Formal Semantics for Workflow Computations*

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

singh@ncsu.edu

Abstract

Workflows are composite multitask activities that accommodate the demands of heterogeneous environments by relaxing the semantic properties of traditional transactions. Workflows are increasingly important in modern computing. But to realize their full potential, they must be specified declaratively, reasoned about formally, and scheduled automatically. Although declarative approaches based on intertask dependencies exist, the semantic or model-theoretic aspects of dependencies have not been fully understood. Further, current approaches are restricted to loop-free tasks. We propose a rigorous formal semantics for workflow computations and dependencies. Our approach represents intertask dependencies as algebraic expressions. It uses efficient symbolic reasoning to capture scheduler transitions and changes of state. It includes an equational system that is guaranteed to yield the most general answers for scheduling, yet is sound and complete.

* A previous version of this paper appears in the Proceedings of the 5th International Workshop on Database Programming Languages (DBPL), held in Gubbio, Italy, in September 1995.
† Partially supported by the NCSU College of Engineering and by the National Science Foundation under grants IRI-8529179 and IRI-9624425.
Contents

1 Introduction 3

2 Overview of the Literature 7

3 Event Algebra
   3.1 Motivations and Key Assumptions 9
   3.2 Syntax and Semantics 10
   3.3 Residuation in Event Algebra 12
   3.4 Symbolic Calculation of Residuals 15

4 Correctness 18
   4.1 Admissibility 19
   4.2 Soundness 20
   4.3 Completeness 21

5 Arbitrary Tasks 21

6 Further Technical Properties 25

7 Conclusions 27

8 Acknowledgments 28

A Proofs of Important Results 31

©Munindar P. Singh 2 12 September 1996
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, which are computing environments consisting of a variety of databases and information systems. Such environments frequently involve fairly intricate semantic constraints among the related subactivities that take place on the different components. Although heterogeneity is ubiquitous in modern computing, much progress remains to be made in database tools for such environments. Database transactions have been intensively studied and robust implementations are available that provide an application programmer with considerable support in homogeneous or centralized environments. Unfortunately, no corresponding support is available in heterogeneous environments. The application programmer must procedurally encode all necessary semantic requirements.

Workflows are widely regarded as the appropriate concept for structuring complex activities, and workflow management systems would provide functions analogous to those provided by present-generation transaction monitors. However, workflows are inherently more complex than traditional transactions. One, transactions come prepackaged with a single notion of semantics that can be captured through a single scheduling mechanism that guarantees serializability and recoverability. This luxury is not available with workflows. For workflows to be an effective aid to the structuring of complex activities in heterogeneous environments, we must allow great flexibility in their specification and scheduling. For this reason, techniques for declaratively specifying and scheduling workflows are crucial to the workflow paradigm. For the same reason, the formal semantics of workflow computations on which the meanings of the specifications and the appropriate scheduling decisions must be based are also crucial. These are the topics of the present paper.

As the prevalence of heterogeneous environments is being appreciated, the importance of workflows is increasing. Although scores of “workflow” products exist, relatively few of these are integrated with databases. Even in these, little support is offered for the semantic or recoverability properties demanded by the serious deployment of workflows in heterogeneous information environments [Kamath & Ramamirtham, 1996]. Accordingly, much attention has recently been focused on workflows in the databases research community [Hsu, 1995; Hsu, 1993]. Workflows are listed as a major database challenge in [Dayal et al., 1993].

Our interest includes the class of workflows known as transactional workflows—workflows constituted from database transactions [Georgakopoulos et al., 1995]. Our theory and implementation apply to workflows whose constituent activities may or may not be database transactions. Indeed, we are able to accommodate arbitrary tasks, including those that may not terminate. However, transactional workflows are
a powerful means to structure complex activities in heterogeneous database environments. They are also a useful means to think about and study the transaction-like properties of complex activities that are often crucial to users and application programmers. For this reason, we draw our examples from transactional workflows, but emphasize the generality of our approach.

The traditional transaction model defines ACID transactions, which have the properties of atomicity, consistency, isolation, and durability. ACID transactions have proved 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 because of distribution and often impossible because of autonomous legacy applications. Local autonomy is often sacrosanct, because of technical and political reasons. Some components, e.g., legacy databases, are technically closed in that they have no visible precommit state. This precludes two-phase commit and other mutual commit protocols involving those components. Further, some components of heterogeneous systems are owned and managed by different organizations or divisions of an enterprise, which cannot or would grant control to another agency.

- Second, the semantic requirements in heterogeneous applications are often quite complex and need more sophisticated task structuring than the traditional model offers [Elmagarmid, 1992]. The field has only just begun to grapple with the subtleties of upcoming applications, e.g., in virtual enterprises [Goldschmidt et al., 1995].

For the above reasons, a number of extended transaction models have been proposed [Elmagarmid, 1992]. Typically, these generalize the ACID model in different ways for different intended applications. 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. If this paradigm is to succeed, at least two major issues must be addressed (which the above approaches do to varying degrees, as elaborated in section 2):

(a) how to express dependencies, and

(b) how to schedule events to satisfy them.

The present work is in this paradigm. We provide a formal language for specifying intertask dependencies and give a formal Tarskian semantics for it. 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

- encodes the knowledge of the scheduling system, and
- makes decisions on events through 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 provide the model-theoretic underpinnings of the specification and scheduling of workflows. 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. Some implementational aspects are reported in [Singh, 1996a]; 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 motivates and formalizes the appropriate notions of soundness and completeness of systems for reasoning about event schedules. It then proves the soundness and completeness of our equations. Section 5 extends our technical development to apply to arbitrary tasks. Section 6 motivates additional technical properties crucial for specification and scheduling and gives formal results showing how our approach satisfies them.
Figure 1: Example task agent handled by most approaches, including ours

Figure 2: Example task agent handled by our approach, but not by others
2 Overview of the Literature

A common theme in the literature is the notion of significant events [Chrysanthis & Ramamritham, 1992]. 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. Previous approaches handle only loop-free agents as in Figure 1, whereas our approach can handle arbitrary agents, as in Figure 2. The significant events label the legal transitions among these states. 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” $e \rightarrow e'$ 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 model-theoretic semantics. An important semantic issue from our standpoint is the distinction between event types and instances. ACTA’s 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]: \( (\text{read}_{t_i}[x] \rightarrow \text{write}_{t_j}[x]) \Rightarrow ((\text{Commit}_{t_j} \rightarrow \text{Commit}_{t_i}) \Rightarrow (\text{Commit}_{t_j} \rightarrow \text{read}_{t_i}[x])) \). This formula clearly uses \( \text{read}_{t_i}[x] \) to refer to two different event instances, one before \( \text{Commit}_{t_j} \), and the other after \( \text{Commit}_{t_j} \). Thus this formula can never be true in the cases where it applies—it can hold only vacuously.
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. Another temporal logic approach is that of Günthörr [1993]. Günthörr’s approach is based on linear temporal logic, and gives a formal semantics. His implementation too is centralized and his approach appears incomplete.

