Synthesizing Distributed Constrained Events from Transactional Workflow Specifications

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

singh@ncsu.edu

Abstract

Workflows are the semantically appropriate composite activities in heterogeneous computing environments. Such environments typically comprise a great diversity of locally autonomous databases, applications, and interfaces. Much good research has focused on the semantics of workflows, and how to capture them in different extended transaction models. Here we address the complementary issues pertaining to how workflows may be declaratively specified, and how distributed constraints may be derived from those specifications to enable local control, thus obviating a centralized scheduler. Previous approaches to this problem were limited and often lacked a formal semantics.

1 Introduction

Workflows are composite, semantically appropriate activities that execute in heterogeneous environments. In such environments, extremely common in practice, the challenge is to interoperate properly without violating the autonomy of the components. Workflows address this challenge [5]. *Transactional* workflows, which primarily involve database transactions as their component activities, are of great interest. The desirable semantic properties for such workflows are best represented by different extended transaction models [4]. These models overcome the limitations of the traditional, ACID, model [6] by relaxing the atomicity, isolation, and consistency requirements of the ACID model in various ways.

The reported work concentrates on the complementary, distributional, aspects: how workflows of any model may be declaratively specified, and how the given specifications may be executed in a distributed manner. We have developed a general facility—with a language and execution model—to specify and schedule intertask dependencies. Our approach makes no assumptions about the component systems or activities—in particular, they may not be databases or transactions!

We propose a language based on a simple form of process algebra that proves remarkably effective in capturing the desired properties of workflows through intertask dependencies [13]. This language deals with the coordinational aspects, but not with the details of the constituent tasks—this is a major strength. Intertask dependencies can be used to formalize the scheduling aspects of a large variety of, and combinations of, workflow and transaction models. Declarative primitives are useful not only for ease of specification, but also because they facilitate run-time modifications of workflows, e.g., in response to exception conditions. Our approach can express the primitives of [10], which can capture those of [3] and [8]. Our approach is more general than previous scheduling approaches: whereas they are limited to loop-free tasks, we can handle arbitrary ones. Our notation and examples are designed to highlight similarities with the literature where possible; only the essential distinctions are made.

Because of the heterogeneity and inherent distribution of the environments where workflows arise, it is crucial that workflow specifications be converted into distributed executions. In this paper, we provide a temporal semantics for our specification language. We use this semantics to give formally define how *guards* on events may be calculated. The guard on each event is localized on that event and used to control its execution in a distributed manner. Our formal definition is not only easily applicable, but yields several important technical results—about correctness and various independence properties, which facilitate rapid calculations of the guards. Certain related but orthogonal issues, such as event attributes, are discussed informally in [14]; we do not repeat those discussions here. Our approach has been implemented.

Section 2 briefly delineates our execution model and architecture for workflow processing. Section 3 introduces dependencies and shows how workflows can be formally expressed using them. It also discusses some important aspects of scheduling dependencies. Section 4 shows how to synthesize constraints on events as

guards that can be localized on the individual events. This is a natural way to distribute the scheduling of workflows. This section also shows how to execute events using their guards. Next it gives some technical results on the guard computation, to prove correctness and to justify some calculational simplifications. Section 5 shows how parametrized events and tasks of arbitrary structure can be accommodated.

2 The Execution Model

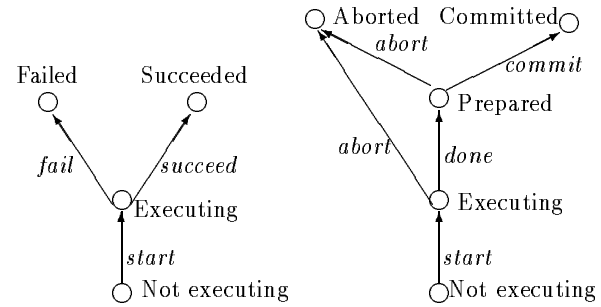


Figure 1: Some Common Task Agents [12]

To best distribute the necessary computations, our approach associates an *agent* with each task. The agent—typically placed close to its task—embodies a coarse description of the task, including only states and transitions (or *events*) that are significant for coordination. Our approach applies to arbitrary agents and to any set of significant events (see Figure 1). The agent performs the important function of interfacing the task with the scheduling system. It informs the system of uncontrollable events like *abort* and requests permission for controllable ones like *commit*. When triggered by the system, it causes appropriate events like *start* in the task. An agent may intercept requests or remote procedure calls made to the task (as in our implementation), or be explicitly informed of all transitions by the task (this requires some reprogramming, but is conceptually simpler).

Our approach maximizes local autonomy, since the task is minimally affected when its agent is inserted into the computational system. The invisible states of the task are not exposed by the agent and its interface is not violated. This clearly precludes certain kinds of interactions among tasks, but that is unavoidable if autonomy is to be preserved.

