Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition[*] [†]

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7535, USA
singh@ncsu.edu

notes by vidal

ABSTRACT

We address the problem of constructing multiagent systems by coordinating heterogeneous, autonomous agents, whose internal designs may not be fully known. A major application area is Web service composition. We develop an approach that (a) takes declarative specifications of the desired interactions, and (b) automatically enacts them. Our approach is based on temporal logic, has a rigorous semantics, and yields a naturally distributed execution.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

General Terms

Design, Verification

Keywords

Temporal logic; service composition

1. INTRODUCTION

Web services enable application development over the Web by supporting program-to-program interactions. Current standards enable the description, discovery, and invocation of services [4]. Services are most valuable when composed in novel ways. However, current approaches are limited to hard-coded, procedural abstractions for composition. These cannot handle interactions flexibly enough to support realistic composition scenarios.

Two key aspects of realistic scenarios are that the services are inherently autonomous and their compositions are often long-lived. For example, a long-lived interaction occurs in e-business when you try to change an order because of some unexpected conditions or try to get a refund for a faulty product. Further, specific configurations may impose their own requirements. Even short-lived settings may require flexibility, e.g., routing an order differently in some cases or checking if the service requester is authenticated and properly authorized before accepting its order.

The Business Process Execution Language (BPEL), a leading traditional approach, specifies compositions of services in the form of a workflow, where the workflow is captured procedurally in terms of branch and join primitives indicating the orderings of different tasks [3]. BPEL specifications orchestrate invocations of services from a central engine, but do not readily accommodate the kinds of flexibility described above.

Recent approaches inspired from artificial intelligence are promising. These capture richer representations of services and support planning. However, the service scheduling component of DAML-S [1] offers a representation at the level of branch and join primitives. Some recent work applies planning and other agent techniques for domain-specific reasoning about services, e.g., [11]. Another family of approaches exploits high-level multiagent abstractions such as commitments [16, 17]. Such abstractions characterize interactions among agents in terms of the commitments of the different parties to one another. The agents can reason about their commitments to others and others' commitments to them in order to decide how to act. However, the actions of the agents must be coordinated so that the overall computation proceeds as desired. And the desired coordination might itself be negotiated and be the subject of commitments. This is where the present approach comes in.

This paper takes a multiagent systems stance on the problem of service composition, emphasizing the distributed computing underpinnings of agents. Agents can apply naturally for composition provided we develop rigorous yet easy-to-use methods to capture complex interactions. The approach of this paper is to achieve service composition by postulating an agent for each service to be composed and by declaratively specifying workflows for the agents. Declarative specifications can support greater flexibility than can traditional primitives of the ilk of branch and join. The agents can dynamically and with maximum flexibility bring about the right events with just enough constraints to satisfy the stated workflow. This idea applies for developing multiagent systems in general, not just those for Web services. A generic facility for agent coordination can be used by a higher-level reasoner to bring about the interactions that it deems fit without having to contend with the details of distributed events.

For reasons of space, more extensive motivation and technical details are deferred to [14]; however, this paper is self-contained. Section 2 describes our specification language. Section 3 shows how flows are enacted. Section 4 formalizes enactment via a series of sophisticated and provably correct strategies. Section 5 discusses some literature and directions.

[*] This research was supported by the National Science Foundation under grants IIS-9624425 and DST-0139037.

[†] I am indebted to the anonymous reviewers for their comments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'03, July 14–18, 2003, Melbourne, Australia.
Copyright 2003 ACM 1-58113-683-8/03/0007 ...$5.00.

2. SPECIFYING SERVICE COMPOSITION

We now discuss how to compose services by specifying constraints on events (described below) of different services. $\mathcal{I}$ is an event-based linear temporal logic—effectively, propositional logic augmented with the *before* $(\cdot)$ temporal operator. *Before* is formally a dual of the more conventional "until" operator; it was studied as the "chop" operator [2]. Because of its similarity with sequence composition in imperative programming languages, the chop facilitates compositional specifications. Here, *before* is used to state minimal ordering requirements so as to facilitate compositionality.