Action Logic Pratt’s approach is relevant as a general-purpose theory of events, although it was not designed for specifying and scheduling workflows [Pratt, 1990]. He proposes an algebra, and motivates a number of potentially useful inferences with which he constrains the possible models for the algebra. One class of models he considers are the regular languages, which correspond to linear histories of events. The branching (partially ordered) histories well-known from serializability theory [Bernstein et al., 1987] (also formalized in ACTA) can be expressed as sets of linear histories. We find this connection fruitful, although we are not interested in conflicts among operations, or in limiting ourselves to terminating computations.

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 denotation they give to a formula is not derived from the denotation 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 propose a set of equations and show how they can be used in performing the most general reasoning about how events may be scheduled. Much of our technical contribution is in resolving the tension among the various equations we propose so that they can be given a model-theory with respect to which they are sound.

3 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 why and where it is necessary. For ease of exposition, we defer parameterization of events to section 5.

3.1 Motivations and Key Assumptions

The property that most guided our approach was the wish to permit “lazy” specifications, yet schedule events eagerly. Specifications should be lazy 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.

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. Our approach captures the notion of change through 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.

### 3.2 Syntax and Semantics

$\mathcal{E}$, the language of event expressions has the following syntax. $\Sigma$ is the set of significant event symbols; $\Gamma$ is the alphabet; $\Xi$ 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** $\Xi \subseteq \mathcal{E}$

**Syntax 3** $0, \top \in \mathcal{E}$

**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 $E_1 \lor E_2 \in \mathcal{E}$

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

Intuitively, $e$ 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 $\lor$ means disjunction. The operator $\land$ means conjunction or interleaving. The operator $\cdot$ means sequencing. For notational simplicity, we assume that $\cdot$ has precedence over $\lor$ and $\land$, and $\land$ has precedence over $\lor$.

The semantics of $\mathcal{E}$ is given in terms of computations or *traces*. Each trace is a sequence of events. 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 $\langle e f \rangle$ means the trace in which event $e$ occurs followed by the event $f$. $\langle \rangle$ is the empty trace.

(Throughout, $\triangleq$ means *is defined as*.)

Let $U_\Gamma \triangleq \Gamma^* \cup \Gamma^\omega$ be our universe. This consists of all possible (finite and infinite) traces over $\Gamma$. For a trace, $\tau \in U_\Gamma$, and an expression $E \in \mathcal{E}$, $\tau \models E$ means that $\tau$ satisfies $E$. $[E]$ gives the *denotation* of an expression: $[E] \triangleq \{\tau : \tau \models E\}$.
Semantics 1 $[f] = \{\tau \in U_\Gamma : \tau \text{ mentions } f\}, \ f \in \Xi$

Semantics 2 $[0] = \emptyset$

Semantics 3 $[\top] = U_\Gamma$

Semantics 4 $[E_1 \cdot E_2] = \{v\tau \in U_\Gamma : v \in [E_1] \text{ and } \tau \in [E_2]\}$

Semantics 5 $[E_1 \lor E_2] = [E_1] \cup [E_2]$

Semantics 6 $[E_1 \land E_2] = [E_1] \cap [E_2]$

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 $E_2$. $E_1 \lor E_2$ denotes the union of the sets for $E_1$ and $E_2$. Lastly, $E_1 \land E_2$ 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 $\lor$, $\cdot$, and $\land$, and distributivity of $\cdot$ over $\lor$ and over $\land$.

Example 1 Let $\Gamma = \{e, \overline{e}, f, \overline{f}\}$ be the alphabet. Then the denotation of $e$,
$[e] = \{(e), (\overline{e}), (ef), (e\overline{f}), (e\overline{e}\overline{f}\overline{f}), \ldots\}$. The denotation of $e \cdot f$,
$[e \cdot f] = \{(e), (e\overline{e}), (ef), \overline{e} f, (f\overline{e}\overline{f}), \ldots\}$. Similarly, the denotation of $e \land f$,
$[e \land f] = \{(e), (\overline{e}), (\overline{ef}), (\overline{ef}\overline{e}), (\overline{e}\overline{f}\overline{e}), \ldots\}$. One can readily verify that
$[e \lor \overline{e}] \neq [\top]$ and $[e \land \overline{e}] \neq [0]$. ■

Definition 1 $E \equiv F$ iff $[E] = [F]$. This is an abbreviation, since $\equiv$ is not an operator in $\mathcal{E}$.

Observation 1 $\tau \in [E]$ iff $(\forall v, \nu : v \tau \nu \in U_\Gamma \Rightarrow v \tau \nu \in [E])$. ■

Observation 1 means that if a trace satisfies $E$, then all larger traces do so too. Conversely, if all traces that include $\tau$ satisfy $E$, then $\tau$ satisfies $E$ too (essentially by setting $v$ and $\nu$ to $\Lambda$).

Definition 2 $D$ is a sequence expression $\triangleq D = e_1 \cdot \ldots \cdot e_n$, where $n \geq 1$ and each $e_i \in \Gamma$. $\text{length}(D) \triangleq n$.

Observation 2 If $D$ is a sequence expression and $\tau \in [D]$ then
$(\forall i : 1 \leq i \leq \text{length}(D) \Rightarrow \tau \in [e_i])$. ■

Observation 3 If $D$ is a sequence expression and $\tau \in [D]$ then
$\text{length}(\tau) \geq \text{length}(D)$. ■

We treat $e$ and $\overline{e}$ symmetrically as events. We thus define a formal complement for each event, including those like $\text{start}$, whose non-occurrence constitutes their complement. This is crucial for eager scheduling—section 6 has more rationale. Traces that satisfy assumptions 1 and 2 of section 3.1 are termed legal. Our universe set (and Example 1) includes illegal traces, but these are eliminated from our formal model in section 4.1. Intuitively, the reader should assume legality everywhere.

