a natural class of models that are motivated therein. Section 5 extends our technical development to apply to arbitrary tasks. Section 6 discusses additional technical properties crucial for the specification and scheduling of workflows that motivated our approach.

2 Overview of the Literature



Handled by most approaches **Not handled by other approaches**

Figure 1: Example task agents handled by our approach

A common theme in the literature is the notion of *significant* events. Tasks being coordinated are modeled in terms of events that are significant for the purposes of coordination. The tasks are interfaced to the scheduler through proxy *agents*. Although the tasks may be complex, their interaction with the system is in terms of significant events, which are captured by the proxy agents. The agents are assumed to behave like finite state automata over a (usually small) set of states. Figure 1 shows two such agents—previous approaches can handle only the loop-free one. One of the pictures in Figure 1 is of an automaton with cycles. For expository reasons, we use this automaton, which exhibits a possible locking behavior of a transaction. The key idea here is the looping. This is essential to capture the reasoning in the ACTA example mentioned below, but which cannot be handled in ACTA. The same structure arises in resource managers in general, because these typically have to repeatedly react to stimuli. Our mathematical approach can apply to events at several levels of granularity; its implementation, however, is practical only for coarse-grained events of the kind considered below.

The significant events label the legal transitions among the states of the automata. Typically, they correspond to transaction manager or operating system primitives, such as *begin*, *commit*, *abort*, *spawn*, and *fail*—these depend on the underlying transaction model. Each state potentially hides a complex computation. Intertask dependencies are constraints across the significant events of different tasks. The satisfaction of dependencies can require the (non-)occurrence and ordering of various events.

ACTA ACTA provides a formal framework to specify the effects of transactions on other transactions and on objects, the latter via *object events* [Chrysanthis & Ramamritham, 1992]. An execution of a transaction is a partial order—denoting temporal precedence—of the events of that transaction (the object events it invokes, plus its significant events). A history of a concurrent execution of a set of transactions contains all events of each of the transactions, along with a partial order that is consistent with the partial orders for the individual transactions. The occurrence of events in a history is denoted explicitly through formulas like $e \in H$. The "predicate" $e \longrightarrow e'$ means that e precedes e' (implicitly in history H). It requires that e and e' occur in H. ACTA restricts its dependencies to finite histories. Also, it does not address scheduling issues.

Whereas ACTA provides a formal syntax, it does not provide a formal *semantics* in the model-theoretic sense. An important semantic issue from our standpoint is the distinction between event types and instances. The formal definitions appear to involve event instances, because they expect a partial order of the events. However, certain usages of the formalism are less clear. For example, consider the intuitive statement that "(when t_i reads a page x that t_j subsequently writes), if t_j commits before t_i, t_i must reread x after t_j commits." ACTA captures this statement by the following formula [Chrysanthis & Ramamritham, 1992, (p. 363)]: $(read_{t_i}[x] \longrightarrow write_{t_j}[x]) \Rightarrow ((Commit_{t_j} \longrightarrow Commit_{t_i}) \Rightarrow (Commit_{t_j} \longrightarrow read_{t_i}[x]))$. This formula clearly uses $read_{t_i}[x]$ to refer to two different event instances, one before $Commit_{t_j}$, and the other after $Commit_{t_j}$.

Rule-Driven Transaction Management Another contribution is [Klein, 1991a; Klein, 1991b]. Klein proposes two primitives for defining dependencies. In Klein's notation, $e \rightarrow f$ means that if e occurs then f also occurs (before or after e). His $e < f$ means that if both events e and f happen, then e precedes f. This work describes a formalism and its intended usage, but even the longer version [Klein, 1991b] gives no formal semantics. The semantics is informally explained using *complete histories*, which are those in which every task has terminated. Further, it is assumed that tasks are expressible as loop-free regular expressions. Thus this approach is not applicable to activities that never terminate, or those that iterate over their significant events before terminating.

Temporal Logic Approaches: Branching and Linear Our previous approach [Attie *et al.*, 1993] is based on a branching-time temporal logic, CTL (or computation tree logic [Emerson, 1990]). This approach formalizes dependencies in CTL and gives a formal semantics. It synthesizes finite state automata for the different dependencies. To schedule events, it searches for an executable, consistent set of paths, one in each of the given automata. This avoids computing product automata, but the individual automata in this approach can be quite large. Further, the CTL representations of the common dependencies are quite intricate. This implementation was centralized. Günthör's [1993] approach is based on linear temporal logic, and gives a formal semantics. His implementation too is centralized and his approach appears incomplete.

Action Logic In view of the limitations of prior approaches, we looked into Pratt's approach for expressing and reasoning about events [1990]. Pratt's work is not specially designed for specifying workflows, but is relevant as a general-purpose theory of events. Our approach can be seen as an enhancement of it for reasoning about workflows. Pratt proposes an algebra, and motivates a number of potentially useful inferences that can be drawn in it. Using these inferences, he constrains the possible models for the algebra. One class of models he considers are the regular languages. These correspond closely to linear histories of events. The branching (partially ordered) histories well-known from serializability theory can be expressed as sets of linear histories [Bernstein *et al.*, 1987]. We find this connection fruitful: since linear histories are easier to deal with formally, we adopt a similar model ourselves.

Remarks on the Above Approaches The database approaches above are nice in different respects, but are either informal and possibly ambiguous, or not accompanied by distributed scheduling algorithms. ACTA and Klein's approaches are noncompositional, since the semantics they assign to a formula is not derived from the semantics of its operands. This makes it difficult to reason symbolically. The restriction to loop-free tasks and the lack of an explicit distinction between event types and instances are the most limiting properties of all four approaches. However, these approaches agree on the *stability* of events—an event once occurred is true forever. This is a natural intuition, and one that we preserve for event instances. Pratt's approach is formal and compositional, but lacks a scheduling algorithm. Further, it too lacks an explicit distinction between event types and instances. Pratt defines an operator for *residuation*, but his equations are too weak to apply in scheduling.

We reviewed the formal literature on intertask dependencies. Other related work includes [Garcia-Molina *et al.*, 1990; Wächter & Reuter, 1992; Nodine, 1993; Breitbart *et al.*, 1993; Agrawal *et al.*, 1993].