$\mathcal{I}$ can capture a remarkable variety of coordination requirements, yet be compiled and executed in a distributed manner. $\mathcal{I}$ incorporates a set of *significant events*—roughly, transitions of different services that are externally visible and significant for coordination. A *dependency*, D, is an expression in $\mathcal{I}$. A *workflow*, $\mathcal{W}$, is a set of dependencies.

We specify the syntax of $\mathcal{I}$ through the following BNF with start symbol *I*. Here, *slant* indicates nonterminals; $\longrightarrow$ and $|$ are meta-symbols of BNF; the remaining symbols are all terminals. $\ll$ and $\gg$ delimit comments.

L$_1$. *I* $\longrightarrow$ *dep* $|$ *dep* $\wedge$ *I* $\ll$conjunction: interleaving$\gg$

L$_2$. *dep* $\longrightarrow$ *seq* $|$ *seq* $\vee$ *dep* $\ll$disjunction: choice$\gg$

L$_3$. *seq* $\longrightarrow$ *bool* $|$ *event* $|$ *event* $\cdot$ *seq* $\ll$before: ordering$\gg$

L$_4$. *bool* $\longrightarrow$ 0 $|$ $\top$

$\Gamma \neq \emptyset$ is the set of event literals used in $\mathcal{I}$. Γ_D is the set of literals mentioned in D and their complements—e.g., $\Gamma_e = \{e, \overline{e}\}$. Our formal semantics is based on runs, i.e., sequences of events. *Legal* runs satisfy the following requirements: (1) event instances and their complements are mutually exclusive and (2) an event instance occurs at most once in a computation. Our universe is $\mathbf{U}_\mathcal{I}$, which contains all legal runs involving event instances from Γ. For $\tau \in \mathbf{U}_\mathcal{I}$ and $I \in \mathcal{I}$, $\tau \models I$ means that I is satisfied over the run τ. This notion can be formalized as follows. Here, τ_i refers to the ith item in τ and $\tau_{[i,j]}$ refers to the subrun of τ consisting of its elements from index i to index j, both inclusive. $|\tau|$ is the last index of τ and may be ω for an infinite run. We use the following conventions: e, f, $\overline{e}$, $\overline{f}$, etc. are literals; D, E, etc. are dependencies; i, j, k, etc. are temporal indices; and τ, etc. are runs.

M$_1$. $\tau \models e$ iff $(\exists i : \tau_i = e)$

M$_2$. $\tau \models I_1 \vee I_2$ iff $\tau \models I_1$ or $\tau \models I_2$

M$_3$. $\tau \models I_1 \wedge I_2$ iff $\tau \models I_1$ and $\tau \models I_2$

M$_4$. $\tau \models I_1 \cdot I_2$ iff $(\exists i : \tau_{[0,i]} \models I_1$ and $\tau_{[i+1,|\tau|]} \models I_2)$

The semantics of individual events has no temporal component. The idea is that the specifier does not care about when an event occurs except for the restrictions specified using the $\cdot$ operator. $I_1 \cdot I_2$ means that I_1 is satisfied before I_2 (thus both are satisfied). The denotation of a dependency I is the set of runs that satisfy I, i.e., $[\![I]\!] = \{\tau : \tau \models I\}$. We define equivalence of two dependencies as $D \equiv E$ iff $[\![D]\!] = [\![E]\!]$.

2.1 Coordination Relationships

As running examples, we use two dependencies due to Klein [9]. In Klein's notation, $e < f$ means that if both events e and f happen, then e precedes f; i.e., f disables e. Also in Klein's notation, $e \rightarrow f$ means that if e occurs then f also occurs (before or after e); i.e., e requires f. We formalize these below.

EXAMPLE 1. *Let $D_< \triangleq \overline{e} \vee \overline{f} \vee e \cdot f$. Let $\tau \in \mathbf{U}_\mathcal{I}$ 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; i.e., e must precede f on τ.*

EXAMPLE 2. *Let $D_\rightarrow \triangleq \overline{e} \vee f$. Let $\tau \in \mathbf{U}_\mathcal{I}$ 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.*

Let's see how our approach captures a variety of coordination requirements as dependencies.