© Munindar P. Singh 11 12 September 1996
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). For simplicity, assume that cancel always commits. Now the desired workflow may be specified as

- (D1) $s_{buy} \lor s_{book}$ (if buy starts, then book must also start);
- (D2) $c_{book} \lor c_{buy} \lor c_{book} \cdot c_{buy}$ (book commits before buy if both commit);
- (D3) $c_{buy} \lor c_{book}$ (buy commits only if book commits);
- (D4) $c_{book} \lor c_{buy} \lor c_{cancel}$ (compensate book by cancel—i.e., if book commits and buy aborts, then cancel must start); and
- (D5) $s_{cancel} \lor c_{buy} \land c_{book}$ (start cancel only if book commits and buy aborts).

The above workflow is satisfied by several legal traces, including $(\tau_1) s_{buy} s_{book} c_{book} c_{buy}$, $(\tau_2) s_{buy} s_{book} c_{book} c_{buy}$, $(\tau_3) s_{buy} s_{book} c_{book} c_{buy} s_{cancel} c_{cancel}$, $(\tau_4) s_{buy} s_{book} s_{cancel}$, and $(\tau_5) s_{book} s_{buy} c_{book} c_{buy}$.

3.3 Residuation in Event Algebra

Events are scheduled—by permitting or triggering—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. Initially, these are given by the stated dependencies. As events occur, the allowed traces get narrowed down.

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 $s_{buy}$. 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.

Intuitively, two questions must be answered for each event under consideration:

- can it happen now?
- what will remain to be done later?
The answers can be determined from the stated dependencies and the history of the system. One can examine the traces allowed by the original dependencies, select those compatible with the actual history, and infer how to proceed. Importantly, our approach achieves this effect symbolically, without examining the traces. Figures 3 and 4 show how the states and transitions of the scheduler may be captured symbolically. The state labels give the corresponding obligations, and the transition labels name the different events. Roughly, an event that would make the scheduler obliged to 0 cannot occur. These figures are based on dependencies D1 and D2 of Example 2, which were exercised in Example 3.

The transitions of Figures 3 and 4 can be captured through an algebraic operator called residuation. It is not added to \( \mathcal{E} \), 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 dependencies. The resultant dependencies implicitly contain the necessary history. This proves effective, because the representations typically are small and the processing is simple. We motivate our semantics for the \( / \) operator and then present an equational characterization of it.

The intuition embodied in Figures 3 and 4 is that to schedule a dependency \( D \), the scheduler first schedules an event \( e \) and then schedules the residual dependency \( D/e \). Our formal semantics reflects this intuition—to satisfy \( D \), the scheduler can allow any of the traces in \( \llbracket D \rrbracket \). Similarly, to schedule \( e \), the scheduler can use any of
the traces in \([\epsilon]\). And, to schedule \([D/\epsilon]\), the scheduler can use any of the traces in \([D/\epsilon]\). Thus the following must hold for correctness:

- \(\{vv : v \in [\epsilon] \text{ and } v \in [D/\epsilon] \} \cap U_{\Gamma} \subseteq [D]\)

Since we would like to allow all the traces that would satisfy the given dependency, we require that \([D/\epsilon]\) be the maximal set that satisfies the above requirement:

- \((\forall Z : (\{vv : v \in [\epsilon] \text{ and } v \in Z \} \cap U_{\Gamma} \subseteq [D]) \Rightarrow Z \subseteq [D/\epsilon])\)

Put another way, \([D/\epsilon]\) is the greatest solution to the inequation

\((\lambda Z : \{vv : v \in [\epsilon] \text{ and } v \in Z \} \cap U_{\Gamma} \subseteq [D])\)

Alternatively, we can put all traces that satisfy the above requirement into \([D/\epsilon]\), as below:

**Semantics 7** \(v \in [D/\epsilon]\) iff \((\forall v : v \in [\epsilon] \Rightarrow (vv \in U_{\Gamma} \Rightarrow vv \in [D]))\)

**Theorem 4** Semantics 7 gives the most general solution to the above inequation. ■

Theorem 4 states that given an occurrence of an event in a scheduler state corresponding to a set of traces, residuation yields the maximal set of traces which could correspond to the resulting state of the scheduler. Intuitively, since the traces corresponding to a state of the scheduler mean its current options, residuation maximizes the options.
3.4 Symbolic Calculation of Residuals

Semantics gives a powerful characterization of the evolution of the state of a scheduler, but offers no suggestions as how to determine the transitions. Fortunately, a set of equations exists using which the residual of any dependency can be computed.

In the following, we assume that the expressions are in conjunctive normal form (CNF). CNF for 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. A CNF representation of each expression is possible because of the distributivity of the operators , , and over each other.

**Observation 5** For any expression , there exists an expression , such that and is in CNF.

Because of the restriction to CNF, we can assume that in the equations below, is a sequence expression, and is a sequence expression or (the latter case allows us to treat a single atom as a sequence, using ). In Equation 4, and may be a sequence expression or or another disjunction (but not include a conjunction). gives the alphabet of an expression; Observation 6 states that events and their complements go together.

**Definition 3**

**Observation 6**

**Equation 1**

**Equation 2**

**Equation 3**

**Equation 4**

**Equation 5**

**Equation 6**

**Equation 7**

**Equation 8**

©Munindar P. Singh 1996
We use the operator \( \equiv \) in the equations to highlight that it might not be interpreted simply as \( \equiv \). Section 4.1 provides further explanation of this. The above equations are carefully designed to guide the reasoning of a scheduler when it is considering whether to allow an event \( e \). These equations take advantage of the assumptions of section 3.1. They have some important properties, including

- dependencies not mentioning an event have no direct effect on it
- the reasoning with respect to different dependencies can be performed modularly
- 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. We first consider two dependencies independently, and then the entire workflow.

Example 4 The reader can easily verify that the above equations characterize all the transitions given in Figures 3 and 4. For instance, looking at Figure 4, one can check that \((\overline{c_{book}} \lor c_{buy} \lor c_{book} \cdot c_{buy}) / c_{buy} \equiv c_{book}\). Thus if \(c_{buy}\) is allowed to happen in the initial state, only \(c_{book}\) may happen then. That is, \(c_{book}\) is prohibited. This makes sense because otherwise we would realize a trace in which \(c_{buy}\) and \(c_{book}\) both occur, and \(c_{book}\) precedes \(c_{book}\).