We instantiate an active entity or *actor* for each event type [1]. Each actor maintains the current *guard* for its event and manages its communications. The guard is a temporal logic expression, which defines the condition under which that event may occur. In the simplest case, when a task agent is ready to make a transition, it attempts the corresponding event. Intuitively, an event can happen only when its guard evaluates to true. If the guard for the attempted event is true, it is allowed right away. Otherwise, it is *parked*. When an event happens, messages announcing its occurrence are sent to actors of other relevant events. These may be at remote sites on the network. When an event announcement arrives, the receiving actor simplifies its guard to incorporate this information. If the guard becomes true, then the appropriate parked event is enabled.

The above is the simplest case. Further complexity is involved for cyclic constraints and for events that may be instantiated multiple times (in tasks of arbitrary structure). Other important situations include those where the given event may have to be proactively triggered or where it cannot be delayed [14].

3 Event Algebra

Our formal language, used to specify acceptable computations or *traces*, is based on an algebra of events [13], which is related to the proposal of [11]. A user would typically be supplied with some graphical notation for specifying workflows, which would be translated into our formal language. Event symbols are the atoms of our language. We also introduce for each event symbol e a symbol $\overline{e}$ corresponding to its complement. Throughout, $\triangleq$ means *is defined as*.

3.1 Syntax

Σ is the set of significant event symbols, Γ is the *alphabet*, and $\mathcal{E}$ is our language of event expressions. A *dependency*, D, is an expression of $\mathcal{E}$. A *workflow*, $\mathcal{W}$, is a set of dependencies. Since previous approaches considered loop-free tasks only, they did not distinguish event types from event instances or tokens. However, this distinction is crucial when arbitrary tasks are considered. For expository ease, we initially take event symbols as naming event tokens. Later, we allow them to be parametrized, i.e., treat them as event types.

Syntax 1 $e \in \Sigma$ implies that $e, \overline{e} \in \Gamma$
Syntax 2 $\Gamma \subseteq \mathcal{E}$
Syntax 3 $E_1, E_2 \in \mathcal{E}$ implies that $E_1 \cdot E_2$, $E_1 + E_2$, $E_1 | E_2 \in \mathcal{E}$
Syntax 4 $0, \top \in \mathcal{E}$

3.2 Semantics

The semantics of $\mathcal{E}$ is given with respect to traces. Technically, traces are finite or infinite sequences of events. Intuitively, each trace describes a fragment of a possible computation in the system. Our semantics thus associates expressions with sets of possible computations. This is important because expressions are used (a) to specify desirable computations and (b) to determine event schedules to realize such computations.

Each trace is a member of the universe, $\mathbf{U}_{\mathcal{E}}$. $\mathbf{U}_{\mathcal{E}}$ contains all traces on Γ such that (a) no trace contains

both an event and its complement, and (b) no event (instance) occurs more than once on any trace. E, ...are expressions; e, ...are events; u, ...are traces; i, ...are indices. For $1 \leq i \leq size(u)$, u_i denotes the ith event in u; it is undefined otherwise. For a trace $u \in \mathbf{U}_{\mathcal{E}}$ and an expression $E \in \mathcal{E}$, $u \models E$ means that u satisfies E. For convenience, we overload event symbols with the events they denote, but our usage is always unambiguous. Traces are written as event sequences enclosed in $\langle$ and $\rangle$ brackets. Thus $\langle e\overline{f} \rangle$ means the trace in which event e occurs followed by the event $\overline{f}$. $\lambda \triangleq \langle\rangle$ is the empty trace.

Definition 1
$\mathbf{U}_{\mathcal{E}} \triangleq \{u : u \in \Gamma^* \cup \Gamma^\omega \wedge ((\exists j : u_j = e) \Rightarrow (\forall j : u_j \neq \overline{e})) \wedge (\forall i, j : (u_i = e \wedge u_j = e) \Rightarrow i = j)\}$

Semantics 1 $u \models f$ iff $(\exists j : u_j = f)$, if $f \in \Gamma$
Semantics 2 $u \models E_1 + E_2$ iff $u \models E_1 \vee u \models E_2$
Semantics 3 $u \models E_1 \cdot E_2$ iff $(\exists v, w : u = vw \wedge v \models E_1 \wedge w \models E_2)$
Semantics 4 $u \models E_1 | E_2$ iff $u \models E_1 \wedge u \models E_2$
Semantics 5 $u \models \top$

This semantics validates various useful properties of the given operators, e.g., associativity of $+$, $\cdot$, and $|$, and distributivity of $\cdot$ over $+$ and over $|$. For convenience, we define the *intension* or *denotation* of an expression as $[\![E]\!] \triangleq \{u : u \models E\}$. Thus the atom e denotes the set of traces at which event e occurs. $E_1 \cdot E_2$ denotes memberwise concatenation of the traces in the denotation E_1 with those for E_2. $E_1 + E_2$ denotes the union of the sets for E_1 and E_2. Lastly, $E_1 | E_2$ denotes the intersection of the sets for E_1 and E_2.