D$_1$. e feeds or enables f. f requires e to occur before: $e \cdot f \vee \overline{f}$

D$_2$. e conditionally feeds f. If e occurs, it feeds f: $\overline{e} \vee e \cdot f \vee \overline{f}$

D$_3$. Guaranteeing e enables f. f can occur only if e has occurred or will occur: $e \vee \overline{e} \wedge \overline{f}$

D$_4$. e initiates f. f occurs iff e precedes it: $\overline{e} \wedge \overline{f} \vee e \cdot f$

D$_5$. e and f jointly require g. If e and f occur in any order, then g must also occur (in any order): $\overline{e} \vee \overline{f} \vee g$

D$_6$. g compensates for e failing f. If e happens and f does not, then perform $g : (\overline{e} \vee f \vee g) \wedge (\overline{g} \vee e) \wedge (\overline{g} \vee \overline{f})$

The above are mostly self-explanatory. D$_6$ captures requirements such as that if e occurs, but is not matched with f, then g must occur, and g must not occur otherwise. This is a typical requirement in information applications with data updates, where g corresponds to an action to restore the consistency of the information (potentially) violated by the success of e and the failure of f. Hence the need to *compensate* for e if f does not occur.

Consider the following simple scenario inspired by supply chains. Here an assembly service composes three services, supplying hoses, valves, and elbow joints. The assembly orders a matching hose and valve to create a requested assembly. For simplicity, each service (A, V, H, E) can be started and may complete successfully or fail. The elbow joints service supports cancellation (undo), which always succeeds. Thus, the events defined are A_s, A_c, V_s, V_c, H_s, H_c, E_s, E_c, and E_u (subscripts s, c, and u indicate start, successfully complete, and undo, respectively), and their complements. The failure of a service is the complement of its successful completion, e.g., the failure of the valves service is $\overline{V_c}$.

- If (and only if) an assembly is started, start the valve and hose services: $(\overline{A_s} \vee V_s \wedge H_s) \wedge (\overline{V_s} \vee A_s) \wedge (\overline{H_s} \vee A_s)$.
- As soon as the hose completes successfully, start elbow joints, except that if the valve service has failed before elbow joints are started, do not start elbow joints: $(\overline{H_c} \vee E_s \wedge V_c \vee \overline{V_c} \cdot \overline{E_s} \vee E_s \cdot \overline{V_c})$.
- If the valve service has failed but elbow joints have completed successfully, then and only then undo the elbow joints: $(V_c \vee \overline{E_c} \vee E_u) \wedge (\overline{E_u} \vee \overline{V_c}) \wedge (\overline{E_u} \vee E_c)$.

2.2 Residuation

Imagine a scheduler that schedules events to satisfy all stated dependencies. A dependency is satisfied when a run in its denotation is realized. We characterize the state of the scheduler by the runs it can allow—the state determines which events may or may not occur. Initially, the allowed runs are given by the stated dependencies. As events occur, the allowed runs get narrowed down.

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 runs allowed by the original dependencies, select those compatible with the actual history, and infer how to proceed. However, our approach achieves this effect symbolically, *without* examining the runs. This is important, because it makes our reasoning depend on the finite specifications, not on the potentially infinite runs.

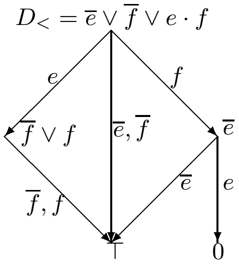

Figure 1: Scheduler states and transitions for $D_<$

The dependencies stated in a workflow fully describe the initial state of the scheduler; successive states are computed symbolically. Figure 1 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. An event that would make the scheduler obliged to 0 cannot occur.

For a source state D and transition by event e, the target state is given by D/e. This refers to the *residuation* of D by e and corresponds to the largest set of runs satisfying the given dependency:

M$_5$. $\nu \in [\![D/e]\!]$ iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_{\mathcal{I}} \Rightarrow \upsilon\nu \in [\![D]\!]))$

EXAMPLE 3. *(Figure 1) 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).*

2.3 Symbolic Calculation of Residuals

M$_5$ 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. Importantly, dependencies not mentioning an event have no direct effect on it and reasoning with respect to different dependencies can be performed modularly.