3 Our Approach: Event Algebra

Our formal language is based on an algebra of event types, which is related to the action logic of [Pratt, 1990]. However, we make crucial enhancements to Pratt's syntax (complement events) and semantics (admissibility) to derive our key results. In our presentation, we carefully isolate admissibility from the basic semantics and motivate it carefully to show exactly where it is necessary. Event symbols are the atoms of our language. Unlike Pratt, we also introduce for each original event symbol another symbol corresponding to its complement. Lastly, we allow the events to be parameterized so as to state constraints for workflows whose constituent tasks may be nonterminating. Previous approaches are limited to single-shot tasks. However, for ease of exposition, we defer parameterization of events to section 5.

3.1 Motivations and Key Assumptions

Besides the features already discussed, the property that most guided our approach was the wish to permit declarative specifications, yet schedule events eagerly. Specifications should be declarative in that they should describe the conditions that must hold over entire computations, without regard to how an acceptable schedule may be generated. This accords well with the spirit of declarative specifications. However, the execution mechanism should be able to allow or trigger events on the basis of whatever information is available at the given stage of the computation. A good approach should encode and process this information as well.

3.2 Syntax and Semantics

$\mathcal{E}$, the language of event expressions has the following syntax. Σ is the set of significant event symbols; Γ is the *alphabet*, which consists of events and their complements; Ξ contains atomic event symbols. A *dependency* or an expression is a member of $\mathcal{E}$. A *workflow* is a set of dependencies.

Syntax 1 $\Gamma = \{e, \overline{e} : e \in \Sigma\}$

Syntax 2 $\Xi = \Gamma$

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

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

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

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

Syntax 7 $E_1, E_2 \in \mathcal{E}$ implies that $E_1 | E_2 \in \mathcal{E}$

Intuitively, Γ and Ξ refer to events (they will be differentiated in section 5). Thus e means that e occurs somewhere. The constant 0 refers to a specification that is always false; $\top$ refers to one that is always true. The operator $+$ means disjunction. The operator $|$ means conjunction; $|$ thus refers to the interleaving of its arguments. The operator $\cdot$ refers to sequencing of its arguments.

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 (a) to specify desirable computations and (b) to determine event schedules to realize good computations. For convenience, we overload event symbols with the events they denote. Our usage is always unambiguous. Traces are written as event sequences enclosed in $\langle$ and $\rangle$ brackets. Thus $\langle e\overline{f}\rangle$ means the trace in which event e occurs followed by the event $\overline{f}$. $\lambda \triangleq \langle\,\rangle$ is the empty trace. (Throughout, $\triangleq$ means *is defined as*.)

Let $\mathbf{U}_\Gamma \triangleq \Gamma^* \cup \Gamma^\omega$ be our universe. This consists of all possible (finite and infinite) traces over Γ. For a trace, $\tau \in \mathbf{U}_\Gamma$, and an expression $E \in \mathcal{E}$, $\tau \models E$ means that τ satisfies E. $[\![\,]\!]$ gives the *denotation* of an expression: $[\![E]\!] \triangleq \{\tau : \tau \models E\}$.

Semantics 1 $[\![f]\!] = \{\tau \in \mathbf{U}_\Gamma : \tau \text{ mentions } f\}$, $f \in \Xi$

Semantics 2 $[\![0]\!] = \emptyset$

Semantics 3 $[\![\top]\!] = \mathbf{U}_\Gamma$

Semantics 4 $[\![E_1 \cdot E_2]\!] = \{\upsilon\tau \in \mathbf{U}_\Gamma : \upsilon \in [\![E_1]\!] \text{ and } \tau \in [\![E_2]\!]\}$

Semantics 5 $[\![E_1 + E_2]\!] = [\![E_1]\!] \cup [\![E_2]\!]$

Semantics 6 $[\![E_1|E_2]\!] = [\![E_1]\!] \cap [\![E_2]\!]$

Thus the atom e denotes the set of traces in which event e occurs. $E_1 \cdot E_2$ denotes memberwise concatenation of the denotations 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 $\Gamma = \{e, \overline{e}, f, \overline{f}\}$ be the alphabet. Then the denotation of e is given by, $[\![e]\!] = \{\langle e\rangle, \langle e\overline{e}\rangle, \langle ef\rangle, \langle fe\rangle, \langle e\overline{f}\rangle, \langle ee\overline{e}f\overline{f}\rangle, \ldots\}$. The denotation of $e \cdot f$, $[\![e \cdot f]\!] = \{\langle ef\rangle, \langle e\overline{e}f\rangle, \langle eff\overline{e}\rangle, \langle fee\overline{e}f\overline{f}\rangle, \ldots\}$. Similarly, the denotation of $e|f$, $[\![e|f]\!] = \{\langle ef\rangle, \langle fe\rangle, \langle e\overline{e}f\rangle, \langle ff\overline{e}e\rangle, \langle fe\overline{e}f\overline{f}\rangle, \ldots\}$. One can readily verify that $[\![e + \overline{e}]\!] \neq [\![\top]\!]$ and $[\![e|\overline{e}]\!] \neq [\![0]\!]$. ∎

Definition 1 We write $E \equiv F$ iff $[\![E]\!] = [\![F]\!]$. Note that this is only a convenient abbreviation: $\equiv$ is not an operator in our object language, $\mathcal{E}$.

Observation 1 $\tau \in [\![E]\!]$ iff $(\forall \upsilon, \nu : \upsilon\tau\nu \in \mathbf{U}_\Gamma \Rightarrow \upsilon\tau\nu \in [\![E]\!])$ ∎

Observation 1 means that if a trace satisfies E, then all larger traces do so too. Conversely, if all traces that include τ satisfy E, then τ satisfies E too (essentially by setting υ and ν to λ).

Definition 2 D is a sequence expression $\triangleq$ $D = e_1 \ldots e_n$, where each $e_i \in \Xi$.

Observation 2 If $D = e_1 \ldots e_n$ is a sequence expression, then $\tau \in [\![D]\!]$ implies that $(\forall i : 1 \leq i \leq n \Rightarrow \tau \in [\![e_i]\!])$ ∎

Observation 2 means that traces that satisfy a sequence expression form a subset of the traces that satisfy any of the events in that sequence expression. Intuitively, if a trace satisfies "do a, then b," it also satisfies "do a" and "do b."

We treat e and $\overline{e}$ symmetrically as events. We thus define a formal complement for each event, including those like *start*, whose non-occurrence when established constitutes their complement. This is crucial for eager scheduling—section 6 has more rationale.