Example 1 Let $\Gamma = \{e, \overline{e}, f, \overline{f}\}$. Then $\mathbf{U}_{\mathcal{E}} = \{\lambda, \langle e\rangle, \langle f\rangle, \langle \overline{e}\rangle, \langle \overline{f}\rangle, \langle ef\rangle, \langle fe\rangle, \langle e\overline{f}\rangle, \langle \overline{f}e\rangle, \langle \overline{e}f\rangle, \langle f\overline{e}\rangle, \langle \overline{e}\overline{f}\rangle, \langle \overline{f}\overline{e}\rangle\}$. Also, $[\![0]\!] = \{\}$ and $[\![\top]\!] = \mathbf{U}_{\mathcal{E}}$. $[\![e]\!] = \{\langle e\rangle, \langle ef\rangle, \langle fe\rangle, \langle e\overline{f}\rangle, \langle \overline{f}e\rangle\}$ and $[\![ef]\!] = \{\langle ef\rangle\}$. One can verify that $[\![e + \overline{e}]\!] \neq \top$ and $[\![e|\overline{e}]\!] = 0$. ∎

As running examples, we use two dependencies due to [10]. These are well-known in the literature: they have been used in [2, 8] and are related to the primitives in [3]. In Klein's notation, $e \rightarrow f$ means that if e occurs then f also occurs (before or after e). This may be formalized as $(\overline{e} + f)$—see Example 2. Klein's $e < f$ means that if both events e and f happen, then e precedes f. This may be formalized as $(\overline{e} + \overline{f} + e \cdot f)$—see Example 3.

Example 2 Let $D_{\rightarrow} = \overline{e} + f$. Let v be a trace that satisfies $D_{\rightarrow}$ (i.e., $v \models D_{\rightarrow}$). v satisfies e, iff e occurs on v, whence $\overline{e}$ cannot occur on v. Hence, v must satisfy f. There is no constraint as to the relative order of e and f. ∎

Example 3 Let $D_{<} = \overline{e} + \overline{f} + e \cdot f$. Let v be a trace such that $v \models D_{<}$. v satisfies both e and f iff both e and f occur on v. Thus neither $\overline{e}$ nor $\overline{f}$ can occur on v. Hence, to satisfy $D_{<}$, v must satisfy $e \cdot f$, which requires that an initial part of v satisfy e and the remainder satisfy f. In other words, if e and f both occur on v, then e precedes f. ∎

Example 4 Consider a workflow which attempts to *buy* an airline ticket and *book* a car for a traveler. The key semantic requirement is that both or neither task should have an effect. Mutual (e.g., two-phase) commit protocols cannot be executed, since the airline and car rental agency are different enterprises and possibly their databases don't have a visible precommit state. We can use the fact that there are several mutually indistinguishable instances of plane seats and rental cars to relax the scheduling requirements.

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

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

3.3 Enforcing Dependencies

The scheduler must ensure that any trace that is realized satisfies all stated dependencies. An important component of the state of the scheduler is determined by the dependencies it is enforcing, because they specify the traces it must allow. As events occur, the possible traces get narrowed down. An event e occurs when the scheduler (a) accepts that event if requested by the task agent in which that event arises, (b) triggers that event in the task agent on its own accord, or (c) rejects the complement of that event if the complement is requested by the task agent. The scheduler has no choice but to accept nonrejectable events like *abort* [14]. Consider how the state of the scheduler evolves

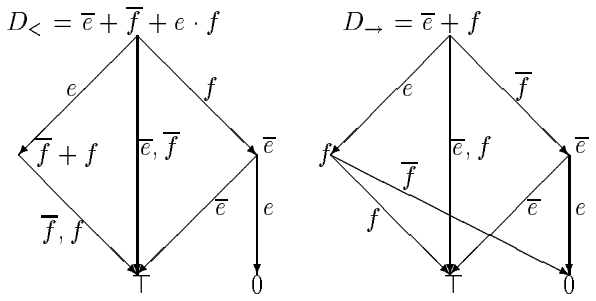


Figure 2: Scheduler States and Transitions Represented Symbolically

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

Example 5 Figure 2 shows the state changes for dependencies $D_< = (\overline{e} + \overline{f} + e \cdot f)$ and $D_\rightarrow = (\overline{e} + f)$. If the complement of events e or f happens, then $D_<$ is necessarily satisfied. If e happens, then either f or $\overline{f}$ can happen later. But if f happens, then only $\overline{e}$ must happen afterwards (e cannot be permitted any more, since that would mean f precedes e). Similarly, for $D_\rightarrow$ the scheduler can permit $\overline{e}$ or f to happen right away, but if e happens first, it must be followed by f and if f happens first, it must be followed by $\overline{e}$. ∎

3.4 Residuation

We now formalize the requirements illustrated above. The determination of whether and when an event e may occur relative to some dependency must be based on

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

Our approach relies on the fact that when dependencies are expressed in $\mathcal{E}$, then the above decisions can be made symbolically and efficiently. In effect, the scheduler's states are captured initially by the stated dependencies, and throughout processing, by expressions derived from them. The operation of computing the resulting set of traces corresponds to *residuation* in our algebra. We define residuation as an operator / (which is not in $\mathcal{E}$, so that it cannot be used to specify workflows—e.g., e/f is not a well-formed dependency).

