Formal Aspects of Workflow Management
Part 1: Semantics*†

Munindar P. Singh‡
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
singh@ncsu.edu

June 18, 1997

Abstract

Workflows are composite activities that achieve interoperation of a variety of system and human tasks. Workflows must satisfy subtle domain-specific integrity and organizational requirements. Consequently, flexibility in execution is crucial. A promising means to achieve flexibility is through declarative specifications (Part 1) with automatic distributed scheduling techniques (Part 2).

Intertask dependencies are constraints among the tasks that constitute a workflow. We propose a rigorous formal semantics for workflow computations and dependencies. Importantly, our approach uses symbolic reasoning to capture scheduler transitions. It includes an equational system that is guaranteed to yield the most general answers for scheduling, yet is sound and complete.

workflows, temporal logic, formal semantics.

1 Introduction

Workflows are composite activities that typically involve a variety of computational and human tasks, and span multiple systems. Workflows arise naturally in heterogeneous environments, which are computing environments consisting of a variety of databases and

*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.

†The early components of the reported research were performed in the Carnot Project at MCC in Austin, Texas. I participated in several discussions with Paul Attie, Greg Meredith, and Chris Tomlinson. Many significant revisions and enhancements were made at NCSU. I am thankful to Krithi Ramamritham and Chubin Lin for helpful comments. I alone am responsible for the contents, however.

‡Partially supported by the NCSU College of Engineering, by the National Science Foundation under grants IRI-9529179 and IRI-9624425, and by IBM.

information systems. Such environments frequently involve fairly intricate integrity and organizational constraints among different sites.

For traditional (homogeneous or centralized) environments, database transactions provide effective and robust support for applications. Unfortunately, corresponding support is not available in heterogeneous environments. 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. Accordingly, increasing attention has focused on workflows [8, 20].

Although several workflow products exist, relatively few of these are integrated with databases. Even the best of those integrated with databases often have centralized implementations, and offer little support for semantic or recoverability properties. For example, InConcert lists distribution as a future challenge [31]. ActionWorkflow is also centralized, and more geared toward computer-supported collaborative work [37]. Flowmark offers strong support for business processes, but has a centralized implementation [26]. However, associated with these products are useful techniques for process modeling and capture, e.g., ActionWorkflow's "language/action" model of human interaction, and Flowmark's activity-network model of processes.

Important open issues for products and research include (a) semantic or recoverability properties demanded by serious applications; (b) distributed implementations; (c) ease of specifying and modifying workflows; and (d) formal models of workflow computations to give a solid foundation for the reasoning necessary to design, analyze, and reliably enact workflows [22].

Technical Motivation

Workflows incorporate both integrity and organizational requirements [5, 29]. The former include abstractions from extended transaction models that relate to correctness, concurrency control, failure handling, and recovery. The latter include abstractions from business modeling that relate to organizational structure, roles of participants, and coordination among them and the information system. Both aspects would be facilitated by an infrastructure that includes a generic and rigorous approach to workflow specification and scheduling.

The traditional transaction model defines ACID transactions, which have the properties of *a*tomicity, *c*onsistency, *i*solation, and *d*urability. ACID transactions have proved remarkably effective in a number of data processing applications [17]. Unfortunately, they are not well-suited to heterogeneous systems. First, atomic commit protocols are inefficient because of distribution and often impossible because of autonomous components. Second, the semantic requirements in heterogeneous applications are often complex and need more sophisticated task structuring than ACID transactions [10].

Extended transaction models (ETMs) [10] generalize the ACID model in different ways. Each ETM requires customized scheduling. This motives generic approaches that provide a small number of primitives to specify ETMs, e.g., [16, 6, 2]. These approaches provide declarative *intertask dependencies* to specify workflows, which may correspond to different ETMs. This paradigm presupposes that two major issues be addressed:

(a) how to express dependencies, and

(b) how to schedule events to satisfy dependencies.

Contributions

We provide a formal language for specifying intertask dependencies and give it a formal *model-theoretic semantics* ([13, pp. 79–94] is a good introduction to model-theoretic semantics—this paper is self-contained, however). Our approach addresses both of the above issues—expressing dependencies and scheduling events. It has two strengths. 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 distinguishes between event types and instances. Second, our approach has key features crucial to scheduling: it encodes the knowledge of the scheduling system, and provides an operator for the most general decisions on events, and computes it through symbolic reasoning. We provide rigorous definitions of workflow computations, and show how those definitions may be used to specify workflows and to formally reason about their properties. Details of scheduling are reported in [32].

Our approach applies to workflows whose constituent activities may or may not be database transactions. However, we draw our examples from *transactional workflows*, which include database transactions as component activities [15].

Organization

Section 2 describes our system model. Section 3 presents our event algebra for representing and reasoning about dependencies. It 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 establishes the soundness and completeness of our equations. Section 5 extends our technical development to apply to arbitrary tasks. Section 6 shows how we satisfy technical properties crucial for specification and scheduling. Section 7 reviews the related research. Appendix A gives the proofs. Appendix B shows a translation from information control nets to our notation.

2 Execution Model

A workflow in execution involves the coordinated execution of different tasks. Following [6], we use *significant* events to model the tasks. Typically, the events correspond to transaction manager or operating system primitives, such as *begin*, *commit*, *abort*, *spawn*, and *fail*. The events specify the visible operations of a task. Intertask dependencies, which specify workflows, are constraints across the significant events of different tasks.

The tasks are interfaced to the scheduler through proxy *agents*, which hide the complexity of the tasks and show only transitions corresponding to significant events. Figure 1 shows a task agent suitable for a transaction that has an explicit prepared state. Figure 2 shows a task agent suitable for a computation that loops over and retries some subcomputation. Our approach is not limited to the above agents. The agent informs the system of uncontrollable

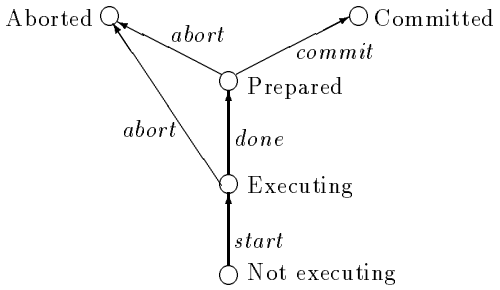


Figure 1: Task agent for a typical transaction [11, p. 535]

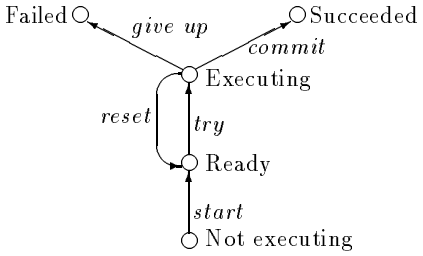


Figure 2: Task agent for a looping computation

events like *abort* and requests permission for controllable ones like *commit*. When triggered by the system, it causes appropriate events like *start* in the task.

Consider the Workflow Management Coalition (WfMC) reference model [36]. Our system can be viewed as providing a "workflow enactment service." Our algebraic language provides a rudimentary means to formally specify processes into which high-level languages may be translated. Based on the specifications, our run-time environment deals with "workflow client applications" that can attempt events in the enactment service. The run-time environment can also trigger "invoked applications," which can be arbitrary programs. We emphasize that, whereas our system has been prototyped, it is by no means an implementation of the WfMC model.