E$_1$. $0/e \doteq 0$

E$_2$. $\top/e \doteq \top$

E$_3$. $(E_1 \wedge E_2)/e \doteq ((E_1/e) \wedge (E_2/e))$

E$_4$. $(E_1 \vee E_2)/e \doteq (E_1/e \vee E_2/e)$

E$_5$. $(e \cdot E)/e \doteq E$, if $e \notin \Gamma_E$

E$_6$. $D/e \doteq D$, if $e \notin \Gamma_D$

E$_7$. $(e' \cdot E)/e \doteq 0$, if $e \in \Gamma_E$ (e' is any event literal)

E$_8$. $(\overline{e} \cdot E)/e \doteq 0$

EXAMPLE 4. *Verify that the above equations yield the transitions of Figure 1.*

The scheduler can take a decision to accept, reject, or trigger an event only if *no* dependency is violated by that decision. There are several ways to apply the above 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 run that satisfies all dependencies. The following theorem is proved in [14].

THEOREM 1. *Equations* E$_1$–E$_8$ *are sound and complete.* ■

3. DISTRIBUTED SCHEDULING

One of our requirements is that the agents act as autonomously, constrained only their coordination relationships. This presupposes that the decisions on events be taken based on local information. Further, distribution promises greater scalability and reliability by placing decision-making functionality right where the decision needs to be made.

To enable sound local decisions, we place a *guard* on each event. The guard on an event is a condition such that when it is true, it is OK to let the event happen. The guards depend on the dependencies that have been specified. We want the guards to be as general as possible. Moreover, as some events occur, other events can become enabled or disabled, i.e., the guards of the latter events can become true or false. This means that the guards of events can be modified based on messages from other events.

In other words, our approach requires (a) initially determining the guards on each event, (b) arranging for the relevant information to flow from one event to another, and (c) modifying the guards to assimilate the information received from other events.

3.1 Temporal Logic for Internal Reasoning

The guard on an event is the weakest condition whose truth guarantees correctness if the event occurs. Guards must be temporal expressions so that decisions taken on different events can be sensitive to the state of the system. The guards are compiled from the stated dependencies; in practice, they are quite succinct.

$\mathcal{T}$ is our temporal language for guards. $\Box E$ 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. $E \cdot F$ means that F has occurred preceded by E. For simplicity, we assume the following binding precedence (in decreasing order): $\neg$; $\cdot$; $\Box$ and $\Diamond$; $\wedge$; $\vee$. The syntax of $\mathcal{T}$ is given in BNF with T as the start symbol.

L$_5$. *T* $\longrightarrow$ *conj* | *conj* $\wedge$ *T*

L$_6$. *conj* $\longrightarrow$ *disj* | *disj* $\vee$ *conj*

L$_7$. *disj* $\longrightarrow$ *bool* | $\Box$ *seq* | $\Diamond$ *seq* | $\neg$ *event*

The semantics of $\mathcal{T}$ is given with respect to a run (as for $\mathcal{I}$) *and* an index into that run (unlike for $\mathcal{I}$). In addition, we need an auxiliary notion of semantics, which requires two indices. Our semantics characterizes progress along a given computation to determine the decision on each event. It has important differences from traditional linear temporal logics [7]. One, our runs are sequences of events, not of states. Two, our main semantic definitions are given in terms of a pair of indices, i.e., intervals, rather than a single index. For $0 \leq i \leq k$, $u \models_{i,k} E$ means that E is satisfied over the subsequence of u between i and k. For $k \geq 0$, $u \models_k E$ means that E is satisfied on u at index k—implicitly, i is set to 0. A run u is *maximal* iff for each event, either the event or its complement occurs on u. The universe, $\mathbf{U}_{\mathcal{T}}$, is the set of maximal runs.

M$_6$, which involves just one index i, invokes the semantics with the entire run until i. The second index is interpreted as the present moment. M$_{10}$ introduces a nonzero first index. M$_7$ and M$_{10}$ capture the dependence of an expression on the immediate past, bounded by the first index of the semantic definition. M$_8$, M$_9$, M$_{11}$, M$_{12}$, M$_{13}$, and M$_{14}$ are as in traditional semantics. M$_{13}$ and M$_{14}$ involve looking into the future. M$_7$ implicitly treats events as being in the scope of a past-time operator. Consequently, M$_{12}$ interprets $\neg$ as *not yet*.

Truth at an index corresponds to truth from the beginning to that index. An event is true in a range iff it occurs within that range. A sequence formula is true over a range iff its first component is true before the second component and both components are true within the same range.