Semantics 6 $v \models E_1/E_2$ iff $(\forall u : u \models E_2 \Rightarrow (uv \in \mathbf{U}_\mathcal{E} \Rightarrow uv \models E_1))$

In our usage, E_1 is a general expression and E_2 is an event. The above model-theoretic definition clearly meets the criteria motivated above. It is possible to characterize residuation symbolically by a set of equations or rewrite rules. The equations below assume that the given expression is in a form where there is no | or + in the scope of $\cdot$. This holds for CNF, which can be obtained by repeated application of the distribution laws. Thus E in equations 3, 7, and 8 must be an atom or a sequence expression. For simplicity, we identify $\overline{\overline{e}}$ with e. Γ_E is the set of events mentioned in E, and their complements.

Residuation 1 $0/E = 0$
Residuation 2 $\top/E = \top$
Residuation 3 $(e \cdot E)/e = E$
Residuation 4 $(E_1 + E_2)/e = (E_1/e + E_2/e)$
Residuation 5 $(E_1|E_2)/E = ((E_1/E)|(E_2/E))$
Residuation 6 $E/e = E$, if $e, \overline{e} \notin \Gamma_E$
Residuation 7 $(e' \cdot E)/e = 0$, if $e \in \Gamma_E$ and $e \neq e'$
Residuation 8 $(e' \cdot E)/e = 0$, if $\overline{e} \in \Gamma_E$

Example 6 In Figure 2, we can verify that residuating each state label by any of its out-edge labels yields the label of the next state. For instance, $(\overline{e}+\overline{f}+e\cdot f)/e = ((\overline{e}/e)+(\overline{f}/e)+(e\cdot f/e)) = (0+\overline{f}+f) = (\overline{f} + f)$. Similarly, $(\overline{e} + f)/\overline{f} = \overline{e}$. ∎

Theorem 1 Equations 1 through 8 are sound. The proof requires an interesting quotient construction, which is developed in [13]. ∎

4 Guards on Events

The above development naturally leads to a centralized *dependency-centric* scheduler, in which dependencies are explicitly represented in one place in the system. However, that approach would suffer from all the problems attendant to centralization. Rather than invent a distributed dependency-centric scheduler, we designed a distributed *event-centric* one. This enables all the information pertinent to decisions about an event to be localized on that event. An implementation of this approach requires

- determining the conditions on the events by which decisions can be taken on their occurrence
- setting up messages so that the relevant information flows from one event to another
- providing an algorithm by which the different messages can be assimilated.

We satisfy these prerequisites through the symbolic computation of *guards* on events. In order to properly convert from the dependency representations to the event representations, we consider all possible computations relevant to each dependency to determine the various conditions in which a given event can occur: (a) what should have happened already, (b) what should *not* have happened yet, and (c) what should be guaranteed to happen eventually. From these conditions, we determine the *guard* of the event. The guard is the weakest condition that must hold for the event to occur, so that correctness is preserved. It turns out that in the dependencies that are most often of interest in specifying common workflows, the guards of

the participating events are succinct temporal expressions. Before we specify how guards are obtained, we must augment the underlying language so as to have a sufficiently expressive formalism.

4.1 Temporal Logic

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

Syntax 5 $\mathcal{E} \subseteq \mathcal{T}$
Syntax 6 $E_1, E_2 \in \mathcal{T}$ implies that $\Box E_1$, $\Diamond E_1$, $\neg E_1$, $E_1 + E_2$, $E_1|E_2$, $E_1 \cdot E_2 \in \mathcal{T}$

The semantics of $\mathcal{T}$ is given with respect to a trace (as for $\mathcal{E}$) *and* an index into that trace. The semantics of $\mathcal{T}$ enables the progress along a given computation to be exactly characterized and used to determine the scheduler's action at each event. Our semantics is essentially the standard one for linear temporal logics, except that our traces are sequences of events, not of states. For $i \geq 0$, $u \models_i E$ means that E is satisfied on u at index i. For $j \geq 0$, u^j is the suffix of u from index j. Syntax rule 5 can be thought as a simple type coercion from $\mathcal{E}$ to $\mathcal{T}$, which justifies fresh semantic definitions for expressions in $\mathcal{E}$.