Example 5 Consider Example 2 again. The initial state of the scheduler is given by \(S_0 = D1 \land D2 \land D3 \land D4 \land D5\). The scheduler can allow \(buy\) to start first, because the resulting state \(S_1 = S_0 / s_{buy}\) is consistent. This simplifies to \(S_1 = s_{book} \land D2 \land D3 \land D4 \land D5\). The occurrence of \(c_{buy}\) next would leave the scheduler in state \(s_{book} \land \overline{c_{book}} \land c_{book} \land \overline{c_{cancel}}\), which is inconsistent because \(\overline{c_{book}} \land c_{book} = 0\) (legal trace cannot contain both). However, since \(S_2 = S_1 / s_{book} = T \land D2 \land D3 \land D4 \land D5\) is consistent, \(s_{book}\) can occur in \(S_1\). In \(S_2\), \(book\) can commit, resulting in the state \(S_3 = T \land (\overline{c_{buy}} \lor c_{buy}) \land T \land (c_{buy} \lor \overline{s_{cancel}}) \land (\overline{c_{buy}} \lor \overline{s_{cancel}})\). \(S_3\) allows either of \(c_{buy}\) 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 in the above equations 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 7**  $E/f \doteq E$, if $f \not\in \Gamma_E$

We now give some additional results. We define $\text{size}(E)$ to be the number of nodes in the parse tree of $E$. (Using abstract syntax, we consider different parse trees as different expressions.) The following results show that the equations yield simpler results than the input expression (Observations 8, 9, and 10), are deterministic (Observation 12), are convergent (Lemma 13), and produce CNF expressions, which can be input to other equations (Observation 14).

**Observation 8** If $D/e \doteq F$ is one of the above equations, then $\text{size}(D) \geq \text{size}(F)$

**Observation 9** If $D/e \doteq F$ is one of the above equations, then $\Gamma_D \supseteq \Gamma_F$

**Observation 10** If $D/e \doteq F$ is one of the above equations, then $\{e, \tau\} \not\subseteq \Gamma_D$

**Lemma 11** The number of operations required to compute $D/e$ is linear in $\text{size}(D)$.

**Observation 12** For any expression $D \in \mathcal{E}$ and any event $e \in \Gamma$, exactly one equation applies in computing $D/e$.

**Lemma 13** For any expression $D \in \mathcal{E}$ and any event $e \in \Gamma$, our equations yield exactly one expression belonging to $\mathcal{E}$.

**Observation 14** If $D/e \doteq F$ and $D$ is in CNF, then $F$ is in CNF.

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. By Observation 7, 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. Further details of a distributed scheduler based on this approach are reported in [Singh, 1996a].

©Munindar P. Singh 17 12 September 1996
4 Correctness

Correctness involves proving that the equations are sound and complete. Briefly, many of our equations turn out to be sound in the above model. However, some do not. Those equations rely on some of our other assumptions, which are not properly reflected in the above model. We motivate enhancements in order to establish that all equations are sound.

We now formalize the notions of soundness and completeness. Definition 4 formalizes a notion akin to entailment; definition 5 formalizes a notion akin to proof.

Definition 4 $D/e \models F$ iff $[D/e] = [F]$

Definition 5 $D/e \vdash F$ iff $D/e \models F_0 \models \ldots \models F_n = F$, using any of the above equations in each of the $\models$ steps.

The usual meaning of soundness is that whatever is provable is an entailment (roughly, everything that is proved is true) and the usual meaning of completeness is that all the entailments are provable (roughly, everything true can be proved). These are captured in our framework as definitions 6 and 7. Since soundness can be proved piecemeal for each equation, we define it in terms of individual equations in definition 8. Definition 8 interprets $\models$ as $\equiv$, i.e., equivalence or equality of denotations. Lemma 15 follows.

Definition 6 A system of equations is sound iff $D/e \vdash F$ implies that $D/e \models F$

Definition 7 A system of equations is complete iff $D/e \models F$ implies that $D/e \vdash F$

Definition 8 An equation $D/e \models F$ is sound iff $D/e \models F$ implies that $D/e \models F$

Lemma 15 Equations 1 through 6 are sound. ■

But what about Equations 7 and 8? The following lemma appears discouraging at first. However, it turns out that this can be corrected with the assumptions of section 3.1.

Lemma 16 Equations 7 and 8 are not sound. ■
4.1 Admissibility

Equations 7 and 8, along with Equation 6, are especially troublesome to prove sound in typical models. Individually, they can be satisfied in different models, but not together. The present model satisfies Equation 6, but if we had defined the universe set to include only the legal traces, then Equation 6 would not have been satisfied. 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.

Recall that assumptions 1 and 2 of section 3.1 define legal traces. Admissible traces are those that satisfy those assumptions as well as Assumption 3 below, which captures maximality.

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

Intuitively, admissible traces characterize the maximal (and legal) behavior of the environment, i.e., of the given collection of tasks. Let $A_\Theta$ be the set of all admissible traces on the alphabet $\Theta$, where $\epsilon \in \Theta$ iff $\tau \in \Theta$. As before, we identify $\overline{\tau}$ with $\epsilon$.

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 $\epsilon$. Suppose the scheduler is taking a decision on $\epsilon$. We know that $e'$ has not occurred yet, or it would have been residuated out already. Therefore, if we let $\epsilon$ happen now, then either we must (a) prevent $e'$, or (b) eventually let $e'$ happen followed by an instance of $\epsilon$ or $\tau$. Option (a) clearly violates $D$, since all traces in $\llbracket D \rrbracket$ must mention $e'$ (by Observation 2). Option (b) violates admissibility. Thus, assuming admissibility, we can prove Equation 7 the sound. 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 6 In general $\epsilon \neq 0$. However, after $\epsilon$ or $\overline{\tau}$ has occurred, another occurrence of $\epsilon$ is impossible. Hence, $\epsilon$ is effectively equivalent to 0.

Definition 9 $A$ is an admissible set iff $A = A_\Theta$ for some alphabet $\Theta$.

Initially the admissible set is the entire set of admissible traces for $\Gamma$. After event $\epsilon$ happens, the admissible set must be shrunk to exclude traces mentioning $\epsilon$ or $\overline{\tau}$.
because they cannot occur any more. Let $A$ be an admissible set before $e$ occurs. Then, after $e$, $A$ must be replaced by $A \uparrow e$, where $A \uparrow e$ yields the resulting set of admissible traces. The operator $\uparrow$ thus abstractly characterizes execution.