M$_6$. $u \models_i E$ iff $u \models_{0,i} E$

M$_7$. $u \models_{i,k} f$ iff $(\exists j : i \leq j \leq k$ and $u_j = f)$, where $f \in \Gamma$

M$_8$. $u \models_{i,k} E \vee F$ iff $u \models_{i,k} E$ or $u \models_{i,k} F$

M$_9$. $u \models_{i,k} E \wedge F$ iff $u \models_{i,k} E$ and $u \models_{i,k} F$

M$_{10}$. $u \models_{i,k} E \cdot F$ iff $(\exists j : i \leq j \leq k$ and $u \models_{i,j} E$ and $u \models_{j+1,k} F)$

M$_{11}$. $u \models_{i,k} \top$

M$_{12}$. $u \models_{i,k} \neg E$ iff $u \not\models_{i,k} E$

M$_{13}$. $u \models_{i,k} \Box E$ iff $(\forall j : k \leq j \Rightarrow u \models_{i,j} E)$

M$_{14}$. $u \models_{i,k} \Diamond E$ iff $(\exists j : k \leq j$ and $u \models_{i,j} E)$

There are some important motivations for the above semantics. Our approach includes two kinds of negation: a strong kind applied on events as in $\overline{e}$, and a weak one as in $\neg e$. Logical frameworks with two negations have been studied before in the context of reasoning about action. However, the existing frameworks use the weak negation as a default, epistemic, or nonmonotonic negation, along the lines of negation as failure [10]. By contrast, we describe a purely monotonic system: although $\neg e$ indicates less information than $\overline{e}$, the decisions made during scheduling are cautious and do not have to be withdrawn.

Our approach takes events to be *stable*; i.e., once an event instance has occurred, it remains occurred forever. This idea is important for distributed systems to accommodate message delay [8]. However, it is in tension with the semantics of *before*. Specifically, if e and f hold forever, then a run that satisfies $e \cdot f$ would also satisfy $f \cdot e$, thereby losing ordering information. Our semantics avoids this undesirable effect.

We define $E \cong F$ iff E and F are true over the same index pairs over the same runs.

EXAMPLE 5. *The possible maximal runs for $\Gamma = \{e, \overline{e}\}$ are $\{\langle e \rangle, \langle \overline{e} \rangle\}$. On different runs, e or $\overline{e}$ may occur. Initially, neither e nor $\overline{e}$ has happened, so runs $\langle e \rangle$ and $\langle \overline{e} \rangle$ both satisfy $\neg e$ and $\neg \overline{e}$ at index 0. Run $\langle e \rangle$ satisfies $\Diamond e$ at 0, because event e will occur on it; similarly, run $\langle \overline{e} \rangle$ 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.*

3.2 Deriving Guards from Specifications

Our objective is to determine the guards purely symbolically. However, for expository ease, and to formally establish correctness of our approach, we begin with an obvious but inefficient approach and improve it step by step until we obtain the desired approach. We now use $\mathcal{T}$ to compile guards from dependencies.

Since the guards must yield precisely the computations that are allowed by the given dependencies, a natural intuition is that the guard of an event covers each computation in the denotation of the specified dependency, and no more.

We associate a set of *paths*, $\Pi(D)$, with a dependency D. A path $\rho \in \Pi(D)$ is a sequence of event symbols (no two of which are equal or complements) that residuate D to $\top$—the dependency is satisfied if the events in the path occur in the order of residuation. We require that $\Gamma_\rho \supseteq \Gamma_D$, i.e., all events in D (or their complements) feature in ρ. Each path is effectively a correct execution for its dependency. A path may have more events than those explicitly mentioned in a dependency. This is not a problem: Section 4.2 develops an equivalent approach that only looks at the dependency itself, not the paths. Clearly, the paths in $\Pi(D)$ satisfy D. Moreover, we can show that there is a unique dependency corresponding to any set of paths.

Since each path ρ in a dependency D satisfies D, if an event e occurs on ρ, it is clearly allowed by D, *provided* e occurs at the right time. In other words, e is allowed when (1) the events on ρ up to e have occurred in the right sequence, and (2) the events of ρ after e have not occurred, but will occur in the right sequence.