The traces used in the semantics of $\mathcal{L}$ are further restricted to be maximal. Specifically, the top-level calls to the semantics of $\mathcal{L}$ are made with maximal traces; the recursive calls may not be so. Maximality is captured as follows: $\mathbf{U}_\mathcal{T} \triangleq \{u : u \in \mathbf{U}_\mathcal{E} \wedge (\forall e \in \Gamma : (\exists j : u_j = e \vee u_j = \overline{e}))\}$. Thus $\lambda \notin \mathbf{U}_\mathcal{T}$, for $\Gamma \neq \emptyset$ (we assume $\Gamma \neq \emptyset$).

Semantics 7 $u \models_i f$ iff $(\exists j \leq i : u_j = f)$, if $f \in \Gamma$
Semantics 8 $u \models_i E_1 + E_2$ iff $u \models_i E_1$ or $u \models_i E_2$
Semantics 9 $u \models_i E_1 \cdot E_2$ iff $(\exists j \leq i : u \models_j E_1 \wedge u^j \models_{i-j} E_2)$
Semantics 10 $u \models_i E_1|E_2$ iff $u \models_i E_1 \wedge u \models_i E_2$
Semantics 11 $u \models_i \top$
Semantics 12 $u \models_i \Box E$ iff $(\forall j : i \leq j \Rightarrow u \models_j E)$
Semantics 13 $u \models_i \Diamond E$ iff $(\exists j : i \leq j \wedge u \models_j E)$
Semantics 14 $u \models_i \neg E$ iff $u \not\models_i E$

$\Box E$ means that E will always hold; $\Diamond E$ means that E will eventually hold (thus $\Box e$ entails $\Diamond e$); and $\neg E$ means that E does not (yet) hold. Semantic condition 7 validates the *stability* of events: if an event is satisfied at a given index, it is satisfied at all future indices as well. Because of stability, we have $\Box e = e$. However, but $\Box\neg e \neq \neg e$ still holds.

Example 7 Let $u = \langle efg \ldots \rangle$ be a trace in $\mathbf{U}_\mathcal{T}$. One can verify that $u \models_0 \Diamond g$; $u \models_0 \neg e|\neg f|\neg g$; $u \models_0 \Diamond(f \cdot g)$; $u \models_1 \Box e|\neg f|\neg g$; $u \not\models_1 e \cdot g$; and $u \models_2 e \cdot g$. ∎

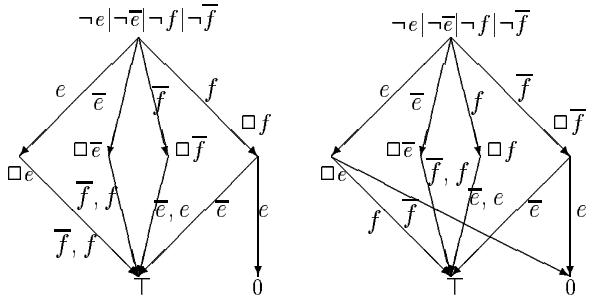
Example 8 Figure 3 shows the possible traces for $\Gamma = \{e, \overline{e}\}$. For larger alphabets, the set of traces is larger, but there is no conceptual difference. On different possible traces, e or $\overline{e}$ may occur. Initially, neither e nor $\overline{e}$ has happened, so traces $\langle e \rangle$ and $\langle \overline{e} \rangle$ both satisfy $\neg e$ and $\neg\overline{e}$ at index 0. Trace $\langle e \rangle$ satisfies $\Diamond e$ at 0, because event e will occur on it; similarly, trace $\langle \overline{e} \rangle$ satisfies $\Diamond\overline{e}$ at 0. After event e occurs, $\Box e$ becomes true, $\neg e$ becomes false, and $\Diamond e$ and $\neg\overline{e}$ remain true. The table in figure 3 illustrates the following results: (a) $\Box e + \Box\overline{e} \neq \top$—neither e nor $\overline{e}$ may have occurred at certain times, e.g., initially; (b) $\Diamond e + \Diamond\overline{e} = \top$—eventually either e or $\overline{e}$ will occur; (c) $\Diamond e|\Diamond\overline{e} = 0$—both e and $\overline{e}$ will not occur; (d) $\Diamond e + \Box\overline{e} \neq \top$—initially, $\overline{e}$ has not happened, but e may not be guaranteed; (e) $\neg e$ is the boolean complement of $\Box e$ ($\neg e + \Box e = \top$ and $\neg e|\Box e = 0$); and (f) $\neg e + \Box\overline{e} = \neg e$ ($\Box\overline{e}$ entails $\neg e$). The above and allied results were our main motivation in designing the formal semantics of $\mathcal{T}$. ∎

	$\langle e \rangle, 0$	$\langle e \rangle, 1$	$\langle \overline{e} \rangle, 0$	$\langle \overline{e} \rangle, 1$
$\neg e$	✓		✓	✓
$\Box e$		✓		
$\Diamond e$	✓	✓		
$\neg\overline{e}$	✓	✓	✓	
$\Box\overline{e}$				✓
$\Diamond\overline{e}$			✓	✓

Figure 3: Temporal Operators Related to Events

By enriching the formal language, we are able to make certain useful distinctions explicit. Two opposite intuitions about events can be seen. One, e is true precisely when e has occurred and will remain occurred forever. Two, e is true precisely when it is definite that e will occur, even if e has not occurred yet. The former corresponds to the $\Box$ operator under stability. The latter corresponds to the $\Diamond$ operator.