We must give some additional semantic intuitions before we can apply the above development to a concrete example. Our approach is based on residuation in our algebra. As explained in section 3.3, residuation has the right semantic properties to formalize the behavior of a scheduler. However, to obtain the necessary independence and modularity properties, and closed-form answers for symbolic reasoning, we need stronger equations than Pratt's. We discovered that our equations were not sound in any of the usual models! We realized that this was because the usual models lacked an explicit *combination* of a notion of change—of the system evolving because of events—and a notion of the system's knowledge—its state for scheduling decisions. Soundness was easily proved when we added that combination to the usual regular language models. The notion of change is captured through our stronger equations for residuation, and the notion of knowledge through a quotient construction, which identifies expressions that are equivalent with respect to the desired behavior of the scheduler. Our key assumptions are as follows:

Assumption 1 An event instance excludes its complementary event instance from any computation.

Assumption 2 An event instance occurs at most once in any computation.

Assumption 3 An event instance or its complement eventually occurs in each computation.

Traces that satisfy assumptions 1 and 2 are termed *legal*. Our universe set (and Example 1) includes illegal traces, but these are eliminated from our formal model in section 4. Intuitively, the reader should assume legality everywhere.

Example 2 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 transactions (as in Figure 1, left). For simplicity, assume that *cancel* always commits. Now the desired workflow may be specified as (D1) $\overline{s_{buy}} + s_{book}$ (if *buy* starts, then *book* must also start); (D2) $\overline{c_{book}} + \overline{c_{buy}} + c_{book} \cdot c_{buy}$ (*book* commits before *buy* if both commit); (D3) $\overline{c_{buy}} + c_{book}$ (*buy* commits only if *book* commits); (D4) $\overline{c_{book}} + c_{buy} + s_{cancel}$ (compensate *book* by *cancel*—i.e., if *book* commits and *buy* aborts, then *cancel* must start); and (D5) $\overline{s_{cancel}} + \overline{c_{buy}} | c_{book}$ (start *cancel* only if *book* commits and *buy* aborts).

The above workflow is satisfied by several legal traces, including (τ_1) $s_{buy} s_{book} c_{book} c_{buy}$, ($\tau_2$) $s_{buy} s_{book} \overline{c_{buy}}\, \overline{c_{book}}$, ($\tau_3$) $s_{buy} s_{book} c_{book} \overline{c_{buy}} s_{cancel} c_{cancel}$, ($\tau_4$) $\overline{s_{buy}}\, \overline{s_{book}}\, \overline{s_{cancel}}$, and ($\tau_5$) $s_{book} s_{buy} c_{book} c_{buy}$. ▮

3.3 Scheduling by Residuation

Events are scheduled—by permitting or triggering tasks to proceed—to satisfy all stated dependencies. A dependency is satisfied when a trace in its denotation is realized. We characterize the state of the scheduler by the traces it can allow. The allowable traces are given initially by the dependencies in the stated workflow. As events occur, the allowed traces get narrowed down. Let us consider the processing informally first.

Example 3 Consider Example 2. Suppose *buy* starts first. Then D1 requires that *book* start sometime; the other dependencies have no effect, since they don't mention s_{buy} or $\overline{s_{buy}}$. Now if *buy* were to commit *next*, D2 would prevent *book* from committing, yet D3 would require *book* to commit. Because of this inconsistency, *buy* cannot commit next. However, *book* can start right after the start of *buy*, thereby satisfying the remaining obligation from D1. After *book* starts, *buy* would still not be allowed to commit, but *book* will be allowed to commit. After *book* commits, *buy* can commit thus completing the workflow, or *buy* can abort thus causing *cancel* to be started.

The above assumes that *buy* starts first. The dependencies we stated assume that *buy* is the task that is triggered from outside, and gets the whole workflow rolling. If we wish to allow *book* to play the same role, we have to state an additional dependency in the reverse direction of D1. Simplicity of exposition is the only reason why don't do that here. ▮

The above example shows that while scheduling a workflow, two questions must be answered for each event that is considered for scheduling:

- can it happen now?
- what will remain to be done later?

Clearly, the answers can be determined given the original dependencies plus the history of the system so far. One can examine the traces allowed by the original dependencies, select those compatible with the actual history, and infer how they might proceed. Most interestingly, in our approach, both above questions are answered by symbolic reasoning. Figures 2 and 3 show how the state of the scheduler may be represented symbolically and how transitions among the various states may also be captured symbolically. These figures are based on dependencies D1 and D2 of Example 2, which were exercised in Example 3. The state labels correspond to the current obligation of the scheduler and the transition labels to the event whose occurrence causes the obligation to change. In the case of Figure 2, the scheduler is initially obliged to the entire dependency D1. If s_{book} happens the obligation changes to $\top$, meaning that D1 is satisfied. If s_{buy} happens, it changes to s_{book}, meaning that s_{book} must happen. Roughly, an event that would make the scheduler obliged to 0 cannot occur. Figure 3 is similar.

The transitions exhibited by Figures 2 and 3 can be captured through an algebraic operator that we introduce. This operator is called *residuation*. It is not added to our formal object language, since it is not used in the formulation of dependencies, only in their processing. Dependencies are *residuated* by the events that occur to yield simpler

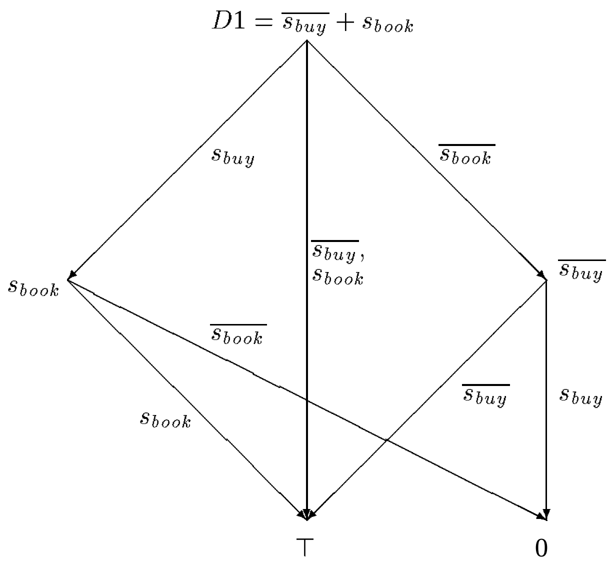


Figure 2: Scheduler transitions for dependency D1

dependencies. The resultant dependencies implicitly contain the necessary history. This proves highly efficient, because the representations required are small and the processing is simple. The residuation operator $/$ is given the usual semantics (see section 4), but a stronger equational characterization. We present this next. We use the operator $\doteq$ in stating the equation so as to highlight that it might not be interpreted simply as $\equiv$. Section 4 provides further explanation of this.