**Definition 10** $A \uparrow e \triangleq \{ \nu : \langle e \rangle \nu \in A \}$

**Observation 17** $A \uparrow e \cap \llbracket e \rrbracket = \emptyset$

**Observation 18** $A \uparrow e \cap \llbracket e \rrbracket = \emptyset$

We use admissible sets to define a coarser notion of equivalence ($\approx_A$) than equality of denotations ($\equiv$). Below, let $\Theta$ be a set of events, such that $e \in \Theta$ iff $\tau \in \Theta$.

**Definition 11** For an admissible set $A$, $E_1 \approx_A E_2 \triangleq A \cap [E_1] = A \cap [E_2]$.

**Example 7** Let $A = A_\emptyset$. Then, $e \lor \tau \approx_A \top$ ($A$ is maximal). Also, $e \land \tau \approx_A \bot$ ($A$ is consistent).

**Example 8** Let $A = A_\emptyset$, such that $e \not\in \Theta$. Then, $e \approx_A \tau$. Also, $e \approx_A \bot$.

**Lemma 19** 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!

However, the quotient construction would be valid only if we can 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. Theorem 20 states that if $E$ and $E'$ are adm-equivalent, then after event $f$ occurs, their respective residuals due to $e$ will also be adm-equivalent. In other words, the ongoing behavior of the scheduler cannot be affected by substituting $E'$ for $E$.

**Theorem 20** Let $E \approx_A E'$. Then, for all $f \in \Gamma$, $E/f \approx_{A^f} E'/f$.

### 4.2 Soundness

Theorem 20 is crucial in justifying the following formal notion of soundness, *adm-soundness*, which (in contrast to Definition 8) uses adm-equivalence instead of equality. This notion also accommodates the change of state implicit in event occurrence.

**Definition 12** $D/e \models F$ is adm-sound iff for all admissible sets $A$, $D/e \approx_{A[e]} F$
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 21** If $D/e \models F$ is sound, then $D/e \models F$ is adm-sound. ■

**Lemma 22** Equations 7 and 8 are adm-sound. ■

**Theorem 23** Equations 1 through 8 are adm-sound. ■

Theorem 23 thus establishes that all our equations are correct. It takes advantage of our intuitive assumptions of how the scheduler behaves and how events occur. Theorem 23 is important because it enables us to combine the benefits of most general solutions (Theorem 4) with the benefits of efficient computation (Lemma 11).

### 4.3 Completeness

After soundness, completeness is the natural question. Let $D$ be a dependency and $e$ be an event. We do not attempt to show that our equations will yield any expression that might characterize the denotation of $D/e$. By Lemma 13, our equations yield just one answer, whereas numerous expressions could have a denotation of $[D/e]$. Instead, we can show that our equations will indeed find an expression that is equivalent to the desired result with respect to the desired behavior of the scheduler.

**Definition 13** A set of equations is **adm-complete** iff $D/e \models F$ implies that $D/e \models F'$ and for all admissible sets $A$, $F \approx_{Ae} F'$

**Theorem 24** Equations 1 through 8 are adm-complete. ■

### 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, while event instances are not repeated, an event type may be instantiated multiple times. This assumption can be readily accommodated without any conceptual enhancement to the above development.

Each task is associated with a number of significant event types. Each event type is associated with at most one task—e.g., commit of *buy* is different from commit of *book*. Event symbols are interpreted as types; event instances are instantiated from types through parametrization. Some of the parameters in dependencies 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 instances
are unique. Typical parameters include transaction and task IDs, database keys, timestamps, and so on. We can uniquely identify each event instance by combining its task ID with the value of a monotonic counter that records the total number of significant event instances 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 define $E_{_P}$ as the language generated from Syntax rules [] as well as 8 and 9 given below.

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, $\Gamma$ includes all (ground) event literals and $\Xi$ includes all event atoms. The universe depends on $\Gamma$, and includes all traces formed from all possible event instances. Let $\delta(e)$ give the degree of $e$, i.e., the number of parameters needed to instantiate $e$.

**Syntax 7** $\Xi = \Gamma$

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

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

Semantics rules [] as well as 8 and 9 below.

Semantics 9 applies to an expression containing any variables—this is where 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** $[[f[p_1 \ldots p_m]] = \{ \tau \in \mathcal{U}_\Gamma : \tau$ 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 (a) events from the same task have the same variable parameters, and (b) all references to the same event type involve the same tuple of parameters. These assumptions are reasonable because our focus is on intertask dependencies. They enable us to interpret dependencies and schedule events properly.

We now consider two different ways of scheduling paramterized 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. The workflow is then scheduled as described in previous sections. We redo Example 2 below.

© Munindar P. Singh 22 12 September 1996
Example 9 Now we use \( t \) as the trip or reservation id to parametrize the workflow. The parameter \( t \) is bound when the \( \text{buy} \) 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

- \((D1')\) \( s_{\text{buy}}[t] \lor s_{\text{book}}[t] \);
- \((D2')\) \( c_{\text{book}}[t] \lor c_{\text{buy}}[t] \lor c_{\text{book}}[t] \cdot c_{\text{buy}}[t] \);
- \((D3')\) \( c_{\text{buy}}[t] \lor c_{\text{book}}[t] \);
- \((D4')\) \( c_{\text{book}}[t] \lor c_{\text{buy}}[t] \lor s_{\text{cancel}}[t] \); and
- \((D5')\) \( s_{\text{cancel}}[t] \lor c_{\text{buy}}[t] \land c_{\text{book}}[t] \).

Let \( t \) be bound by various natural numbers. The above workflow is satisfied by an infinite number of legal traces, including:

- \((\tau_6)\) \( s_{\text{buy}}[65]s_{\text{book}}[65]c_{\text{book}}[65]c_{\text{buy}}[65] \),
- \((\tau_7)\) \( s_{\text{buy}}[34]s_{\text{book}}[34]s_{\text{cancel}}[34] \), and
- \((\tau_8)\) \( s_{\text{buy}}[78]s_{\text{book}}[34]s_{\text{buy}}[34]c_{\text{book}}[34]s_{\text{book}}[78]c_{\text{book}}[78]c_{\text{buy}}[78]c_{\text{buy}}[34] \).