4.2 Computing Guards



$D_< = \overline{e} + \overline{f} + e \cdot f$ $\quad$ $D_\rightarrow = \overline{e} + f$

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

We now show how to formally calculate $\mathsf{G}(D, e)$, the guard on e due to dependency D. Each dependency is true of a set of traces. For each trace, we consider whether it allows the given event and if so, what must have occurred before, what must not have occurred before, and what must occur after the given event.

Through a small insight, we are able to replace certain sequences by conjunctions—the guards on other events ensure that no spurious traces are realized. Below, the first term reflects the case when e occurs first, and the remaining terms reflect other events occurring first. Here $\Gamma_{D^e} \triangleq \Gamma_D - \{e, \overline{e}\}$.

Definition 2 $\mathsf{G}(D, e) \triangleq (\Diamond(D/e)|\bigwedge_{f \in \Gamma_{D^e}} \neg f) + \Sigma_{f \in \Gamma_{D^e}}(\Box f|\mathsf{G}(D/f, e))$

Figure 4 illustrates our intuition. Each picture encodes all traces on $\Gamma = \{e, \overline{e}, f, \overline{f}\}$. The initial node in the dependency representation is labeled $\neg e|\neg\overline{e}|\neg f|\neg\overline{f}$ to indicate that no event has occurred yet. The nodes in the middle layer are labeled $\Box e$, etc., to indicate that the corresponding event has occurred. To reduce clutter, labels like $\Diamond e$ and $\neg e$ are not shown after the initial state. To compute the guard on an event, we sum the contributions of different paths on which it occurs. The contribution of a path ending in 0 is 0. For a path ending in $\top$, the contribution is the label of the pre-state for the event conjoined with the label of the future post-state. In each case, we delete the event and its complement from the label.

Example 9 We use the above definition to compute guards for some simple dependencies.

1. $\mathsf{G}(\top, e) = (\top|\Diamond\top) + 0 = \top$.
2. $\mathsf{G}(0, e) = (\top|\Diamond 0) + 0 = 0$.
3. $\mathsf{G}(e, e) = (\top|\Diamond\top) + 0 = \top$.
4. $\mathsf{G}(\overline{e}, e) = (\top|\Diamond 0) + 0 = 0$.
5. $\mathsf{G}(D_<, \overline{e}) = (\neg f|\neg\overline{f}|\Diamond\top) + (\Box f|\mathsf{G}(\overline{e}, \overline{e})) + (\Box\overline{f}|\mathsf{G}(\top, \overline{e}))$. It is easy to see that this reduces to $\top$.
6. $\mathsf{G}(D_<, e) = (\neg f|\neg\overline{f}|\Diamond(\overline{f} + f)) + (\Box f|\mathsf{G}(\overline{e}, e)) + (\Box\overline{f}|\mathsf{G}(\top, e))$. Using $\Diamond(\overline{f} + f) = \top$ and the above two guards, this reduces to $(\neg f|\neg\overline{f}) + \Box\overline{f}$, which reduces to $\neg f + \Box\overline{f}$. Hence, $\mathsf{G}(D_<, e) = \neg f$.
7. Similarly, $\mathsf{G}(D_<, \overline{f}) = \top$.
8. $\mathsf{G}(D_<, f) = (\neg e|\neg\overline{e}|\Diamond\overline{e}) + \Box e + \Box\overline{e}$, which simplifies to $\Diamond\overline{e} + \Box e$.

Consider the guards from $D_<$. Event $\overline{e}$ can occur at any time, and e can occur if f has not yet happened (possibly because it will never happen). Similarly, $\overline{f}$ can occur anywhere, but f can occur only if e has occurred or $\overline{e}$ is guaranteed. ∎

The guard on an event e due to a workflow $\mathcal{W}$ is the conjunction of the guards due to the dependencies in $\mathcal{W}$ that mention e—see theorems ?? and ?? in section 4.4.

4.3 Execution By Guard Evaluation

Execution with guards is conceptually straightforward. The guard on an event e is evaluated when it is submitted. Evaluation usually means checking if the guard is $\top$, in which case e can occur. This results in a message $\Box e$ being sent to all events that depend on e. In other cases (see below), a $\Diamond e$ may be sent to indicate *promising*. We use a limited set of proof rules to reduce guards when messages arrive. $\Box e$ reduces subexpressions $\Box e$ or $\Diamond e$ to $\top$, and $\neg e$ to 0. However, $\Box e$ and $\neg e$ are unaffected when $\Diamond e$ is received. And, $\Box e$ and $\Diamond e$ reduce to 0 and $\neg e$ to $\top$, when $\Box\overline{e}$ or $\Diamond\overline{e}$ is received.