In the following, we assume that the formulas being residuated are in conjunctive normal form (CNF). CNF for our language $\mathcal{E}$ refers to a normal form in which the conjunction is applied at the outermost level; disjunction is applied at the next level in; and, sequencing at the innermost level. The literals are event symbols, complemented or otherwise. Because of the restriction to CNF, we can assume that in the equations below, D is a sequence expression, and E is a sequence expression or $\top$ (the latter case allows us to treat a single atom as a sequence, using $f \equiv f \cdot \top$). $\Gamma_D \triangleq \{e, \overline{e} : e$ is mentioned in $D\}$—essentially the alphabet restricted to D.

Equation 1 $0/e \doteq 0$

Equation 2 $\top/e \doteq \top$

Equation 3 $(E_1|E_2)/e \doteq ((E_1/e)|(E_2/e))$

Equation 4 $(E_1 + E_2)/e \doteq (E_1/e + E_2/e)$

Equation 5 $(e \cdot E)/e \doteq E$, if $e, \overline{e} \notin \Gamma_E$

Equation 6 $D/e \doteq D$, if $e, \overline{e} \notin \Gamma_D$

Equation 7 $(e' \cdot E)/e \doteq 0$, if $e \in \Gamma_E$

Equation 8 $(e' \cdot E)/e \doteq 0$, if $\overline{e} \in \Gamma_E$

The above equations are can be used to symbolically calculate the automata corresponding to different dependencies. Thus, these automata need not be explicitly represented. For a given state D of the scheduler and an event e, D/e gives the resulting state.

The above equations are carefully designed to guide the reasoning of a scheduler. They take advantage of the assumptions of section 3.1. These equations have some useful properties—dependencies not mentioning an event have

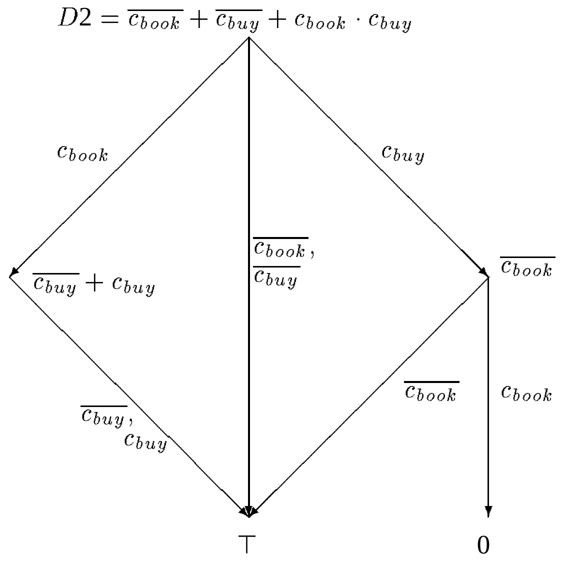

$D2 = \overline{c_{book}} + \overline{c_{buy}} + c_{book} \cdot c_{buy}$
c_{book}
c_{buy}
$\overline{c_{book}}$, $\overline{c_{buy}}$
$\overline{c_{buy}} + c_{buy}$
$\overline{c_{book}}$
$\overline{c_{buy}}$, c_{buy}
$\overline{c_{book}}$
c_{book}
$\top$
0

Figure 3: Scheduler transitions for dependency D2

no direct effect on it; the reasoning with respect to different dependencies can be performed modularly; and, the history of the scheduler need not be recorded explicitly. The dependencies stated in a workflow thus fully describe the state of the scheduler; successive states are computed symbolically through residuation. We now formalize the reasoning described in Example 3.

Example 4 Consider Example 2 again. The initial state of the scheduler is given by $S_0 = D1|D2|D3|D4|D5$. The scheduler can allow *buy* to start first, because the resulting state $S_1 = S_0/s_{buy}$ is consistent. This simplifies to $S_1 = s_{book}|D2|D3|D4|D5$. The occurrence of c_{buy} next would leave the scheduler in state $s_{book}|\overline{c_{book}}|c_{book}|\top|\overline{s_{cancel}}$, which is inconsistent because $\overline{c_{book}}|c_{book} = 0$ (both cannot occur in any legal trace). However, since $S_2 = S_1/s_{book} = \top|D2|D3|D4|D5$ is consistent, s_{book} can occur in S_1. In S_2, *book* can commit, resulting in the state $S_3 = \top|(\overline{c_{buy}} + c_{buy})|\top|(c_{buy} + s_{cancel})|(\overline{c_{buy}} + \overline{s_{cancel}})$. S_3 allows either of $\overline{c_{buy}}$ and c_{buy} to occur. Since $S_3/c_{buy} = \overline{s_{cancel}}$, the workflow completes if *buy* commits. Since $S_3/\overline{c_{buy}} = s_{cancel}$, *cancel* must be started if *buy* aborts. ∎

A significant advantage of considering CNF is the following. The dependencies are independently stated and effectively conjoined to define a workflow (a set of dependencies can be treated as the conjunction of its members). If CNF is assumed, then the conjunction of several dependencies essentially reduces to one (albeit large) dependency. Consequently, no additional processing is required in putting the dependencies into an acceptable syntactic form for reasoning. Also, Equation 3 enables the different conjuncts or different dependencies to be residuated independently. It is also simple to establish the following result, which guarantees independence of the residuation of dependencies by events that they do not mention.

Observation 3 $E/f \doteq E$, if $f \notin \Gamma_E$ ∎

Event scheduling depends on the 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. By Observation 3, only dependencies mentioning an event are *directly* relevant in scheduling it. Of course, we also need to consider events that are caused by events that are caused by the given one, and so on—these might be involved in other dependencies.

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. This paper does not focus on these operational aspects of scheduling. Some of those, which also involve distribution, are reported in [Singh, 1996]. A distributed scheduler based on the results of this paper has been implemented. It is best suited to coarse granularity events and has been used to control the invocation of tasks to satisfy a given workflow.

4 Soundness Through Admissibility

The above equations can prove quite effective in scheduling. However, we must also show that they are sound in some natural models that reflect our key intuitions about computations and dependencies. For our trace-related intuitions, a regular language model is quite natural. In such a model, the standard definition of residuation accords well with our intuition of / as an operator that tells us what remains from a dependency after an event occurs. Indeed, residuation provides the weakest possible interpretation for /, so we formally define / as follows:

Semantics 7 $\nu \in [\![D/e]\!]$ iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_\Gamma \Rightarrow \upsilon\nu \in [\![D]\!]))$

The above definition is a variation of one given in [Pratt, 1990]. Now we talk about the soundness of our equations.