The traces \( \tau_6 \) and \( \tau_7 \) are as before but with explicit parameters. Trace \( \tau_8 \) 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 10 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 \( T_1 \).

\[
D_M(x, y) = (b_2[y] \cdot b_1[x] \lor e_1[x] \lor b_2[y] \lor e_1[x] \cdot b_2[y]) \land (b_1[x] \lor e_1[x])
\]

Intuitively, \( D_M(x, y) \) states that either \( T_2 \) enters its critical section before \( T_1 \) or

By Semantics 9, the above dependency is interpreted as \((\forall x, y : D_M(x, y))\). Suppose that \( b_1[x] \) for a specific and unique \( x \) occurs. This instantiates and residuates
the above expression to \((e_1[x] \lor b_2[y] \lor e_1[x] \cdot b_2[y]) \land (e_1[x])\). Thus the overall dependency becomes \((\forall x, y : x \neq x \Rightarrow D_M(x, y)) \land (e_1[x] \lor b_2[y] \lor e_1[x] \cdot b_2[y]) \land (e_1[x])\). 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 \(e_1[x]\) yields \((\forall x, y : x \neq x \Rightarrow D_M(x, y)) \land \top \land \top\). Thus \(e_1[x]\) can occur. Furthermore, anything allowed by \(D_M(x, y)\) (except another occurrence of \(b_1[x]\) or \(e_1[x]\)) can occur after \(e_1[x]\).

Suppose there are \(n\) tasks

\[
\text{We implicitly used the first order logic inference rule } (\forall z : D(z)) \equiv (\forall z : z \neq z \Rightarrow D(z)) \land D(z := z), \text{ for any constant } z. \text{ By Observation 7, residuating with an event instance } e[z] \text{ returns the first conjunct unchanged, conjoined with the result of residuating } D(z := z) \text{ by } e[z]. \text{ In this manner, when parametrized events occur, dependencies “grow” to accommodate the appropriate instances explicitly. When the given instantiation of a dependency has been satisfied, that instantiation is no longer needed. If we assume that no event instance 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 quiescence only the original dependency may be stored.}

The above reasoning can be formalized as follows. Here, we assume that \(\vec{v}\) is a tuple of variables that parametrize the occurrences of \(e\) in \(E\). We assumed above that this tuple is unique within each dependency. Similarly, \(\vec{c}\) is a tuple of constants with which the putative instance of \(e\) is instantiated.

**Equation 9** \(E(\vec{v})/e[\vec{c}] = E(\vec{v}) \land (E(\vec{v} := \vec{c})/e[\vec{c}])\)

**Lemma 25** Equation 9 is adm-sound.

Importantly, Example 10 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.

© Munindar P. Singh 24 12 September 1996
The unbound parameters in a dependency are treated as universally quantified. Thus certain enforceable dependencies may become unenforceable when parametrized, e.g., when they require 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 Further Technical Properties

Our formal definitions are carefully designed to specify and reason about workflows. We now give additional rationale for these definitions, and show why some obvious alternatives would not be appropriate.

There are subtle relationships between the operators \( \land \) and \( \cdot \), and the constants \( T \) and \( 1 \). Sometimes, e.g., [Pratt, 1990], \( T \) is the unique maximal element of the algebra and \( 1 \) is the unit of the concatenation operator. That is, \([T] = U\) (as here), whereas \([1] = \{\lambda\}\). However, we have no separate \( 1 \), effectively setting \( 1 \equiv T \).

We justify this as follows. Intuitively, \([E_1] \subseteq [E_2]\) means that \( E_1 \) is a stronger specification than \( E_2 \) and would be harder to meet (this is the common meaning of entailment). Consequently, any reasonable semantics should validate Lemma 26, which states that if the scheduler satisfies \( E \) followed by \( F \), then it satisfies \( E \) and \( F \). But, substituting \( 1 \) for \( F \) yields Observation 27.

**Lemma 26** \([E \cdot F] \subseteq [F] \).

**Observation 27** \( 1 \equiv T \).

We suspect that some approaches require \( 1 \neq T \) to avoid the results that \((e \lor \overline{e}) \cdot f \equiv f \) and \( f \equiv f \cdot (e \lor \overline{e}) \). 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 \( e \lor \overline{e} \equiv T \). Since in our approach, \( e \lor \overline{e} \neq T \), setting \( 1 \equiv T \) does not have the counterintuitive ramification that it might elsewhere.

Lemma 28 states that we can satisfy an interleaving specification by arbitrarily sequencing the expressions.

**Lemma 28** \([E \land F] \supseteq [E \cdot F \lor F \cdot E] \).

One might be tempted to define interleaving in terms of \( \cdot \), by replacing the \( \supseteq \) in Lemma 28 by a \( \triangle \). But that would be problematic. For example, we know that the interleaving of \( ab \) with \( cd \) is satisfied by \( acbd \), which however is not captured by \( abed \lor cdab \). Further, the putative definition would make interleaving non-associative and non-idempotent. Idempotence enables reducing two copies of a dependency to one copy. These properties arise as trivial consequences of our definition of \( \land \).
By treating $\tau$ as an event, we can record its occurrence before its task terminates. This enables formalizing eager scheduling. We have $[e] \cap [\tau] \neq [0]$ and $[e] \cup [\tau] \neq [T]$. An alternative definition makes $[\tau]$ be the set complement of $[e]$. As a result, $A \subseteq [\tau]$. This has the unfortunate consequence that $e \cdot \tau$ semantically equals $e$ (with or without admissibility). In fact, even under admissibility, $e \cdot \tau$ remains satisfiable. By contrast, our definition assigns only inadmissible traces to $e \cdot \tau$. Thus, $e \cdot \tau \approx_A 0$, for any admissible set $A$.

Another alternative considers only maximal traces. This, in effect, assigns a meaning to expressions only when all has been said and done, and cannot express certain nuances that are essential for scheduling. The denotation of an expression is the set of traces that satisfy it. In section 3.3, we appended the denotation of $e$ with that of $D/e$ in order to find a way to satisfy the $D$. If the denotation of $e$ consisted of maximal traces, this would not work so cleanly as above. 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 $e_3$. Our algebra can naturally capture this meaning in a natural manner.

We assume a formal complement for each significant event—both are atoms. Some events, e.g., start and forget, are not typically thought of as having complements. 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 applying the equations schematically. For example, if $e_1, e_2$, and $e_3$ define a three-way split, then we can reason that $(e_1 \cdot f)/e_3 \equiv 0$.