Example 10 Consider the guards due to $D_<$ given in Example 9. If f is attempted first, its guard is not $\top$, so it is parked. Event $\overline{e}$ can occur right away when attempted. When f is informed of this, its guards reduces to $\top$, and it is allowed to occur. ∎

However, greater complexity is required when the guard on an event e includes a subexpression of the form $\neg f$, for which the scheduler must ensure that events e and f agree whether f has happened or not. A related situation arises when guards of two events refer to each other as in Example 11. In that case the events can occur only if both agree that each will happen, thereby enabling the other to happen.

Example 11 Consider dependency $D_\rightarrow$ and its "transpose" $D_\rightarrow^T = (\overline{f} + e)$. These result in e's guard being $\Diamond f$ and f's guard being $\Diamond e$. Thus e requires $\Diamond f$, while f requires $\Diamond e$. ∎

The scheduler must achieve consensus among the events regarding their mutual occurrence or order. Our implemented approach is to setup messages so that one of the events makes a conditional promise to the other, using which the latter can proceed, generate a message, and thereby cause the first to discharge its promise [14].

4.4 Results on Guard Calculations

The main advantage of formalizing the guard calculations is that it enables us to prove the correctness of the resulting executions. The above definition also allows us to prove certain helpful theorems, which justify the simplification of the guard computations in many cases of interest. We state some results next; proofs have been omitted for reasons of space.

Theorem 2 $\mathsf{G}(D + E, e) = \mathsf{G}(D, e) + \mathsf{G}(E, e)$, if $\Gamma_D \cap \Gamma_E = \emptyset$ ∎

Lemma 3 $\mathsf{G}(D, e) = \neg g|\mathsf{G}(D, e) + (\Box g)|\mathsf{G}(D/g, e)$, for any event $g \notin \{e, \overline{e}\}$. ∎

Theorem 4 $\mathsf{G}(D|E, e) = \mathsf{G}(D, e)|\mathsf{G}(E, e)$, if $\Gamma_D \cap \Gamma_E = \emptyset$ ∎

$\Pi(D)$ is the set of computations that satisfy D. Notice that $\mathsf{G}(e_1 \ldots e_k \ldots e_n, e_k) = \Box e_1|\ldots|\Box e_{k-1}|\neg e_{k+1} \ldots \neg e_n|\Diamond(e_{k+1} \ldots e_n)$. Lemma 5 formalizes the intuition behind Definition 2 as explained in section 4.2.

Definition 3 $\Pi(D) \triangleq \{\rho : \rho = e_1 \ldots e_n \wedge ((D/e_1)/\ldots)/e_n = \top\}$

Lemma 5 $\mathsf{G}(D, e) = \Sigma_{(e_1 \ldots e_k \ldots e_n) \in \Pi(D) \wedge e = e_k} \mathsf{G}(e_1 \ldots e_k \ldots e_n, e)$ ∎

This formalization helps in proving the following crucial result.

Definition 4 A workflow $\mathcal{W}$ generates a trace u iff $(\forall j : u_{j+1} = e \Rightarrow (\forall D \in \mathcal{W} : u \models_j \mathsf{G}(D, e)))$.

Theorem 6 $\mathcal{W}$ generates u iff $(\forall D \in \mathcal{W} : u \models D)$. ∎

5 Generalizing to Arbitrary Tasks

So far, our approach has considered specific event instances only. This is clearly too restrictive. We modify the syntax of $\mathcal{E}$ and $\mathcal{T}$ to parametrize event atoms by attaching a tuple of all relevant parameters. Commonly relevant parameters include task ids, database keys, and other unique ids. We now consider two different ways of scheduling parametrized dependencies to handle intra- and inter-workflow requirements.

5.1 Parametrized Workflows

The simplest uses of parameters are *within* given workflows, where the parameters on different events are identical, or at least closely related. Attempting some key event binds the parameters of all events, thus instantiating the workflow afresh. The workflow is then scheduled as described in previous sections. We redo Example 4 below.

Example 12 Now we use *cid* as the customer id to parametrize the workflow. The parameter *cid* 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. (1) $\overline{s_{buy}[cid]} + s_{book}[cid]$, (2) $\overline{c_{buy}[cid]} + c_{book}[cid] \cdot c_{buy}[cid]$, and (3) $\overline{c_{book}[cid]} + c_{buy}[cid] + s_{cancel}[cid]$. ∎

When domain-specific unique identifiers are not available, we can uniquely identify each event by (a) the identity of the agent in which it occurs and (b) the count of the event. Thus each agent can maintain a counter for each event (or a single counter for all events) and increment it whenever it attempts an event or an event is triggered in it. For practical reasons, we consider events in different agents to be different types and have a separate actor for each of them. Different instances of an event are different tokens of the same type.