Definition 3 An equation $D/e \doteq F$ is sound iff $[\![D/e]\!] = [\![F]\!]$

The above definition of soundness essentially interprets $\doteq$ as $\equiv$, i.e., equivalence. As indicated in section 3.1, some of our equations are *not* sound in the regular language model. They turned out not to be sound in a number of other standard models we experimented with. However, some are easily shown to be sound in our model of choice, the regular language model introduced above.

Lemma 4 Equations 1 through 5 are sound. ■

But what about Equations 6, 7, and 8? These equations are especially troublesome to prove sound in typical models. Individually, they can be satisfied in different models, but not together. Finding a model-theoretic characterization of the equations thus became a major technical challenge in our approach. We addressed this by defining a class of nonstandard, but intuitively natural, models based on what we term *admissibility*.

Admissible traces are those that satisfy assumptions 1, 2, and 3 of section 3.1. Intuitively, these traces characterize the maximal (and legal) behavior of the environment, i.e., of the given collection of tasks. Let $\mathbf{A}_\Theta$ be the set of all admissible traces on the alphabet Θ, where $e \in \Theta$ iff $\overline{e} \in \Theta$. As before, we identify $\overline{\overline{e}}$ with e.

Intuitively, admissibility is designed to formalize our special method of applying residuation to scheduling decisions. Consider Equation 7. Assume the scheduler is enforcing a dependency $D = (e' \cdot E)$, where E is a sequence expression mentioning e. Suppose the scheduler is taking a decision on e. We know that e' has not occurred yet, or it would have been residuated out already. Therefore, if we let e happen now, then either we must (a) prevent e', or (b) eventually let e' happen followed by another instance of e. Option (a) clearly violates D, since all traces in $[\![D]\!]$ must mention e' (by Observation 2). Option (b) violates the assumption that events are not repeated. Admissibility precludes option (b), and thus helps prove the soundness of Equation 7. We formalize this concept next.

Technically, the way in which admissibility operates is in capturing the context of evaluation, given by the state of the scheduler. Two expressions are interchangeable with respect to a set of admissible traces A if they allow exactly the same subset of A. As a result, two expressions that have different denotations may end up being interchangeable in certain evaluation contexts.

Example 5 In general $e \not\doteq 0$. However, after e or $\overline{e}$ has occurred, we know that another occurrence of e is impossible. Hence, we can set e equivalent to 0. ■

Definition 4 A is an admissible set iff $A = \mathbf{A}_\Theta$ for some alphabet Θ

Initially the admissible set is the entire set of admissible traces for Γ. After event e happens, the admissible set must be shrunk to exclude traces mentioning e or $\overline{e}$, because they cannot occur any more. Let A be an admissible set before e occurs. Then, after e, A must be replaced by $A{\restriction}e$, where $A{\restriction}e$ yields the resulting set of admissible traces. The operator ${\restriction}$ thus abstractly characterizes execution. This is the notion of change described in section 3.1. Formally,

Definition 5 $A{\restriction}e \triangleq \{\nu : e\upsilon \in A\}$

We use admissible sets to define a coarser notion of equivalence ($\approx_A$) than equality of denotations ($\equiv$).

Definition 6 $E_1 \approx_A E_2 \triangleq A \cap [\![E_1]\!] = A \cap [\![E_2]\!]$, for any admissible set A.

Example 6 Let $A = \mathbf{A}_\Theta$ for some alphabet Θ. Then, $e + \overline{e} \approx_A \top$ (since A is maximal). Also, $e|\overline{e} \approx_A 0$ (since A is consistent). ▌

Example 7 Let $A = \mathbf{A}_\Theta$ for some alphabet Θ, such that $e \notin \Theta$. Then, $e \approx_A \overline{e}$. Also, $e \approx_A 0$. ▌

Lemma 5 For all admissible sets A, $\approx_A$ is an equivalence relation. ■

We refer to $\approx_A$ as *adm-equivalence*. We now use it to define a quotient structure on our original models. The quotient model preserves all equalities, but only some of the inequalities. That is, for all A, $E_1 \equiv E_2 \Rightarrow E_1 \approx_A E_2$, but $E_1 \approx_A E_2 \not\Rightarrow E_1 \equiv E_2$. This is as it should be: otherwise, there would be no reason for defining the quotient construction!

The fact that we have some equivalence relation does not entail that we can use it to define a viable quotient structure over the states of a scheduler. For adm-equivalence to define a viable quotient structure, we must also establish that the behavior of the scheduler is not affected by replacing one adm-equivalent expression for another. This is a crucial requirement upon which our whole technical development hinges. Fortunately, it can be met.

Theorem 6 Let $E \approx_A E'$. Then, for all $f \in \Gamma$, $E/f \approx_{A\upharpoonright f} E'/f$. ■

Theorem 6 states that if E and E' are adm-equivalent, then after event f occurs, their respective residuals will also be adm-equivalent. In other words, the ongoing behavior of the scheduler cannot be affected by substituting E' for E. That is, event occurrences are well-behaved with respect to the quotient. This is crucial in justifying the following formal notion of soundness, *adm-soundness*, which (in contrast to Definition 3) uses adm-equivalence instead of equality. This notion also accommodates the change of state implicit in the occurrence of events.

Definition 7 $D/e \doteq F$ is adm-sound iff for all admissible sets A, $D/e \approx_{A\upharpoonright e} F$

We have thus established adm-soundness as a reasonable formal notion of correctness for our equations. Since $\approx_A$ is reflexive, we also have the following.

Lemma 7 If $D/e \doteq F$ is sound, then $D/e \doteq F$ is adm-sound. ■

Lemma 8 Equations 6 through 8 are adm-sound. ■

Theorem 9 Equations 1 through 8 are adm-sound. ■

5 Arbitrary Tasks

Our approach as described so far may appear to resemble traditional approaches in assuming that events do not occur more than once. However, a crucial observation is that event *tokens* do not occur more than once; an event *type* may be instantiated multiple times. This assumption can be readily accommodated in our approach. No conceptual enhancement to what was presented above is necessary.