3 Event Algebra

Because events describe the operations of tasks, and workflows coordinate tasks by constraining their significant events, to represent workflows, we represent events and dependencies on them. Our formal language is based on an algebra of event types, which is related to Pratt's action logic [28]. Our key motivation is that specifications should describe entire computations, without regard to the details scheduling. But 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.

However, we need special equations to obtain the necessary independence and modularity properties, and closed-form answers for symbolic reasoning. We discovered that our equations were not sound in 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 both notions through a quotient construction, which identifies expressions that are equivalent with respect to the desired behavior of the scheduler.

3.1 Syntax and Semantics

$\mathcal{E}$, the language of event expressions has the following syntax. $\Sigma \neq \emptyset$ is the set of significant event symbols; , $= \{e, \overline{e} : e \in \Sigma\}$ is the *alphabet*. A *dependency* or an expression is a member of $\mathcal{E}$. A *workflow* is a set of dependencies.

Syntax 1 , $\subseteq \mathcal{E}$

Syntax 2 $0, \top \in \mathcal{E}$

Syntax 3 $E_1, E_2 \in \mathcal{E}$ implies $E_1 \vee E_2$, $E_1 \wedge E_2$, $E_1 \cdot E_2 \in \mathcal{E}$

Intuitively, e means that event e occurs somewhere. The constant 0 refers to a specification that is always false; $\top$ refers to one that is always true. The operator $\vee$ means disjunction. The operator $\wedge$ means conjunction or interleaving. The operator $\cdot$ (dot) means sequencing. To reduce the number of parentheses, we assume that $\cdot$ has precedence over $\wedge$, and $\wedge$ has precedence over $\vee$.

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 (we could underline the latter, but that would add more clutter than clarity!). 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 $\mathbf{U}_{\mathcal{E}} \triangleq , {}^{*} \cup , {}^{\omega}$ be our universe. This consists of all possible (finite and infinite) traces over , . For a trace, $\tau \in \mathbf{U}_{\mathcal{E}}$, 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 $[\![f]\!] = \{\tau \in \mathbf{U}_{\mathcal{E}} : \tau \text{ mentions } f\}$, $f \in$,

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

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

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

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

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

The syntax and semantics of $\mathcal{E}$ is carefully designed to mirror propositional logic. Atoms e and $\overline{e}$ are literals, 0 is false, $\top$ is true, $\vee$ is or, $\wedge$ is and. The only differences with propositional logic are (a) an atom e denotes the set of traces in which event e occurs, and (b) $\cdot$ is an additional operator denoting sequencing of its arguments. Technically, $E_1 \cdot E_2$ corresponds to memberwise concatenation of the denotation of E_1 with that of E_2; $E_1 \vee E_2$ corresponds to their union; and $E_1 \wedge E_2$ to their intersection. This semantics validates useful properties, e.g., associativity of $\vee$, $\cdot$, and $\wedge$, and distributivity of $\cdot$ over $\vee$ and $\wedge$.

Example 1 Let , $= \{e, \overline{e}, f, \overline{f}\}$ be the alphabet. Then the denotation of e, $[\![e]\!] = \{\langle e\rangle, \langle e\overline{e}\rangle, \langle ef\rangle, \langle fe\rangle, \langle e\overline{f}\rangle, \langle ee\overline{e}f\overline{f}\rangle, \ldots\}$. The denotation of $e \cdot f$, $[\![e \cdot f]\!] = \{\langle ef\rangle, \langle e\overline{e}f\rangle, \langle eff\overline{e}\rangle, \langle fee\overline{e}f\overline{f}\rangle, \ldots\}$. One can verify that $[\![e \vee \overline{e}]\!] \neq [\![\top]\!]$ and $[\![e \wedge \overline{e}]\!] \neq [\![0]\!]$. ∎

Definition 1 $E \equiv F$ iff $[\![E]\!] = [\![F]\!]$. This is an abbreviation ($\equiv$ is not in $\mathcal{E}$).

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

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 constitutes their complement. This is crucial for eager scheduling—section 6 gives additional rationale. Traces that satisfy assumptions 1 and 2 are termed *legal*:

Assumption 1 Event instances and their complements are mutually exclusive.

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

Our universe set (and Example 1) includes illegal traces, but these are eliminated from our formal model in section 4.1. The reader should assume legality everywhere.

As running examples, we use two dependencies due to [23], and related to the primitives in [2, 7, 19]. $e < f$ means that if both events e and f happen, then e precedes f; $e \rightarrow f$ means that if e occurs then f also occurs (before or after e). Again, these behave as propositional logic with a temporal flavor:

Example 2 Let $D_< \triangleq \overline{e} \vee \overline{f} \vee e \cdot f$. Let $\tau \in \mathbf{U}_\mathcal{E}$ satisfy $D_<$. If τ satisfies both e and f, then e and f occur on τ. Thus, neither $\overline{e}$ nor $\overline{f}$ can occur on τ. Hence, τ must satisfy $e \cdot f$, which requires that an initial part of τ satisfy e and the remainder satisfy f. That is, e must precede f on τ. ∎

Example 3 Let $D_\rightarrow \triangleq \overline{e} \vee f$. Let $\tau \in \mathbf{U}_\mathcal{E}$ satisfy $D_\rightarrow$. If τ satisfies e, then e occurs on τ. Thus, $\overline{e}$ cannot occur on τ. Hence, f must occur somewhere on τ. ∎

Example 4 Consider a workflow which attempts to *buy* an airline ticket and *book* a hotel for a traveler, such that the ticket is bought if and only if the hotel is booked. Mutual commit protocols cannot be executed, since the airline and hotel are different enterprises and their databases may not have a visible precommit state.

Assume that (a) the booking can be canceled: thus *cancel* compensates for *book*, and (b) the ticket is nonrefundable: *buy* cannot be compensated. Assume all subtasks have at least

start, *commit*, and *abort* events, as in Figure 1. For simplicity, assume that *book* and *cancel* always commit. This may be specified as follows. (D_1) $\overline{s_{buy}} \vee s_{book}$ (initiate *book* upon starting *buy*); (D_2) $\overline{c_{buy}} \vee c_{book} \cdot c_{buy}$ (if *buy* commits, it commits after *book*—this is reasonable since *buy* cannot be compensated and commitment of *buy* effectively commits the entire workflow); (D_3) $\overline{c_{book}} \vee c_{buy} \vee s_{cancel}$ (compensate *book* by *cancel*); and (D_4) $\overline{s_{cancel}} \vee c_{book} \wedge \overline{c_{buy}}$ (start *cancel* only if necessary, i.e., if *book* aborted or *buy* committed).

Note that D_2 explicitly orders c_{book} before c_{buy}, but D_1, D_3, and D_4 do not order any events. However, to be triggered, the events should have the attribute *triggerable*—introduced in [32]. The scheduler causes the events to occur when necessary, and may order them before or after other events as it sees fit. ∎

We now consider Leymann's *spheres of joint compensation (SOJCs)*, and show how to represent them [25]. An SOJC is a set of activities that either all commit or all are compensated. The compensation happens in the reverse order of the order in which the activities committed. We consider only the scheduling aspects of SOJCs.