In addition to validating our equations, the admissibility construction enables certain simplifications. Lemma 29 works because the union and intersection of sets of maximal (and legal) traces are also sets of maximal (and legal) traces. Surprisingly, Lemma 30 holds because $\cdot$ inherently assumes nonmaximal traces. Thus the fact that its arguments were equal under maximality carries little significance.

**Lemma 29** $E \approx_A E'$ implies that each of the following holds:

(a) $E \land F \approx_A E' \land F$
(b) $F \land E \approx_A F \land E'$
(c) $E \lor F \approx_A E' \lor F$
(d) $F \lor E \approx_A F \lor E'$. ■

© Munindar P. Singh 26 12 September 1996
Lemma 30 $E \approx_A E'$ does not imply that either of the following holds:
(a) $E \cdot F \approx_A E' \cdot F$
(b) $F \cdot E \approx_A F \cdot E'$.

Therefore, we can use Lemma 29 to simplify expressions provided we avoid simplification in the context of a $\cdot$. Fortunately, since by Observation 14, our equations always yield CNF output from CNF input, we never have to worry about an expression in which the $\cdot$ operator is outside of a $\lor$ or $\land$. Thus, the simplifications prevented by Lemma 30 would never be needed anyway. So nothing is lost!

7 Conclusions

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 abstractly generates eager schedules from lazy specifications by symbolically computing the preconditions and postconditions of an event. By using admissibility, our definition of residuation is specialized for use in scheduling, and yields stronger and more succinct answers for various scheduling decisions.

We previously developed and demonstrated a working distributed prototype based on our theory [Singh et al., 1994]. The simplicity of our algebra facilitates the specification of workflows. 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.

We obtain succinct representations for many interesting dependencies that arise in practice—no worse and often much better than previous approaches. For example, compensation dependencies are given a representation of size 3 here, but of over size 40 in [Attie et al., 1993]. 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 [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.

Future work includes lifting the algebraic ideas and results to frameworks that
explicitly capture the structure of the computations and their user-defined semantics.

8 Acknowledgments

The early components of the reported research were performed in the Carnot Project at MCC in Austin, Texas under the directorship of Philip Cannata. I benefited from the design and implementation of a previous approach described in [Attie et al., 1993]. I also participated in several discussions with Paul Attie, Greg Meredith, and Chris Tomlinson. I alone am responsible for the contents, however.

References


A Proofs of Important Results

Proof of Lemma 13.

Statement. For any expression $D \in \mathcal{E}$ and any event $e \in \Gamma$, our equations yield exactly one expression belonging to $\mathcal{E}$.

Consider any equation that applies on $D$. If this is equation 4 or equation 3, it results in recursive calls on $/$ but on expressions whose size is strictly smaller than $D$. If it is any other equation, it produces an answer without making any recursive calls. Hence, the equations terminate. Since, by Observation 12, exactly one equation applies at each stage, the final answer is unique. Thus our equations are convergent.

Proof of Lemma 15.

Statement. Equations 1 through 6 are sound.

Equations 1 and 2 follow trivially from Semantics 7. Consider Equation 3. $\nu \in \llbracket (E_1 \land E_2) / e \rrbracket$ iff $(\forall v : v \in [e] \Rightarrow (v \cdot \Gamma \Rightarrow v \cdot \nu \in [E_1 \land E_2] \rrbracket)$. This holds iff $(\forall v : v \in [e] \Rightarrow (v \cdot \Gamma \Rightarrow v \cdot \nu \in [E_1] \cap [E_2] \rrbracket)$, which is equivalent to $(\forall v : v \in [e] \Rightarrow (v \cdot \Gamma \Rightarrow v \cdot \nu \in [E_1] \rrbracket) \land (\forall v : v \in [e] \Rightarrow (v \cdot \Gamma \Rightarrow v \cdot \nu \in [E_2] \rrbracket)$. But this is equivalent to $\nu \in [E_1 / e] \cap [E_2 / e]$.

Equation 4 is the hardest of these equations. We can show that $\llbracket (E_1 \lor E_2) / e \rrbracket \supseteq [E_1 / e] \cup [E_2 / e]$. For the other direction, we use the fact that the expressions have been cast into CNF. Consequently, $E_1$ and $E_2$ can each either be 0, $\top$, a sequence expression, or a disjunction. If either $E_1$ or $E_2$ is 0 or $\top$, then Equation 4 is trivially satisfied. Let $\nu \in \llbracket (E_1 \lor E_2) / e \rrbracket$. Then, since $\langle e \rangle \in [e]$, the trace $\langle e \rangle \cdot \nu \in [E_1 \lor E_2]$. Assume, without loss of generality, that $\langle e \rangle \cdot \nu \in [E_1]$.

1. Let $E_1$ be a sequence expression. There are two cases.

   (a) $E_1 = e \cdot D$. Now, $\langle e \rangle \cdot \nu \in [e \cdot D]$ implies that (using Observation 1) $\nu \in [D]$. Then, by Semantics 4, $(\forall v : v \in [e] : v \cdot \nu \in [e \cdot D])$. Hence, $\nu \in [E_1 / e]$.

   (b) $E_1 = f \cdot D$ and $e \neq f$. By Observation 2, $\langle e \rangle \cdot \nu \in [f \cdot D]$ implies that $\nu \in [f \cdot D]$. By Observation 1, $(\forall v : v \in [e] : v \cdot \nu \in [f \cdot D])$. Hence, $\nu \in [E_1 / e]$.

2. Let $E_1$ be a disjunction. This is the only recursion possible in the structure of $E_1$ (since it is in CNF). The proof follows by structural induction in a straightforward manner.

   Consider Equation 5. Let $\nu \in [E]$. Then, by Semantics 4, $(\forall v : v \in [e] \Rightarrow v \cdot \nu \in [e \cdot E])$. Therefore, $\nu \in \llbracket (e \cdot E) / e \rrbracket$. Hence, $\llbracket E \rrbracket \subseteq \llbracket (e \cdot E) / e \rrbracket$. Conversely,
let \( \nu \in \llbracket (e \cdot E)/e \rrbracket \). Then \( \langle e \rangle \nu \in \llbracket e \cdot E \rrbracket \). By Observation 3, any trace satisfying \( e \) must be at least of length 1. Therefore, by Semantics 4, there must a suffix \( \tau \) of \( \nu \) that satisfies \( E \). Thus, by Observation 1, \( \nu \in \llbracket E \rrbracket \). Hence, \( \llbracket (e \cdot E)/e \rrbracket \subseteq \llbracket E \rrbracket \). Thus, \( \llbracket (e \cdot E)/e \rrbracket = \llbracket E \rrbracket \).