In our architecture, we assume that each task is associated with a number of significant event types. (In our implementation, each event type is implemented as one actor or object.) Each event type is associated with at most one task. Thus commit of *buy* is a different event type than the commit of *book*. Whereas event types are represented, specific event tokens are attempted or triggered. Event tokens are instantiated from event types through parametrization—a tuple of all relevant parameters is attached to each event atom. Thus, event symbols are interpreted as types. When dependencies are stated, some of the parameters can be variables, which are implicitly universally quantified. When events are scheduled, all parameters must be constants. The parameters must be chosen so that event tokens are uniquely determined. Typical parameters include transaction and task IDs, database keys, timestamps, and so on. We can uniquely identify each event token by combining its task ID with the value of a monotonic counter that records the total number of significant event tokens of that task that have been instantiated. Event IDs are reminiscent of *operation IDs* in transaction processing—[Gray, 1981] describes how operation IDs can ensure uniqueness of logged operations in a recovery protocol. It is interesting that this old idea can be adapted to workflows and used in expressing and reasoning about intertask dependencies.

We modify the syntax of $\mathcal{E}$ to parametrize event atoms by attaching a tuple of all relevant parameters. Thus, we replace Syntax 1 and 2 by rules Syntax 8 and 9 given below. Here we assume a set $\mathcal{V}$ of variables and a set $\mathcal{C}$ constants that can be used as parameters. (For simplicity, we do not use multiple sorts for $\mathcal{V}$ and $\mathcal{C}$.) Thus, Γ includes all (ground) event literals and Ξ includes all event atoms. Since the universe depends on Γ, it is automatically redefined to include all traces formed from all possible event tokens that may be instantiated from the available event types using the available parameters. Let $\delta(e)$ give the *degree* of e, i.e., the number of parameters e seeks to become an event token.

Syntax 8 $e \in \Sigma$, $\delta(e) = m$, and $p_1, \ldots, p_m \in \mathcal{C}$ implies that $e[p_1 \ldots p_m]$, $\overline{e}[p_1 \ldots p_m] \in \Gamma$

Syntax 9 $e \in \Sigma$, $\delta(e) = m$, and $p_1, \ldots, p_m \in (\mathcal{V} \cup \mathcal{C})$ implies that $e[p_1 \ldots p_m]$, $\overline{e}[p_1 \ldots p_m] \in \Xi$

The semantics treats the variables as implicitly universally quantified. We replace Semantics 1 by the following two rules. Only Semantics 9 applies to an expression containing any variables—this is where the universal quantification takes place. Here $E(v)$ refers to an expression free in variable v (it may also be free in other variables). $E(v ::= c)$ refers to the expression obtained from $E(v)$ by substituting every occurrence of v by constant c.

Semantics 8 $[\![f[p_1 \ldots p_m]]\!] = \{\tau \in \mathbf{U}_\Gamma : \tau \text{ mentions } f[p_1 \ldots p_m]\}$, $f[p_1 \ldots p_m] \in \Gamma$

Semantics 9 $[\![E(v)]\!] = \bigcap_{c \in \mathcal{C}} [\![E(v ::= c)]\!]$

We assume that events from the same task have the same variable parameters. This is because our focus is on intertask dependencies. Although our formal language allows arbitrary expressions, the ones we can meaningfully (i.e., in an architecturally felicitous manner) interpret and schedule are those that satisfy the above restriction. This restriction forces all reasoning pertinent to individual tasks to be performed by the tasks or their agents. The scheduler only handles the intertask aspects. Thus the events and their parameters define a clean interface between the tasks and the scheduler.

We now consider two different ways of scheduling parametrized dependencies to handle intra-workflow and inter-workflow requirements. In the simplest case, parameters are used *within* a given workflow to relate events in different tasks. Typically, the same variables are used in parameters on events of different tasks. Attempting some key event binds the parameters of all events, thus instantiating the workflow afresh. The workflow is then scheduled as described in previous sections. We redo Example 2 below.

Example 8 Now we use t as the trip or reservation id to parametrize the workflow. The parameter t is bound when the *buy* task is begun. The explanations are as before—now we are explicit that the same customer features throughout the workflow. The desired workflow may be specified as (D1′) $\overline{s_{buy}[t]} + s_{book}[t]$; (D2′) $\overline{c_{book}[t]} + \overline{c_{buy}[t]} + c_{book}[t] \cdot c_{buy}[t]$; (D3′) $\overline{c_{buy}[t]} + c_{book}[t]$; (D4′) $\overline{c_{book}[t]} + c_{buy}[t] + s_{cancel}[t]$; and (D5′) $\overline{s_{cancel}[t]} + \overline{c_{buy}[t]} | c_{book}[t]$.

Let t be bound by various natural numbers. The above workflow is satisfied by an infinite number of legal traces, including the following nonmaximal ones: (τ_6) $s_{buy}[65] s_{book}[65] c_{book}[65] c_{buy}[65]$, ($\tau_7$) $\overline{s_{buy}[34]}\, \overline{s_{book}[34]}\, \overline{s_{cancel}[34]}$, and ($\tau_8$) $s_{buy}[78] s_{book}[34] s_{buy}[34] c_{book}[34] s_{book}[78] c_{book}[78] c_{buy}[78]\ c_{buy}[34]$. The traces τ_6 and τ_7 are as before but with explicit parameters. Trace τ_8 shows how different instantiations of the workflow may interleave. ▌

In the second class of problems, the different events may have unrelated variable parameters. Such cases occur in the specification of concurrency control requirements *across* workflows or transactions.

Example 9 Let the b_i event denote a task T_i's entering its critical section and the e_i event denote T_i's exiting its critical section. Then, mutual exclusion between tasks T_1 and T_2 may be formalized as follows by stating that if T_1 enters its critical section before T_2, then T_1 exits its critical section before T_2 enters. We also state that whenever T_1 enters its critical section, then it also eventually exits it. For simplicity, we ignore the converse requirement, which applies if T_2 enters its critical section before T_1.