Example 5 Let $\{T_1, \ldots, T_n\}$ along with their compensating activities $\{R_1, \ldots, R_n\}$ form an SOJC. Let each T_i and R_i be a transaction as in Figure 1. This SOJC can be captured by the following dependencies.

- (J_1) $\overline{s_{T_i}} \vee s_{T_j}$ (if one activity (T_i) starts, all the others must also start);
- (J_2) $c_{T_i} \vee \overline{c_{T_j}} \vee s_{R_j}$ (if T_i aborts, then each activity T_j that committed must be compensated by R_j);
- (J_3) $\overline{s_{R_l}} \vee \overline{s_{R_m}} \vee c_{T_l} \cdot c_{T_m} \vee c_{R_l} \cdot c_{R_m}$ (compensations commit in the reverse order of the forward activities);
- (J_3') $\overline{s_{R_l}} \vee \overline{s_{R_m}} \vee c_{T_l} \cdot c_{T_m} \vee s_{R_l} \cdot s_{R_m}$ (compensations start in the reverse order of the commitment of the forward activities); and
- (J_4) $c_{T_i} \vee c_{T_j} \cdot \overline{c_{T_i}} \vee \overline{c_{T_j}}$ (no activity commits after one of them aborts).

J_1 may be eliminated if the activities will be started separately. J_2 and J_3 represent the core of the SOJC. J_3 may be replaced by J_3', which states says that the compensates start serially. J_4 says that when an activity aborts, the others are aborted right away—this avoids redundantly committing and then compensating them. ∎

An advantage of a formal notation is that it forces one to decide on the desired meaning, i.e., a specific subset of the dependencies in Example 5.

3.2 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 6 Consider Example 4. Suppose *buy* starts first. Then D_1 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*, D_2 would be violated, because it states that *buy* can commit only after *book* has committed. Suppose *book* starts, thereby satisfying the remaining obligation from D_1. After *book* starts, D_2 would still prevent *buy* from committing. Eventually, if *book* commits, *buy* can commit thus completing the workflow (because of D_2 and D_4), or *buy* can abort thus causing *cancel* to be started (because of D_3). ∎

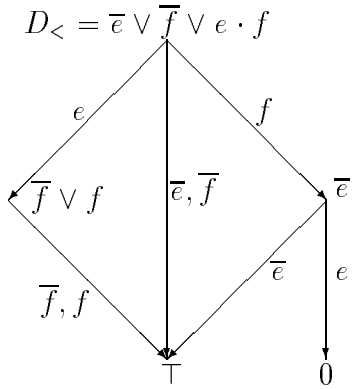


Figure 3: Scheduler states and transitions for $D_<$

Intuitively, two questions must be answered for each event under consideration: (a) can it happen now? and (b) 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.

Figure 3 shows 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.

Example 7 (Figure 3) If $\overline{e}$ or $\overline{f}$ happens, then $D_<$ is necessarily satisfied. If e happens, then either f or $\overline{f}$ can happen later. But if f happens, then only $\overline{e}$ must happen afterwards (e cannot be permitted any more, since that would mean f precedes e). ∎

The transitions of Figure 3 can be captured through an algebraic operator called *residuation*. This operator ($/ : \mathcal{E} \times , \ \mapsto \mathcal{E}$) is not in $\mathcal{E}$, 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 Figure 3 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 $[\![D]\!]$. Similarly, to schedule e, the scheduler can use any of the traces in $[\![e]\!]$. And, to schedule D/e, the scheduler can use any of the traces in $[\![D/e]\!]$. Thus the following must hold for correctness. (We intersect the set with $\mathbf{U}_{\mathcal{E}}$ to eliminate ill-formed traces such as $v\nu$, where v is infinite.)

- $(\{v\nu : v \in [\![e]\!] \text{ and } \nu \in [\![D/e]\!]\} \cap \mathbf{U}_{\mathcal{E}}) \subseteq [\![D]\!]$

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

- $(\forall Z : (\{v\nu : v \in [\![e]\!] \text{ and } \nu \in Z\} \cap \mathbf{U}_{\mathcal{E}} \subseteq [\![D]\!]) \Rightarrow Z \subseteq [\![D/e]\!])$

Put another way, $[\![D/e]\!]$ is the greatest solution to the inequation

$$(\lambda Z : \{v\nu : v \in [\![e]\!] \text{ and } \nu \in Z\} \cap \mathbf{U}_{\mathcal{E}} \subseteq [\![D]\!])$$

Alternatively, $[\![D/e]\!]$ is the set of traces that satisfy the above requirement:

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

Theorem 1 Semantics 7 gives the most general solution to the above inequation. ∎

Theorem 1 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.

3.3 Symbolic Calculation of Residuals

Semantics 7 characterizes 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.

We require that the expressions be in a form such that there is no $\wedge$ or $\vee$ in the scope of the $\cdot$. Such a representation of each expression is possible because the operators $\cdot$, $\vee$, and $\wedge$ distribute over each other. Further, Lemma 17 shows that we can replace all sequences by conjunctions of sequences of length 2, which is extremely convenient for processing, as described in [32]. We assume that all expressions are in this *Two-Sequence Form (TSF).*

Definition 3 $, _D$ gives the alphabet of expression D. $, _0 \triangleq , _\top \triangleq \emptyset$. $, _{E \wedge F} \triangleq , _{E \vee F} \triangleq , _{E \cdot F} \triangleq , _E \cup , _F$. $, _e \triangleq , _{\overline{e}} \triangleq \{e, \overline{e}\}$. Notice that $e \in , _D$ iff $\overline{e} \in , _D$.

Because of the restriction to TSF, 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 $f \equiv f \cdot \top$).

Equation 1 $0/e \doteq 0$

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

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

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

Equation 5 $(e \cdot E)/e \doteq E$, if $e \notin , _E$

Equation 6 $D/e \doteq D$, if $e \notin , _D$

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

Equation 8 $(\overline{e} \cdot E)/e \doteq 0$

We use the operator $\doteq$ in the equations to highlight that it might not be interpreted simply as $\equiv$. Section 4.1 provides further explanation. The above equations are carefully designed to guide the reasoning of a scheduler when it is considering whether to allow an event e. 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.

The dependencies stated in a workflow thus fully describe the state of the scheduler; successive states are computed symbolically through residuation.

Example 8 The reader can verify that the above equations yield all the transitions of Figure 3. ∎

Example 9 Consider Example 4 again. The initial state of the scheduler is given by $S_0 = D_1 \wedge D_2 \wedge D_3 \wedge D_4$. The scheduler can allow *buy* to start first, because the resulting state $S_1 = S_0/s_{buy}$ is consistent. This simplifies to $S_1 = s_{book} \wedge D_2 \wedge D_3 \wedge D_4$. The occurrence of c_{buy} next would leave the scheduler in state $s_{book} \wedge 0 \wedge \top \wedge \overline{s_{cancel}}$, which equals 0, i.e., is inconsistent. However, since $S_2 = S_1/s_{book} = \top \wedge D_2 \wedge D_3 \wedge D_4$ is consistent, s_{book} can occur in S_1. In S_2, *book* can commit, resulting in the state $S_3 = \top \wedge (\overline{c_{buy}} \vee c_{buy}) \wedge (c_{buy} \vee s_{cancel}) \wedge (\overline{c_{buy}} \vee \overline{s_{cancel}})$. 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. ∎

An advantage of TSF is that the dependencies are independently stated, and a workflow $\mathcal{W}$ corresponds to one (albeit large) dependency which is the conjunction of dependencies in $\mathcal{W}$. No additional processing is required in putting the dependencies into an acceptable syntactic form for reasoning. Equation 3 enables the different dependencies to be residuated independently. Observation 2 means that events are independent of dependencies that do not mention them.