Lastly, consider Equation 6. Let \( \nu \in \llbracket D/e \rrbracket \). Then \( \langle e \rangle \nu \in \llbracket D \rrbracket \). Since \( e, \overline{e} \not\in \Gamma_D \), \( \nu \in \llbracket D \rrbracket \). Thus, \( \llbracket D/e \rrbracket \subseteq \llbracket D \rrbracket \). Let \( \nu \in \llbracket D \rrbracket \). Then, by Observation 1, \( (\forall v : v \in \llbracket e \rrbracket \Rightarrow v \nu \in \llbracket D \rrbracket ) \). Thus, \( \nu \in \llbracket D/e \rrbracket \) or \( \llbracket D/e \rrbracket \supseteq \llbracket D \rrbracket \). Hence, \( \llbracket D/e \rrbracket = \llbracket D \rrbracket \), as desired.

This completes the proof.

**Proof of Lemma 16.**

**Statement.** Equations 7 and 8 are not sound.

The proof is by simple counterexamples. For Equation 7, let \( e' = f \) and \( E = e \). We can verify that \( \langle f \rangle \in \llbracket (f \cdot e)/e \rrbracket \). Thus \( \llbracket (f \cdot e)/e \rrbracket \neq \emptyset \). Similarly, for equation 8, let \( E = f \). We can verify that \( \langle \overline{e} \rangle \in \llbracket (\overline{e} \cdot f)/e \rrbracket \). Thus \( \llbracket (\overline{e} \cdot f)/e \rrbracket \neq \emptyset \).

**Proof of Theorem 20.**

**Statement.** Let \( E \approx_A E' \). Then, for all \( f \in \Gamma \), \( \llbracket f \rrbracket \approx_{A_{\llbracket f \rrbracket}} \llbracket E'/f \rrbracket \).

Let \( \nu \in (A \uparrow f \cap \llbracket E/f \rrbracket) \). Then, \( \langle f \rangle \nu \in A \). Also, \( (\forall v \in \llbracket f \rrbracket : v \nu \in \llbracket E \rrbracket ) \). Therefore, since \( \langle f \rangle \in \llbracket f \rrbracket \), we have that \( \langle f \rangle \nu \in \llbracket E \rrbracket \). Thus \( \langle f \rangle \nu \in (A \cap \llbracket E \rrbracket ) \). Since \( E \approx_A E' \), \( \langle f \rangle \nu \in (A \cap \llbracket E' \rrbracket ) \). By Observation 1, \( \langle f \rangle \nu \in \llbracket E' \rrbracket \) implies that \( (\forall v \in \llbracket f \rrbracket : v \nu \in \llbracket E' \rrbracket ) \). Thus \( \nu \in \llbracket E'/f \rrbracket \). Since \( \nu \in A \uparrow f \), we obtain \( \nu \in (A \uparrow f \cap \llbracket E'/f \rrbracket ) \). Consequently, we have established that \( (A \uparrow f \cap \llbracket E/f \rrbracket ) \subseteq (A \uparrow f \cap \llbracket E'/f \rrbracket ) \). By symmetry, \( (A \uparrow f \cap \llbracket E'/f \rrbracket ) \subseteq (A \uparrow f \cap \llbracket E/f \rrbracket ) \). Thus, \( (A \uparrow f \cap \llbracket E/f \rrbracket ) = (A \uparrow f \cap \llbracket E'/f \rrbracket ) \). Or, \( \llbracket f \rrbracket \approx_{A_{\llbracket f \rrbracket}} \llbracket E'/f \rrbracket \).

**Proof of Lemma 22.**

**Statement.** Equations 7 and 8 are adm-sound.

Consider Equation 7. By Semantics 7, \( \nu \in \llbracket (e' \cdot E)/e \rrbracket \) iff \( (\forall v : v \in \llbracket e \rrbracket \Rightarrow v \nu \in \llbracket e' \cdot E \rrbracket ) \). Since \( \langle e \rangle \in \llbracket e \rrbracket \), this implies that \( \langle e \rangle \nu \in \llbracket e' \cdot E \rrbracket \). By Observation 3, any trace that satisfies \( e' \) must be at least of length 1. Thus, by Semantics 4, a suffix \( \tau \) of \( \nu \) exists such that \( \tau \in \llbracket E \rrbracket \). By Observation 1, \( \nu \in \llbracket E \rrbracket \). Since we normalize expressions to CNF, \( E \) is a sequence expression. It is given that \( e \in \Gamma_E \). Therefore, \( \nu \in \llbracket e \rrbracket \). Thus by Observation 17, \( \nu \not\in A \uparrow e \). Consequently, \( A \uparrow e \cap \llbracket (e' \cdot E)/e \rrbracket = \emptyset \), which equals \( A \uparrow e \cap \llbracket \emptyset \rrbracket \). Hence, Equation 7 is adm-sound.

A similar proof can be constructed for Equation 8.
Proof of Theorem 24.

Statement. Equations 1 through 8 are adm-complete.

By Lemma 13, $D/\epsilon$ always evaluates to a unique expression. Let this be $F'$.
Thus, $D/\epsilon \vdash F'$ always holds for some $F'$. Let $D/\epsilon \models F$. By Theorem 23, $F' \approx_{A_\theta} F$. Hence, we have the result.

Proof of Lemma 25.

Statement. Equation 9 is adm-sound.

Semantics 9 states that the denotation of an expression with variables is the intersection of the denotations of the expression obtained through its various possible instantiations. By Observation 7, all instantiations except $\bar{c}$ are independent of $\epsilon[\bar{c}]$. By admissibility, $\epsilon[\bar{c}]$ or $\bar{c}[\bar{c}]$ cannot occur again. Hence the result.

Proof of Lemma 30.

Statement. $E \approx_A E'$ does not imply that either of the following holds:

(a) $E \cdot F \approx_A E' \cdot F$
(b) $F \cdot E \approx_A F \cdot E'$

Let $\Gamma = \{\epsilon, \overline{\tau}, f, \overline{f}\}$ and $\Theta = \Gamma$. Let $E = \epsilon \lor \overline{\tau}$, $E' = \top$, and $F = f$. Then, $E \approx_A E'$, for all admissible $A$. However, $E \cdot F \not\approx_{A_0} E' \cdot F$. And similarly for the opposite order of arguments.