The notion of event IDs is well-known in transaction processing: e.g., [7] mentions using *operation IDs* in a recovery protocol to ensure uniqueness of operations recorded on persistent storage. Our contribution is in making this obvious notion compatible with intertask dependencies, and showing how to reason over the latter in the presence of parameters.

5.2 Arbitrary Tasks

More challenging cases arise when different events have unrelated parameters. Such cases occur in the specification of concurrency control requirements across workflows or transactions.

Importantly, the event IDs do *not* need to depend on the structure of the associated task agent. 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, if we can handle parameters correctly, we can handle arbitrary tasks correctly!

The unbound parameters in a guard expression are treated as if universally quantified. This means that certain enforceable dependencies may become unenforceable when parametrized, e.g., when they require an infinitely many events to be triggered because of a single event occurrence.

Example 13 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. (Concurrency control requirements such as serializability are similar, except that they impose a uniform order over data access events.)

$b_2[y] \cdot b_1[x] + \overline{e_1[x]} + \overline{b_2[y]} + e_1[x] \cdot b_2[y]$ ∎

We lack the space to elaborate on the required reasoning in detail. However, the following example has all the essential ingredients. It shows how guards on parametrized events can grow and shrink as necessary. Under appropriate conditions, a guard is resurrected. This is essential for dealing with tasks that are not loop-free, or of any fixed structure: indeed, Example 13 makes no assumptions about the conditions under which the two tasks attempt to enter or exit their critical sections.

Example 14 Let the guard on $e[x]$ be $(\neg f[y] + \Box g[y])$. The variable y is not bound. Assume that initially none of the $f[y]$'s has happened. Therefore, $\neg f[y]$ is true, for all y. Thus $e[x]$ can go ahead when it is attempted. Suppose $f[\hat{y}]$ happens, for a particular $\hat{y}$. This reduces the guard on $e[x]$ to $\Box g[\hat{y}] | (\neg f[y] + \Box g[y])$, which is neither $\top$ nor 0. Now if $e[x]$ is attempted, it must wait. Later when $\Box g[\hat{y}]$ arrives at $e[x]$, the guard on $e[x]$ is reduced back to $(\neg f[y] + \Box g[y])$. Then $e[x]$ is once again enabled. ∎

One might wonder about the value of parametrization to our formal theory. If we cared only about intra-workflow parametrization, we 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.

6 Conclusions

A prototype of our system has been implemented [15]. Our approach is provably correct, and applies to many useful workflows in heterogeneous, distributed environments. Much of the required symbolic reasoning can be precompiled, leading to efficiency at runtime. Although we begin with lazy specifications, which characterize entire traces as acceptable or unacceptable, we setup our computations so that information flows as soon as it is available, and activities are not unnecessarily delayed. We believe this will lead to good scalability.

Klein's approach, which is only sketchily described in [10], is the closest to our approach in that it is event-centric and distributed. However, it is limited to loop-free tasks, and doesn't handle event attributes. Klein assumes that the structure of tasks can be expressed in a star-free regular expression—hence the restriction to loop-free events. Günthör's approach is based on temporal logic, but is centralized [8]. These approaches are somewhat *ad hoc* in their details. Lastly, our previous approach, which constructs finite automata for dependencies, is centralized [2]. It avoids generating product automata, but the individual automata themselves can be quite large. The past approaches cannot express or process complex dependencies as easily as described here.

The underlying execution mechanism should provide a consistent view of the temporal order of events. The compilation phase can detect these conditions and add messages to ensure that there are no problems. The consensus requirements we mentioned are actually too strong. In some cases, because of the restricted language in which dependencies are expressed, certain consensus requirements can be eliminated without loss of correctness. We do not discuss these additional optimizations here. An important conceptual point is that mutual dependencies in workflows are included only when necessary, not because the transaction model forces them upon the programmer. Thus inefficiencies are suffered only when unavoidable. Declarative specifications enable modification of the workflows to suit semantic and performance requirements, e.g., so that cross-system dependencies can be removed.

We showed how we can handle parametrized dependencies within our theory. Although it took no significant effort in our approach to do so, this is an important conceptual advance—the simplicity only shows the intuitiveness of our approach. We believe that extralogical parameters can be added to the previous approaches, but to do it as above might be a challenge for them. It would be helpful in our approach to allow the parameters to be restricted in various ways to capture more details of information flow. This should lead into general constraint languages, and is a connection that we plan to explore.

References

[1] Gul A. Agha. *Actors*. MIT Press, Cambridge, MA, 1986.

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

[3] Panos Chrysanthis and Krithi Ramamritham. ACTA: The SAGA continues. In *[4]*, chapter 10. 1992.

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

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

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

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

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

[9] Won Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and Beyond*. ACM Press (Addison-Wesley), New York, NY, 1994.

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

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

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

[13] Munindar P. Singh. Semantical considerations on workflows: Algebraically specifying and scheduling intertask dependencies. In *Proceedings of the 5th International Workshop on Database Programming Languages (DBPL)*, September 1995.

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

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