Observation 2 $E/f \doteq E$, if $f \notin , _E$ ∎

We now give some additional results. We define $size(E)$ to be the number of nodes in the parse tree of E. The following results show that the equations yield simpler results than the input expression (Observation 3), are convergent (Lemma 5), and produce TSF expressions, which can be input to other equations (Observation 6).

Observation 3 If $D/e \doteq F$ is an equation, then $size(D) \geq size(F)$, $, _D \supseteq , _F$, and $\{e, \overline{e}\} \not\subseteq , _D$ ∎

Lemma 4 The number of operations to compute D/e is linear in $size(D)$. ∎

Lemma 5 For $D \in \mathcal{E}$ and $e \in , ,$ our equations yield exactly one expression in $\mathcal{E}$ for D/e. ∎

Observation 6 If D is in TSF, then D/e is in TSF. ∎

The scheduler can take a decision to accept, reject, or trigger an event only if *no* dependency is violated by that decision. By Observation 2, only dependencies mentioning an event are *directly* relevant in scheduling it. However, 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 algebra and the scheduling algorithm is similar to that between a logic and proof strategies for it. For scheduling, the system accepts, rejects, or triggers events to determine a trace that satisfies all dependencies.

4 Correctness

Correctness involves proving that the equations are *sound* and *complete*. Some, but not all, of our equations are sound in the above model. This is because some of our assumptions are not properly reflected in the model. We motivate enhancements in order to establish soundness of all equations. We first formalize the notions of soundness and completeness. Definitions 4 and 5 formalize entailment and provability, respectively.

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

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

The usual meaning of soundness is that whatever is provable is entailed (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 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 $\doteq$ as $\equiv$, i.e., equivalence or equality of denotations. Lemma 7 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 $D/e \doteq F$ is sound iff $D/e \doteq F$ implies that $D/e \models F$

Lemma 7 Equations 1–6 are sound. ∎

But what about Equations 7 and 8? The following lemma appears discouraging at first. However, it can be corrected when our assumptions are incorporated into the model.

Lemma 8 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. We define 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 legal traces that are also maximal, as defined below.

Assumption 3 An event instance or its complement eventually occurs on a trace.

Intuitively, admissible traces characterize the maximal (and legal) behavior of the given collection of tasks. Let $\mathbf{A}_\Theta$ be the set of all admissible traces on the alphabet Θ. (An alphabet is closed under complementation, i.e., $e \in \Theta$ iff $\overline{e} \in \Theta$.) As before, we identify $\overline{\overline{e}}$ with e.

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 an instance of e or $\overline{e}$. Option (a) clearly violates D, since all traces in $[\![D]\!]$ must mention e' (by Observation 23, see appendix A). Option (b) violates admissibility. Thus, assuming admissibility, we can prove Equation 7 to be sound. We formalize this argument next.

Technically, admissibility captures 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 with different denotations may be interchangeable in certain evaluation contexts.

Example 10 In general $e \not\equiv 0$. However, after e or $\overline{e}$ has occurred, another occurrence of e is impossible. Hence, e is effectively equivalent to 0. ∎

Definition 9 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 , . 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 $A \uparrow e$, 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 9 $A \uparrow e \cap [\![e]\!] = \emptyset$ and $A \uparrow e \cap [\![\overline{e}]\!] = \emptyset$ ∎

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

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

Example 11 Let $A = \mathbf{A}_\Theta$, such that $e \notin \Theta$. Then, $e \approx_A \overline{e}$. Also, $e \approx_A 0$. ∎

Lemma 10 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, which preserves all equalities, but only some of the inequalities. 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 11 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.

Theorem 11 Let $E \approx_A E'$. Then, for all $f \in$, , $E/f \approx_{A\uparrow f} E'/f$. ∎

4.2 Soundness and Completeness

Theorem 11 justifies *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 $D/e \doteq F$ is adm-sound iff for all admissible sets A, $D/e \approx_{A\uparrow e} F$

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 12 If $D/e \doteq F$ is sound, then $D/e \doteq F$ is adm-sound. ∎

Lemma 13 Equations 7 and 8 are adm-sound. ∎

Theorem 14 Equations 1–8 are adm-sound. ∎

Theorem 14 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 14 is important because it enables us to combine the benefits of most general solutions (Theorem 1) with the benefits of efficient computation (Lemma 4).

By Lemma 5, our equations yield just one answer for D/e. Although numerous expressions have the same denotation as $[\![D/e]\!]$, we can show that our equations will 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 $D/e \vdash F'$ and for all admissible sets A, $F \approx_{A[e} F'$

Theorem 15 Equations 1–8 are adm-complete. ∎

5 Arbitrary Tasks

Although we require that event *instances* are not repeated, an event *type* may be instantiated multiple times. Event symbols are interpreted as types; event instances are instantiated from types through parametrization. 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 instances of events of that task. Event IDs are reminiscent of *operation IDs* used to uniquify log operations [18]. It is interesting that this old idea can be adapted to workflows.

We define $\mathcal{E}_P$ as the language generated by (a) substituting '$\mathcal{E}_P$' for '$\mathcal{E}$' in syntax rules 2 and 3, and (b) adding rules 4, 5, and 6 below. We assume a set $\mathcal{V}$ of variables and a set $\mathcal{C}$ constants to use as parameters. Thus, , includes all (ground) event literals and Ξ includes all event atoms. The universe, $\mathbf{U}_{\mathcal{E}}$, depends on , , and includes all traces formed from all possible event instances. $\delta(e)$ gives the number of parameters needed to instantiate e.

Syntax 4 $\Xi \subseteq \mathcal{E}_P$

Syntax 5 $e \in \Sigma$, $\delta(e) = m$, $p_1, \ldots, p_m \in \mathcal{C}$
implies $e[p_1 \ldots p_m], \overline{e}[p_1 \ldots p_m] \in$,

Syntax 6 $e \in \Sigma$, $\delta(e) = m$, $p_1, \ldots, p_m \in (\mathcal{V} \cup \mathcal{C})$
implies $e[p_1 \ldots p_m], \overline{e}[p_1 \ldots p_m] \in \Xi$

The semantics of $\mathcal{E}_P$ is given by replacing Semantic rule 1 by 8 and 9 below. Semantics 9 corresponds to universal quantification. $E(v)$ 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 $[\![f[p_1 \ldots p_m]]\!] =$
$\{\tau \in \mathbf{U}_{\mathcal{E}} : \tau \text{ mentions } f[p_1 \ldots p_m]\}$, $f[p_1 \ldots p_m] \in$,

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.

Parameters can be 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, and scheduling it as before. We redo Example 4 below.