We define a series of operators to calculate guards as $\mathsf{G} : \mathcal{I} \times \Gamma \mapsto \mathcal{T}$. $\mathsf{G}_b(\rho, e)$ denotes the guard on e due to path ρ (b stands for *basic*). $\mathsf{G}_b(D, e)$ denotes the guard on e due to dependency D. To compute the guard on an event relative to a dependency D, we sum the contributions of different paths in D. $\mathsf{G}_b(\mathcal{W}, e)$ denotes the guard due to workflow $\mathcal{W}$ and is abbreviated as $\mathsf{G}_b(e)$ when $\mathcal{W}$ is known. This definition redundantly repeats information about the entire path on each event. Later, we shall remove this redundancy to obtain a semantically equivalent, but superior, solution.

DEFINITION 1. $\mathsf{G}_b(\rho, e) \triangleq$ *if* $e = e_i$, *then* $\Box(e_1 \cdot e_2) \wedge \ldots \wedge \Box(e_{i-2} \cdot e_{i-1}) \wedge \neg e_{i+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond(e_{i+1} \cdot e_{i+2}) \wedge \ldots \wedge \Diamond(e_{n-1} \cdot e_n)$, *else* 0.
$\mathsf{G}_b(D, e) \triangleq \bigvee_{\rho \in \Pi(D)} \mathsf{G}_b(\rho, e)$.
$\mathsf{G}_b(\mathcal{W}, e) \triangleq \bigwedge_{D \in \mathcal{W}} \mathsf{G}_b(D, e)$.

OBSERVATION 2.
$u \models_k \mathsf{G}_b(D, e) \Rightarrow (\exists \rho \in \Pi(D) : u \models_k \mathsf{G}_b(\rho, e))$. ■

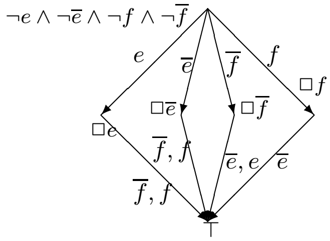

Figure 2: Guards with respect to $D_< = \overline{e} \vee \overline{f} \vee e \cdot f$

Figure 2 illustrates our procedure for the dependency of Example 1. The figure implicitly encodes all paths in $\Pi(D_<)$ (here, for simplicity, $\Gamma = \Gamma_D$). The initial node is labeled $\neg e \wedge \neg \overline{e} \wedge \neg f \wedge \neg \overline{f}$ 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 avoid clutter, labels like $\Diamond e$ and $\neg e$ are not shown after the initial state.

EXAMPLE 6. *Using Figure 2, we can compute the guards for the events in $D_<$. Each path on which e occurs contributes the conjunction of a $\Box$ term (what happens before e) and a $\neg$ and a $\Diamond$ term (what happens after e).*

- $\mathsf{G}_b(D_<, e) = (\neg f \wedge \neg \overline{f} \wedge \Diamond(\overline{f} \vee f)) \vee (\Box \overline{f} \wedge \top)$. *But* $\Diamond(\overline{f} \vee f) \cong \top$. *Hence,* $\mathsf{G}_b(D_<, e) = (\neg f \wedge \neg \overline{f}) \vee \Box \overline{f}$, *which reduces to* $\neg f \vee \Box \overline{f}$, *which equals* $\neg f$.
- $\mathsf{G}_b(D_<, \overline{e}) = (\neg f \wedge \neg \overline{f} \wedge \Diamond(\overline{f} \vee f)) \vee (\Box f \wedge \top) \vee (\Box \overline{f} \wedge \top)$, *which reduces to* $\top$.
- $\mathsf{G}_b(D_<, \overline{f}) = \top$.
- $\mathsf{G}_b(D_<, f) = (\neg e \wedge \neg \overline{e} \wedge \Diamond \overline{e}) \vee \Box e \vee \Box \overline{e} \cong \Diamond \overline{e} \vee \Box e$.

Thus $\overline{e}$ can occur at any time, e can occur if f has not yet happened (possibly because f will never happen), $\overline{f}$ can occur any time, but f can occur only if e has occurred or $\overline{e}$ is guaranteed.