$D_M(x, y) = (b_2[y] \cdot b_1[x] + \overline{e_1[x]} + \overline{b_2[y]} + e_1[x] \cdot b_2[y]) | (\overline{b_1[x]} + e_1[x])$

By Semantics 9, the above dependency is interpreted as $(\forall x, y : D_M(x, y))$ Suppose that $b_1[\hat{x}]$ for a specific and unique $\hat{x}$ occurs. This instantiates and residuates the above expression to $(\overline{e_1[\hat{x}]} + \overline{b_2[y]} + e_1[\hat{x}] \cdot b_2[y]) | (e_1[\hat{x}])$. Thus the overall dependency becomes $(\forall x, y : x \neq \hat{x} \Rightarrow D_M(x, y))|(\overline{e_1[\hat{x}]} + \overline{b_2[y]} + e_1[\hat{x}] \cdot b_2[y])|(e_1[\hat{x}])$. In other words, $b_2[y]$ is disabled for all y, because residuating the above expression with $b_2[y]$ yields 0. However, residuating the above expression with $e_1[\hat{x}]$ yields $(\forall x, y : x \neq \hat{x} \Rightarrow D_M(x, y))|\top|\top$. Thus $e_1[\hat{x}]$ can occur. Furthermore, anything allowed by $D_M(x, y)$ (except another occurrence of $b_1[\hat{x}]$ or $e_1[\hat{x}]$) can occur after $e_1[\hat{x}]$. ▮

We implicitly used the first order logic inference rule $(\forall z : D(z)) \equiv (\forall z : z \neq \hat{z} \Rightarrow D(z))|D(z ::= \hat{z})$, for any constant $\hat{z}$. By Observation 3, residuating with an event token $e[\hat{z}]$ returns the first conjunct unchanged, conjoined with the result of residuating $D(z ::= \hat{z})$ by $e[\hat{z}]$. In this manner, when parametrized events occur, dependencies can "grow" to accommodate the appropriate instances explicitly. These constrain some of the events. When the given instantiation of a dependency has been satisfied, that instantiation is no longer needed. If we assume that no event token is attempted after it or its complement has occurred, we can simplify our representation so that only the original dependency plus the currently live instantiations are explicitly stored. Thus, assuming that the tasks behave properly in assuring uniqueness of their events, in quiescense only the original dependency may be stored.

In this manner, we can handle parametrized dependencies quite naturally. There is no significant increase in complexity in going from dependencies in which all events share the variables to those in which different variables may occur in different events. This enables us to handle not only mutual exclusion as defined above, but also concurrency control requirements in general.

The alert reader would have noticed that Example 9 makes no assumptions about the conditions under which the two tasks attempt to enter or exit their critical sections. This turns out to be true in our approach in other cases as well. Importantly, the event IDs need *not* depend on the structure of the associated task, because our scheduler does not need to know the internal structure of a task agent. An agent may have arbitrary loops and branches and may exercise them in any order as required by the underlying task. Hence, we can handle arbitrary tasks correctly!

One might wonder about the value of parametrization to our formal theory. If we cared only about intra-workflow parametrization, we perhaps wouldn't need parameters explicitly, since they could be introduced extralogically, i.e., by modifying the way in which the theory is applied. However, when we care about inter-workflow parametrization, it is important to be able to handle parametrization from within the theory.

The unbound parameters in a dependency are treated as if universally quantified. This means that certain enforceable dependencies may become unenforceable when parametrized, e.g., when they require an infinitely many events to be triggered because of a single event occurrence. Determining the safe sublanguages is a problem we leave to future research.

6 Important Properties for Specification and Scheduling

We have presented a number of technical definitions and show how they meet several requirements in specifying and scheduling workflows. One might legitimately still wonder if some obvious alternatives would meet those requirements. In this section, we discuss the intuitive and technical rationale behind our syntax and semantic definitions. Some of our deepest concerns in settling upon a set of semantic definitions had to do with the subtle interrelationships between the operators $|$ and $\cdot$ on the one hand, and the constants $\top$ and 1 on the other. There is no separate constant 1 in the present proposal: it is in effect identified with $\top$. Normally, e.g., in [Pratt, 1990], $\top$ is defined as the unique maximal element of the given lattice and 1 is defined as the unit of the concatenation operator. That is, $\top$ would be roughly like in the above, whereas 1 would be associated with $\{\lambda\}$. We now explain why in our approach $1 \equiv \top$.

The relation $\subseteq$ on denotations indicates the strength of specifications. That is, $[\![E_1]\!] \subseteq [\![E_2]\!]$ means that E_1 is a stronger specification than E_2 and would be harder to meet than E_2. Indeed, this is the common intuition behind the notion of entailment in logic. Viewed this way, it is clear that we would like to validate the following claim: if the scheduler satisfies E followed by F, then it satisfies E and F.

Lemma 10 $[\![E \cdot F]\!] \subseteq [\![E]\!]$ and $[\![E \cdot F]\!] \subseteq [\![F]\!]$ ▮

Substituting 1 for F, where 1 is the unit of $\cdot$, this yields that, for all E, $[\![E \cdot 1]\!] \subseteq [\![1]\!]$. In other words, for all E, $[\![E]\!] \subseteq [\![1]\!]$. Thus, $[\![\top]\!] \subseteq [\![1]\!]$. Hence, $1 \equiv \top$.

We suspect the main reason why some approaches require $1 \not\equiv \top$ is to *avoid* the results that $(e + \overline{e}) \cdot f \equiv f$ and $f \equiv f \cdot (e + \overline{e})$. This would cause ordering information to be lost: f may be desired after e or $\overline{e}$, but not before them. But this result arises because those approaches require $e + \overline{e} \equiv \top$. But, in our approach, $e + \overline{e} \not\equiv \top$. Therefore, in our approach, setting $1 \equiv \top$ does not have the counterintuitive ramification that it might elsewhere.

Another important property is the following relationship between $\cdot$ and $|$. Lemma 11 says that you can always satisfy an interleaving specification by an arbitrary sequencing of the expressions involved.

Lemma 11 $[\![E|F]\!] \supseteq [\![E \cdot F + F \cdot E]\!]$ ∎

One might be tempted to *define* interleaving in terms of $\cdot$, by replacing the $\supseteq$ in lemma 11 by a $\triangleq$. But that would be problematic. Intuitively, we know that the interleaving of ab with cd is satisfied by $acbd$, which however is not captured by $abcd + cdab$. Further, the putative definition would not be associative. Moreover, since $|$ is used to conjoin specifications, it should be idempotent. Two copies of a dependency should reduce to just one copy. That is, $e|e$ should equal e, rather than ee. These properties of $|$ arise as trivial consequences of our definition.

By treating $\overline{e}$ as an event, we can record its occurrence before its task terminates. This is important for formally capturing the intuitions associated with eager scheduling. In our semantics $[\![e]\!] \cap [\![\overline{e}]\!] \neq [\![0]\!]$ and $[\![e]\!] \cup [\![\overline{e}]\!] \neq [\![\top]\!]$. This yields some technical advantages over other definitions [Pratt, 1990]. An alternative definition makes $[\![\overline{e}]\!]$ be the set complement of $[\![e]\!]$. As a result, $\lambda \in [\![\overline{e}]\!]$. This has the unfortunate consequence that $e \cdot \overline{e}$ semantically equals e (with or without admissibility). In fact, even under admissibility, $e \cdot \overline{e}$ remains satisfiable—this is clearly unacceptable. By contrast, our definition of complementation assigns only inadmissible traces to $e \cdot \overline{e}$. Thus, $e \cdot \overline{e} \approx_A 0$, for any admissible set A.