Example 12 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 by making the trip id explicit in each dependency, e.g., (D_1') $\overline{s_{buy}[t]} \vee s_{book}[t]$. Let t be bound to a natural number.

The above workflow is satisfied by infinitely many legal traces, e.g., (τ_6) $s_{buy}[65]s_{book}[65]c_{book}[65]$ $c_{buy}[65]$, and (τ_7) $\overline{s_{buy}[34]}\,\overline{s_{book}[34]}\,\overline{s_{cancel}[34]}$. 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. ∎

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

Example 13 Let b_i denote the event of task T_i entering its critical section and e_i denote the event of T_i 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.

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

Intuitively, $D_M(x,y)$ (M for mutual exclusion) states that is both tasks enter their critical sections, then either T_2 enters its critical section first, or T_1 exits its critical section before T_2 can enter it. For simplicity, we ignore the converse requirement, which applies if T_2 enters its critical section before T_1.

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 $(\overline{b_2[y]} \vee e_1[\hat{x}] \cdot b_2[y])$. Thus the overall dependency becomes $(\forall x, y : x \neq \hat{x} \Rightarrow D_M(x,y)) \wedge (\overline{b_2[y]} \vee e_1[\hat{x}] \cdot b_2[y])$. 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 $(\forall x, y : x \neq \hat{x} \Rightarrow D_M(x,y)) \wedge \top$. 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 $e_1[\hat{x}]$.

For n tasks, we can state dependencies between each pair of tasks. When $b_1[\hat{x}]$ occurs, it causes all the other b_i events to be disabled. In a centralized implementation, all the events seek permission from the central scheduler, which allows no more than one of them at a time. In a distributed implementation, the challenge is in exchanging information among the events so that an event such as $b_1[\hat{x}]$ happens only when it has prohibited the other events. In either case, the mechanism is independent of the number of events. ∎

We implicitly used the inference rule $(\forall z : D(z)) \equiv (\forall z : z \neq \hat{z} \Rightarrow D(z)) \wedge D(z ::= \hat{z})$, for any constant $\hat{z}$. By Observation 2, residuating with an event instance $e[\hat{z}]$ returns the first conjunct unchanged, conjoined with the result of residuating $D(z ::= \hat{z})$ by $e[\hat{z}]$. In this manner, dependencies "grow" to accommodate the appropriate instances explicitly. When the given instantiation is satisfied, it is no longer needed. By uniqueness of instances

attempted, we can safely forget about past instances. Thus, in quiescense only the original dependency may be stored.

To formalize the above reasoning, assume that $\vec{v}$ is a tuple of variables that parametrize the occurrences of e in E. Similarly, $\vec{c}$ is a tuple of constants with which the putative instance of e is instantiated.

Equation 9 $E(\vec{v})/e[\vec{c}] \doteq E(\vec{v}) \wedge (E(\vec{v} ::= \vec{c})/e[\vec{c}])$

Lemma 16 Equation 9 is adm-sound. ∎

Importantly, Example 13 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, parameters 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.

Since unbound parameters are treated as universally quantified, enforceable dependencies may become unenforceable when parametrized, e.g., when they require triggering infinitely many events after a single event occurrence. Determining the safe sublanguages is left to future research.

6 Further Technical Properties

Lemma 17 shows that we need not represent sequences of length greater than 2. This is an easy but important result, because it leads to great simplification in processing, as described in [32].

Lemma 17 $e_1 \cdot \ldots \cdot e_n \equiv e_1 \cdot e_2 \wedge \ldots \wedge e_{n-1} \cdot e_n$ ∎

We show why some alternatives to our approach would not be appropriate. There are subtle relationships between the operators $\wedge$ and $\cdot$, and the constants $\top$ and 1. Sometimes, e.g., [28], $\top$ is the unique maximal element of the algebra and 1 is the unit of the concatenation operator. That is, $[\![\top]\!] = \mathbf{U}$ (as here), whereas $[\![1]\!] = \{\Lambda\}$. However, we have no separate 1, effectively setting $1 \equiv \top$. Intuitively, $[\![E_1]\!] \subseteq [\![E_2]\!]$ means that E_1 is a stronger specification than E_2, because E_1 allows fewer traces. Consequently, any reasonable semantics should validate Lemma 18, which states that if the scheduler satisfies E followed by F, then it satisfies F (and E). Substituting 1 for F yields Observation 19.

Lemma 18 $[\![E \cdot F]\!] \subseteq [\![F]\!]$. ∎

Observation 19 $1 \equiv \top$. ∎

Some approaches require $1 \neq \top$ to *avoid* the results that $(e \vee \overline{e}) \cdot f \equiv f$ and $f \equiv f \cdot (e \vee \overline{e})$. 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 $e \vee \overline{e} \equiv \top$. Since in our approach, $e \vee \overline{e} \neq \top$, setting $1 \equiv \top$ does not have the counterintuitive ramification that it might elsewhere.

By treating $\overline{e}$ as an event, we can record its occurrence before its task terminates. This enables eager scheduling. We have $[\![e]\!] \cap [\![\overline{e}]\!] \neq [\![0]\!]$ and $[\![e]\!] \cup [\![\overline{e}]\!] \neq [\![\top]\!]$. If instead we defined $[\![\overline{e}]\!]$ as the set complement of $[\![e]\!]$, we would obtain $\Lambda \in [\![\overline{e}]\!]$. This would entail that $e \cdot \overline{e} = e$, and $e \cdot \overline{e}$ is satisfiable. By contrast, $e \cdot \overline{e} \approx_A 0$, for admissible A.

If the semantics used only maximal traces, we would not be able to represent event ordering well, especially for complement events. If an event did not occur, we cannot state that it did not occur before or after another event.

We assume a formal complement for each significant event. 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. When a 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 alternative events. Multiway splits are rare in practice.

The admissibility construction enables certain simplifications. Lemma 20 works because the union and intersection of sets of maximal (and legal) traces are also sets of maximal (and legal) traces. Surprisingly, Lemma 21 holds because $\cdot$ inherently assumes nonmaximal traces. Thus the fact that its arguments were equal under maximality carries little significance.

Lemma 20 $E \approx_A E'$ implies that each of the following holds: (a) $E \wedge F \approx_A E' \wedge F$; (b) $F \wedge E \approx_A F \wedge E'$; (c) $E \vee F \approx_A E' \vee F$; and (d) $F \vee E \approx_A F \vee E'$. ∎

Lemma 21 $E \approx_A E'$ does *not* imply that either of the following holds: (a) $E \cdot F \approx_A E' \cdot F$; and (b) $F \cdot E \approx_A F \cdot E'$. ∎

Therefore, we can use Lemma 20 to simplify expressions provided we avoid simplification in the context of a $\cdot$. Fortunately, since by Observation 6, our equations always yield TSF output from TSF input, we never have to worry about an expression in which the $\cdot$ operator is outside of a $\vee$ or $\wedge$. Thus, the simplifications prevented by Lemma 21 would never be needed anyway. So nothing is lost!

7 Overview of the Literature

We consider only the transaction models and approaches that represent them in formal frameworks. Execution environments are discussed in [32].