3.3 Scheduling with Guards

To execute an event e, check if its guard is $\top$ (execute e), 0 (reject e), or neither (make e wait). Whenever an event e occurs, notify all events depending on e that $\Box e$ now holds, thereby causing their guards to be updated.

EXAMPLE 7. *Using the guards from Example 6, if e is attempted and f has not already happened, e's guard evaluates to $\top$. Consequently, e is allowed and a notification $\Box e$ is sent to f (and $\overline{f}$). Upon receipt of this notification, f's guard is simplified from $\Diamond\overline{e} \vee \Box e$ to $\top$. Now if f is attempted, it can happen immediately.*

If f is attempted first, it must wait because its guard is $\Diamond\overline{e} \vee \Box e$ and not $\top$. Sometime later if $\overline{e}$ or e occurs, a notification of $\Box\overline{e}$ or $\Box e$ is received at f, which simplifies its guard to $\top$, thus enabling f. The guards of $\overline{e}$ and $\overline{f}$ equal $\top$, so they can happen at any time.

The above development shows how we can compute the semantics of $\mathcal{T}$, i.e., realize the appropriate runs, incrementally. But in some situations potential race conditions and deadlocks can arise. To ensure that the necessary information flows to an event when needed, the execution mechanism should be more astute in terms of recognizing and resolving mutual constraints among events. This reasoning is essentially encoded in terms of heuristic graph-based reasoning. Although these heuristics can handle many interesting cases, they are not claimed to be complete.

4. FORMALIZATION

We have two main motivations for carrying out a formalization of our approach. Formalization can help in proving the correctness of an approach and in justifying improvements in efficiency, e.g., in updating guards incrementally as messages are exchanged and to simplify guards prior to execution.

Correctness is a concern when (a) guards are compiled, (b) guards are preprocessed, and (c) events are executed and guards updated. Correctness depends on how the guards are used to yield actual computations. That is, correctness depends on the *evaluation strategy*, which determines how events are scheduled. We formalize evaluation strategies by stating what initial values of guards they use, and how they update the guards. We begin with a strategy that is simple but correct and produce a series of more sophisticated, but semantically equivalent (hence correct), strategies.

DEFINITION 2. *An evaluation strategy is a function* $\mathsf{S} : \wp(\mathcal{I}) \mapsto (\Gamma \times \mathbf{U}_{\mathcal{T}} \times \mathbf{N} \mapsto \mathcal{T})$.

Given workflow $\mathcal{W}$, S yields a function $\mathsf{S}(\mathcal{W})$, which captures the evolution of guards and execution of events. Given an event e, a run v, and index j in v, $\mathsf{S}(\mathcal{W}, e, v, j)$ equals the current guard of e at index j of v. Here v corresponds to the run being "generated" and j indicates how far the computation has progressed. Formally, an evaluation strategy $\mathsf{S}(\mathcal{W})$ *generates* run $u \in \mathbf{U}_{\mathcal{T}}$ if for each event e that occurs on u, u satisfies e's current guard due to $\mathsf{S}(\mathcal{W})$ at the index preceding e's occurrence. We write this as $\mathsf{S}(\mathcal{W}) \leadsto u$.

DEFINITION 3. $\mathsf{S}(\mathcal{W}) \leadsto_i u \triangleq (\forall j : 1 \leq j \leq i \Rightarrow u \models_{j-1} \mathsf{S}(\mathcal{W}, u_j))$.

DEFINITION 4. $\mathsf{S}(\mathcal{W}) \leadsto u \triangleq (\forall i : i \leq |u| \Rightarrow \mathsf{S}(\mathcal{W}) \leadsto_i u)$.

The idea is that an evaluation strategy incrementally generates the given run. At any index in the run, an event may take place if its guard is true at the preceding index in the run. That is, an event may be allowed only at a specified index in the run. A given partial run may be completed in various ways, all of which would respect the stated dependencies.

Although the guard is verified at the designated index on the run, its verification might involve future indices on that run. That is, the guard may involve $\Diamond$ expressions that happen to be true on the given run at the index of e's occurrence. Because generation looks ahead into the future, it is more abstract than execution.

In order to establish model-theoretic correctness of the initial compilation procedure given by Definition 1, we begin with a trivial strategy, S_b. S_b sets the guards using G_b and *never* modifies them. Theorem 4 establishes the soundness and completeness of guard compilation.