Another possible definition considers only maximal traces. In effect, this assigns a meaning to expressions only when all has been said and done. Consequently, it cannot express certain nuances that are essential for scheduling. For example, a problem arises if we attempt to give the semantics of certain kinds of expressions generated by nested $<$ dependencies of [Klein, 1991a], such as $(e_1 < e_2) < e_3$. Intuitively, $E < F$ means that if E and F both hold, then E precedes F. Apparently, the expression $(e_1 < e_2) < e_3$ should mean that if $e_1 < e_2$ holds, it comes to hold before e_3. But $e_1 < e_2$ holds if e_1 does *not* occur. However, if complements are defined in terms of maximal traces, it is difficult to decide whether e_1 did not occur before e_3 or did not occur after. Our algebra can naturally capture this meaning (or whatever else is desired) in a natural manner.

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. Some events are not typically thought of as having complements. For example, whereas *abort* and *commit* usually are complements of each other, *start* and *forget* usually have no complements at all. A superfluous formal complement causes no harm, because it is never instantiated. However, a complement is used for every event that is optional. Further, when the task agent has a multiway split (instead of two-way between *abort* and *commit*), 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.

7 Discussion

We developed a rigorous model-theoretic semantics for events and dependencies that satisfies both workflow intuitions and formal semantics criteria. This semantics provides a means to check the consistency and enforceability of dependencies. It also provides a simple way to generate eager schedules from declarative ("lazy") specifications, in a way that symbolically computes the preconditions and postconditions of performing an event. By using admissibility, we obtain a notion of residuation intimately related to the eager scheduling of events under dependencies. Our notion of residuation is specialized for use in scheduling, and yields stronger and more succinct answers for various scheduling decisions.

We have developed and demonstrated a working distributed prototype based on our theory [Singh *et al.*, 1994]. Our approach yields succinct representations for many interesting dependencies that arise in practice, e.g., compensation dependencies are of size 3 here, but of over size 40 in [Attie *et al.*, 1993]. The simplicity of our algebra facilitates specifications. Other approaches rely on fine syntactic variations, e.g., nested $<$ operators, which are confusing [Klein, 1991a] at best. Our approach involves no unintuitive semantic assumptions, but makes use of every available aspect of the problem to gain expressiveness and efficiency. The actual reasoning in our case is symbolic, i.e., using expressions that compactly represent branching histories or the corresponding sets of linear histories. In practice, we appear to obtain succinct representations—no worse than and often much better than previous approaches—but a detailed analysis of the complexity issues remains to be performed.

More general logic programming techniques for reasoning about integrity constraints and transactions are no doubt important, but the connection has not been explored yet [Bonner & Kifer, 1993; Lipeck, 1990]. It appears that we deal with lower-level scheduling issues, whereas the above approaches deal with application-level constraints. We speculate that they could supply the dependencies that are input to our approach. Our focus here was on identifying the core scheduling and semantic issues here, which will be relevant no matter how the final implementation is achieved.

References

[Agrawal *et al.*, 1993] Agrawal, Divyakant; Abbadi, Amr El; and Singh, Ambuj K.; 1993. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems* 18(3):460–486.

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

[Bernstein *et al.*, 1987] Bernstein, Philip A.; Hadzilacos, Vassos; and Goodman, Nathan; 1987. *Concurrency Control and Recovery in Database Systems.* Addison Wesley.

[Bonner & Kifer, 1993] Bonner, Anthony J. and Kifer, Michael; 1993. Database programming in transaction logic. In *Proceedings of the 4th International Workshop on Database Programming Languages (DBPL).*

[Breitbart *et al.*, 1993] Breitbart, Y.; Deacon, A.; Schek, H.-J.; Sheth, A.; and Weikum, G.; 1993. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *SIGMOD Record* 22(3).

[Chrysanthis & Ramamritham, 1992] Chrysanthis, Panos and Ramamritham, Krithi; 1992. ACTA: The SAGA continues. In *[Elmagarmid, 1992].* Chapter 10.

[Dayal *et al.*, 1993] Dayal, Umesh; Garcia-Molina, Hector; Hsu, Mei; Kao, Ben; and Shan, Ming-Chien; 1993. Third generation TP monitors: A database challenge. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Industrial track paper.

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

[Emerson, 1990] Emerson, E. A.; 1990. Temporal and modal logic. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B. North-Holland Publishing Company, Amsterdam, The Netherlands.

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

[Garcia-Molina *et al.*, 1990] Garcia-Molina, Hector; Gawlick, Dieter; Klein, Johannes; Kleissner, Karl; and Salem, Kenneth; 1990. Coordinating multi-transaction activities. Technical Report CS-TR-247-90, Princeton University Computer Science Department.

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

[Gray, 1981] Gray, J.N.; 1981. The transaction concept: Virtues and limitations. In *Proceedings of the 7th VLDB.*

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

[Hsu, 1993] Hsu, Meichun, editor. *Special Issue on Workflow and Extended Transaction Systems.* IEEE Data Engineering, 16(2). Contains 13 articles.

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

[Klein, 1991b] Klein, Johannes; 1991b. Coordinating reliable agents. Digital Equipment Corporation, Mountain View, CA. Submitted for publication.

[Lipeck, 1990] Lipeck, Udo W.; 1990. Transformation of dynamic integrity constraints into transaction specifications. *Theoretical Computer Science* 76:115–142.

[Nodine, 1993] Nodine, Marian H.; 1993. Supporting long-running tasks on an evolving multidatabase using interactions and events. In *Conference on Parallel and Distributed Information Systems.*

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

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

[Singh *et al.*, 1994] Singh, Munindar P.; Tomlinson, Christine; and Woelk, Darrell; 1994. Relaxed transaction processing. In *Proceedings of the ACM SIGMOD*. Research prototype demonstration description.

[Singh, 1996] Singh, Munindar P.; 1996. Synthesizing distributed constrained events from transactional workflow specifications. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*.

[Wächter & Reuter, 1992] Wächter, Helmut and Reuter, Andreas; 1992. The ConTract model. In *[Elmagarmid, 1992]*. Chapter 7.