7.1 Models

Database Approaches Several extended transaction models have been proposed [10]. *Sagas* are composed of several subtransactions or *steps* [14], which can be optimistically

committed, thereby increasing concurrency but reducing isolation. If a step fails, consistency can be restored by compensating the previously committed steps in the reverse order. The *NT/PV* (nested transactions with predicates and views) model is based on multiple coexisting versions of a database [24]. Transactions are modeled as binary relations on database states, rather than as functions. Specifications for transactions are given as precondition and postcondition pairs. Several transaction models can be expressed in the NT/PV model.

ConTracts group a set of transactions into a multitransaction activity [34]. Each ConTract consists of (a) a set of steps: sequential ACID transactions that define the algorithmic aspects of the ConTract, and (b) a script or execution plan: a (possibly parallel) program invoking the steps that defines the structural aspects of the ConTract. ConTracts are forward-recoverable through failures and interruptions. A *long-running activity* is a set of transactions (possibly nested) and other activities [9]. Control and data flow may be specified in a script or with event-condition-action rules. Other important models include *flex transactions* [4] and *cooperating activities* [30].

Organizational Approaches There is also increasing interest in the organizational aspects of workflow management. [21] proposes "trigger modeling" as a technique to capture some of the interrelationships among components of a workflow from an organizational perspective. This model comprises the concepts of activities, events, and actors, and relates them to each other. [5] also relate workflows with organizational modeling.

Remarks on the Above Approaches We believe the above approaches are at a higher level than our approach, which could provide a rigorous infrastructure in which to realize them.

7.2 Representing Models

ACTA ACTA provides a formal framework to specify the effects of transactions on other transactions and objects, but does not address scheduling [6]. In ACTA, 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. $e \in H$ means that e occurs in history H. $e \longrightarrow e'$ (in history H) means that e and e' occur in H, and e precedes e'.

ACTA provides a formal syntax, but not a 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 events. However, certain usages are less clear. For example, consider the statement: "(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 by the formula: $(read_{t_i}[x] \longrightarrow write_{t_j}[x]) \Rightarrow ((Commit_{t_j} \longrightarrow Commit_{t_i}) \Rightarrow (Commit_{t_j} \longrightarrow read_{t_i}[x]))$ [6, p. 363]. This formula uses $read_{t_i}[x]$ to refer to two different event instances, one before $Commit_{t_j}$, and the other after

$Commit_{t_j}$. Thus this formula can hold only vacuously. This is not a major error, but it shows that ACTA is not designed for representing repeating events.

Rule-Driven Transaction Management Klein proposes two primitives for defining dependencies [23]. 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 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.

Temporal Logic Approaches Our previous approach [2] is based on a branching-time temporal logic, CTL (or computation tree logic [12]). 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 Günthör [19]. Günthör'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 This approach is a general-purpose theory of events, but is not designed for specifying and scheduling workflows [28]. Pratt proposes an algebra, and motivates some potentially useful inferences that constrain the possible models for the algebra. Our definitions of complements and admissibility are major extensions beyond [28]. Regular languages, which correspond to linear histories of events, are a class of models. The branching (partially ordered) histories well-known from serializability theory [3] (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 necessarily terminating computations.

Remarks on the Above Approaches The database approaches have useful features, 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 limiting properties common to 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—his inferences are too weak to apply in scheduling. It too does not distinguish between event types and

instances. Our approach goes beyond ACTA in characterizing the reasoning required for scheduling. Roughly, it is to ACTA what unification is to predicate logic.

8 Conclusions

We developed a model-theoretic semantics for events and dependencies that satisfies both workflow intuitions and formal semantics criteria. This semantics provides a basis for checking the consistency and enforceability of dependencies. It 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 strong and succinct answers for various scheduling decisions.

We obtain succinct representations for many interesting dependencies—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 [2]. We have not analyzed the complexity in detail.

More general logic programming techniques for reasoning about integrity constraints and transactions are no doubt important, but the connection has not been explored yet. 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 is on identifying the core scheduling and semantic issues, which will be relevant no matter how the final implementation is achieved.

It has been recently argued that various transaction models can be expressed in existing workflow products, and therefore existing workflow products are sufficient [1]. We reject this position. Compilability into machine code does not make higher-level programming languages irrelevant! Extended transaction and workflow models provide a programming discipline through which computations can be structured. If they could not be translated to lower-level representations, they would not be useful! We agree with [35] that higher-level abstractions are necessary for programming in complex environments.

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

References

[1] G. Alonso, D. Agrawal, A. Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proc. Data Engg (ICDE)*, 1996.

[2] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proc. 19th VLDB*, 134–145, 1993.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[4] O. Bukhres, J. Chen, W. Du, A. Elmagarmid, and R. Pezzoli. InterBase: An execution environment for heterogeneous software systems. *IEEE Computer*, 26(8):57–69, 1993.

[5] C. Bußler and S. Jablonski. An approach to integrated workflow modeling and organization modeling in an enterprise. In *W. Enabling Technologies: Infrastructure for Collab. Enterprises*. IEEE Comp. Soc., 1994.

[6] P. Chrysanthis and K. Ramamritham. ACTA: The SAGA continues. In *[10]*, ch. 10. 1992.

[7] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Trans. Database Syst.*, 19(3):450–491, 1994.

[8] U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao, and M.-C. Shan. Third generation TP monitors: A database challenge. In *Proc. ACM SIGMOD*, 1993.

[9] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Proc. 17th VLDB*, 1991.

[10] A. Elmagarmid, ed. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[11] R. Elmasri and S. Navathe. *Fundamental of Database Systems*. Benjamin Cummings, 2nd ed., 1994.

[12] E. Emerson. Temporal and modal logic. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, volume B. North-Holland, Amsterdam, 1990.

[13] H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[14] H. Garcia-Molina and K. Salem. Sagas. In *Proc. ACM SIGMOD*, 1987.

[15] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Dist. & Par. Databases*, 3(2):119–152, 1995.

[16] D. Georgakopoulos, M. Hornick, and F. Manola. Customizing transactions models and mechanisms in a programmable environment supporting reliable workflow automation. *IEEE TKDE*, 8(4):630–649, 1996.

[17] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[18] J. Gray. The transaction concept: Virtues and limitations. In *Proc. 7th VLDB*, 1981.

[19] R. Günthör. Extended transaction processing based on dependency rules. In *Proc. RIDE-IMS*, 1993.

[20] M. Hsu, ed. *Special Issue on Workflow Systems*, vol. 18(1) of *Bull. IEEE TC Data Engg.* 1995.

[21] S. Joosten. Trigger modelling for workflow analysis. In *Proc. CON: Workflow Management.* R. Oldenbourg Verlag, Munich, 1994.

[22] M. Kamath and K. Ramamritham. Bridging the gap between transaction management and workflow management. In *Proc. NSF Workshop on Workflow and Process Automation in Info. Syst.*, 1996.

[23] J. Klein. Advanced rule driven transaction management. In *Proc. IEEE COMPCON*, 1991.

[24] H. Korth and G. Speegle. Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *ACM Trans. Database Syst.*, 19(3):492–535, 1994.

[25] F. Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *German DB Conf.(BTW)*, 51–70, 1995.

[26] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Syst. J.*, 33(2):326–348, 1994.

[27] G. Nutt. Using workflow in contemporary IS applications. TR CU-CS-663-93, U. Colorado, 1993.

[28] V. Pratt. Action logic and pure induction. In *Proc. Euro. W. Logics in AI*, Springer LNCS 478, 1990.

[29] K. Ramamritham and P. Chrysanthis. A taxonomy of correctness criteria in database applications. *VLDB J.*, 5(1):85–97, 1996.

[30] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, & P. Muth. Towards a cooperative transaction model — the cooperative activity model. In *Proc. VLDB*, 1995.

[31] S. Sarin. Workflow and data management in InConcert. In *Proc. Data Engg (ICDE)*, 1996.

[32] M. Singh. Formal aspects of workflow management, Part 2: distributed scheduling. TR-97-05, Computer Science, NCSU, Raleigh, May 1997. *Extends [33].* `http://www.csc.ncsu.edu/ faculty/ mpsingh/ papers/ databases/ wf_scheduling.ps`.

[33] M. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proc. Data Engg (ICDE)*, 1996.

[34] H. Wächter and A. Reuter. The ConTract model. In *[10]*, ch. 7. 1992.

[35] G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *Comm. ACM*, 35(11):89–99, 1992.

[36] The WfMC. The workflow management coalition (WfMC) reference model. `http://www.aiai.ed.ac.uk/WfMC/`, 1995.

[37] Ziff Communications Company. Computer select – software product spec: Data sources report, 1994.

A Proofs of Important Results

Auxiliary Definitions and Results

Observation 22 $\tau \in [\![E]\!]$ iff $(\forall \upsilon, \nu : \upsilon\tau\nu \in \mathbf{U}_{\mathcal{E}} \Rightarrow \upsilon\tau\nu \in [\![E]\!])$ ∎

Observation 22 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).

Observation 23 If D is a sequence expression and $\tau \in [\![D]\!]$ then
$(\forall i : 1 \leq i \leq length(D) \Rightarrow \tau \in [\![e_i]\!])$. ∎

Observation 24 If D is a sequence expression and $\tau \in [\![D]\!]$ then $length(\tau) \geq length(D)$. ∎

Proof of Lemma 5.

For $D \in \mathcal{E}$ and $e \in$, , D/e is a unique expression in $\mathcal{E}$.

Proof. 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 exactly one equation applies at each stage, the final answer is unique. Thus our equations are convergent.

Proof of Lemma 7.

Equations 1–6 are sound.

Proof. Equations 1 and 2 follow trivially from Semantics 7. Consider Equation 3. $\nu \in [\![(E_1 \wedge E_2)/e]\!]$ iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_{\mathcal{E}} \Rightarrow \upsilon\nu \in [\![E_1 \wedge E_2]\!]))$. This holds iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_{\mathcal{E}} \Rightarrow \upsilon\nu \in [\![E_1]\!] \cap [\![E_2]\!]))$, which is equivalent to $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_{\mathcal{E}} \Rightarrow \upsilon\nu \in [\![E_1]\!]))$ $\wedge$ $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_{\mathcal{E}} \Rightarrow \upsilon\nu \in [\![E_2]\!]))$. But this is equivalent to $\nu \in [\![E_1/e]\!] \cap [\![E_2/e]\!]$.

Equation 4 is the hardest. We can show that $[\![(E_1 \vee E_2)/e]\!] \supseteq [\![E_1/e]\!] \cup [\![E_2/e]\!]$. For the opposite direction, if either E_1 or E_2 is 0 or $\top$, then Equation 4 is trivially satisfied. Let $\nu \in [\![(E_1 \vee E_2)/e]\!]$. Then, since $\langle e \rangle \in [\![e]\!]$, the trace $\langle e \rangle\nu \in [\![E_1 \vee E_2]\!]$. Assume, without loss of generality, that $\langle e \rangle\nu \in [\![E_1]\!]$.

1. Let E_1 be a sequence expression. There are two cases. (a) $E_1 = e \cdot D$. Now, $\langle e \rangle\nu \in [\![e \cdot D]\!]$ implies that (using Observation 22) $\nu \in [\![D]\!]$. Then, by Semantics 4, $(\forall \upsilon \in [\![e]\!] : \upsilon\nu \in [\![e \cdot D]\!])$. Hence, $\nu \in [\![E_1/e]\!]$. (b) $E_1 = f \cdot D$ and $e \neq f$. By Observation 23, $\langle e \rangle\nu \in [\![f \cdot D]\!]$ implies that $\nu \in [\![f \cdot D]\!]$. By Observation 22, $(\forall \upsilon \in [\![e]\!] : \upsilon\nu \in [\![f \cdot D]\!])$. Hence, $\nu \in [\![E_1/e]\!]$.

2. Let E_1 be a disjunction. The proof follows by structural induction.

3. Let E_1 be a conjunction. The proof follows by an application of Equation 3.

Consider Equation 5. Let $\nu \in [\![E]\!]$. Then, by Semantics 4, $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow \upsilon\nu \in [\![e \cdot E]\!])$. Therefore, $\nu \in [\![(e \cdot E)/e]\!]$. Hence, $[\![E]\!] \subseteq [\![(e \cdot E)/e]\!]$. Conversely, let $\nu \in [\![(e \cdot E)/e]\!]$. Then $\langle e \rangle \nu \in [\![e \cdot E]\!]$. By Observation 24, any trace satisfying e must be at least of length 1. Therefore, by Semantics 4, there must a suffix τ of ν that satisfies E. Thus, by Observation 22, $\nu \in [\![E]\!]$. Hence, $[\![(e \cdot E)/e]\!] \subseteq [\![E]\!]$. Thus, $[\![(e \cdot E)/e]\!] = [\![E]\!]$.

Lastly, consider Equation 6. Let $\nu \in [\![D/e]\!]$. Then $\langle e \rangle \nu \in [\![D]\!]$. Since $e, \overline{e} \notin \ , _D$, $\nu \in [\![D]\!]$. Thus, $[\![D/e]\!] \subseteq [\![D]\!]$. Let $\nu \in [\![D]\!]$. Then, by Observation 22, $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow \upsilon\nu \in [\![D]\!])$. Thus, $\nu \in [\![D/e]\!]$ or $[\![D/e]\!] \supseteq [\![D]\!]$. Hence, $[\![D/e]\!] = [\![D]\!]$.

Proof of Lemma 8.

Equations 7 and 8 are *not* sound.

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

Proof of Theorem 11.

Let $E \approx_A E'$. Then, for all $f \in \ , \ , E/f \approx_{A \uparrow f} E'/f$.

Proof. Let $\nu \in (A \uparrow f \cap [\![E/f]\!])$. Then, $\langle f \rangle \nu \in A$. Also, $(\forall \upsilon \in [\![f]\!] : \upsilon\nu \in [\![E]\!])$. Therefore, since $\langle f \rangle \in [\![f]\!]$, we have that $\langle f \rangle \nu \in [\![E]\!]$. Thus $\langle f \rangle \nu \in (A \cap [\![E]\!])$. Since $E \approx_A E'$, $\langle f \rangle \nu \in (A \cap [\![E']\!])$. By Observation 22, $\langle f \rangle \nu \in [\![E']\!]$ implies that $(\forall \upsilon \in [\![f]\!] : \upsilon\nu \in [\![E']\!])$. Thus $\nu \in [\![E'/f]\!]$. Since $\nu \in A \uparrow f$, we obtain $\nu \in (A \uparrow f \cap [\![E'/f]\!])$. Consequently, we have established that $(A \uparrow f \cap [\![E/f]\!]) \subseteq (A \uparrow f \cap [\![E'/f]\!])$. By symmetry, $(A \uparrow f \cap [\![E'/f]\!]) \subseteq (A \uparrow f \cap [\![E/f]\!])$. Thus, $(A \uparrow f \cap [\![E/f]\!]) = (A \uparrow f \cap [\![E'/f]\!])$. Or, $E/f \approx_{A \uparrow f} E'/f$.

Proof of Lemma 13.

Equations 7 and 8 are adm-sound.

Proof. Consider Equation 7. By Semantics 7, $\nu \in [\![(e' \cdot E)/e]\!]$ iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow \upsilon\nu \in [\![e' \cdot E]\!])$. Since $\langle e \rangle \in [\![e]\!]$, this implies that $\langle e \rangle \nu \in [\![e' \cdot E]\!]$. By Observation 24, any trace that satisfies e' must be at least of length 1. Thus, by Semantics 4, a suffix τ of ν exists such that $\tau \in [\![E]\!]$. By Observation 22, $\nu \in [\![E]\!]$. Since we convert expressions to TSF, E is a sequence expression. It is given that $e \in \ , _E$. Therefore, $\nu \in [\![e]\!]$. Thus by Observation 9, $\nu \notin A \uparrow e$. Consequently, $A \uparrow e \cap [\![(e' \cdot E)/e]\!] = \emptyset$, which equals $A \uparrow e \cap [\![0]\!]$. Hence, Equation 7 is adm-sound and similarly Equation 8.

Proof of Theorem 15.

Equations 1–8 are adm-complete.

Proof. By Lemma 5, D/e always evaluates to a unique expression. Let this be F'. Thus, $D/e \vdash F'$ always holds for some F'. Let $D/e \models F$. By Theorem 14, $F' \approx_{A \| e} F$.

Proof of Lemma 16.

Equation 9 is adm-sound.

Proof. By Semantics 9, the denotation of an expression is the intersection of the denotations of all its possible instantiations. By Observation 2, all instantiations except $\vec{c}$ are independent of $e[\vec{c}]$. By admissibility, $e[\vec{c}]$ or $\overline{e}[\vec{c}]$ cannot occur again.

Proof of Lemma 21.

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

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

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

Proof. Let , $= \{e, \overline{e}, f, \overline{f}\}$ and $\Theta =$, . Let $E = e \vee \overline{e}$, $E' = \top$, and $F = f$. Then, $E \approx_A E'$, for admissible A, but $E \cdot F \not\approx_{\mathbf{A}_\Theta} E' \cdot F$. Similarly for the opposite order.

B Information Control Nets

In order to show how our approach can capture workflows expressed in other approaches, we consider a representative graphical notation called *information control nets (ICN)* [27]. ICN can express various control structures. ICN has a formal graphical syntax, but not a model-theoretic semantics.

Figure 4: ICN: An activity

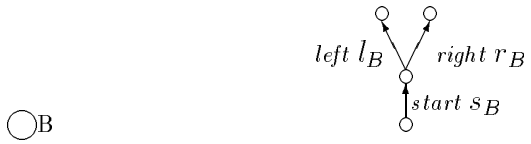


Figure 5: ICN: A decision node

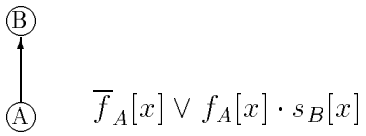
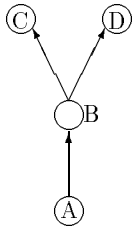


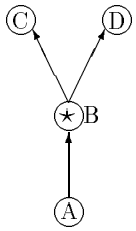
Figure 6: ICN: Control flow

ICN allows two main kinds of nodes—*activities:* those that are modeled as having a single start-finish execution branch, and *decision nodes:* those that are conditional. Figures 4 and 5 show how we model activities and decision nodes. Using these, we can readily capture



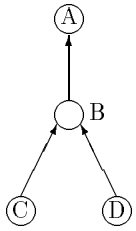
$$\overline{f}_A[x] \vee (f_A[x] \cdot l_B[x] \cdot s_C[x]) \vee (f_A[x] \cdot r_B[x] \cdot s_D[x])$$

Figure 7: ICN: Disjunctive (out) branching



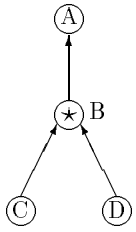
$$(\overline{f}_A[x] \vee f_A[x] \cdot s_C[x]) \wedge (\overline{f}_A[x] \vee f_A[x] \cdot s_D[x])$$

Figure 8: ICN: Conjunctive (out) branching



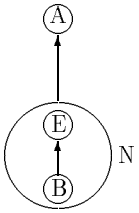
$$(\overline{f}_C[x] \vee f_C[x] \cdot s_A[x]) \wedge (\overline{f}_D[x] \vee f_D[x] \cdot s_A[x])$$

Figure 9: ICN: Disjunctive (in) branching



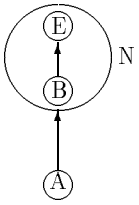
$$\overline{f}_C[x] \vee \overline{f}_D[x] \vee (f_C[x] \cdot f_D[x] \cdot s_A[x]) \vee (f_D[x] \cdot f_C[x] \cdot s_A[x])$$

Figure 10: ICN: Conjunctive (in) branching



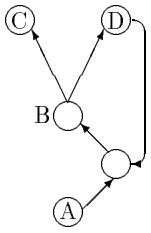
$$\overline{f}_{N.E}[x] \vee f_{N.E}[x] \cdot s_A[x]$$

Figure 11: ICN: Nesting Inbound



$$\overline{f}_A[x] \vee f_A[x] \cdot s_{N.B}[x]$$

Figure 12: ICN: Nesting Outbound



$$\overline{f}_A[x] \vee f_A[x] \cdot s_B[x\ 0]$$
$$(l_B[x\ i] \cdot s_C[x]) \vee (r_B[x\ i] \cdot s_D[x\ i])$$
$$\overline{f}_D[x\ i] \vee (f_D[x\ i] \cdot s_B[x\ (i+1)])$$

Figure 13: ICN: Iteration

the meanings of the constructs for control flow (Figure 6), outbound disjunctive branching (Figure 7), outbound conjunctive branching (Figure 8), inbound disjunctive branching (Figure 9), and inbound conjunctive branching (Figure 10).

ICN allows nodes to be nested. Without loss of generality, we restrict ourselves to nested nodes with a single begin subnode and a single end subnode. For a nested node N, these are referred to as $N.B$ and $N.E$, respectively. Figures 12 and 11 show the translation. Lastly, we consider iteration in Figure 13. The flow of control from A to B initializes the iteration. The iteration ends when C is invoked. Each time D is invoked, it passes control back to B. In the formal expressions, x represents the parameters of interest to the application; i represents the loop counter, which helps uniquify events from different iterations.