DEFINITION 5. $(\forall v, j : \mathsf{S}_b(\mathcal{W}, e, v, j) \triangleq \mathsf{G}_b(\mathcal{W}, e))$.

OBSERVATION 3. $\mathsf{S}_b(\mathcal{W}) \leadsto u$ *iff* $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(\mathcal{W}, u_j))$. ∎

For runs u and v, $u \sim_i v$ means that u agrees with v up to index i. $u \sim_i D$ means that u agrees with dependency D up to index i. Now we can show that if a run agrees all the way with D, then it satisfies D. Further, if $u \models_{k-1} \mathsf{G}_b(w, u_k)$, then u contains the events of w in the same order.

THEOREM 4. $\mathsf{S}_b(\mathcal{W}) \leadsto u$ *iff* $(\forall D \in \mathcal{W} : u \models D)$.

PROOF. *Consider any dependency D in $\mathcal{W}$. Since D is obeyed by the strategy, at least one path must allow the given run; this means the run contains the events of this path; hence, the run satisfies the dependency. Conversely, treating a given run as a path in each dependency ensures that the run can be generated by the strategy.* ∎

4.1 Evaluating Guards

At run time, a guard equal to $\top$ means the given event can occur; a guard equal to 0 means the given event cannot occur. For all guards in between, we would potentially have to modify them in light of messages about other events. The operator $\div$ captures the processing required to assimilate different messages into a guard. This operator embodies a set of "proof rules" to reduce guards when events occur or are promised. Table 1 defines these rules. Because our sequences are limited to two literals, we do not consider longer sequences. Our improved guard definition only creates two-sequence guards expressions, because it works straight from the dependency syntax.

Old Guard G	**Message** M	**New Guard** $G \div M$
$G_1 \vee G_2$	M	$G_1 \div M \vee G_2 \div M$
$G_1 \wedge G_2$	M	$G_1 \div M \wedge G_2 \div M$
$\Box e$	$\Box e$	$\top$
$\Box\overline{e}$	$\Box e$ or $\Diamond e$	0
$\Diamond e$	$\Box e$ or $\Diamond e$	$\top$
$\Diamond\overline{e}$	$\Box e$ or $\Diamond e$	0
$\Box(e_1 \cdot e_2)$	$\Box(e_1 \cdot e_2)$	$\top$
$\Box(e_1 \cdot e_2)$	$\Box(e_2 \cdot e_1)$	0
$\Box(e_1 \cdot e_2)$	$\Box\overline{e_i}$ or $\Diamond\overline{e_i}$	0
$\Diamond(e_1 \cdot e_2)$	$\Box(e_1 \cdot e_2)$ or $\Diamond(e_1 \cdot e_2)$	$\top$
$\Diamond(e_1 \cdot e_2)$	$\Box(e_2 \cdot e_1)$ or $\Diamond(e_2 \cdot e_1)$	0
$\Diamond(e_1 \cdot e_2)$	$\Box\overline{e_i}$ or $\Diamond\overline{e_i}$	0
$\neg e$	$\Box e$	0
$\neg\overline{e}$	$\Box e$ or $\Diamond e$	$\top$
G	M	G, otherwise

Table 1: Assimilating Messages

When the dependencies involve sequence expressions, the guards can end up with sequence expressions, which indicate ordering of the relevant events. In such cases, the information that is assimilated into a guard must be consistent with that order. For this reason, the updates in those cases are more complex. Lemma 5 means that the operator $\div$ preserves the truth of the original guards.

LEMMA 5. $(\exists k \leq j : u \models_k M$ *and* $u \models_j G \div M) \Rightarrow u \models_j G$.

PROOF. *The proof is by induction on the structure of expressions. The base cases can be verified by inspection. Let* $(\exists k \leq j : u \models_k M$ *and* $u \models_j (G_1 \div M \vee G_2 \div M))$. *If* $u \models_j G_1 \div M$, *then* $u \models_j G_1$ *(inductive hypothesis). Therefore,* $u \models_j G_1 \vee G_2$, *and similarly for* G_2. $G_1 \wedge G_2$ *is analogous.* ■

The repeated application of $\div$ to update the guards corresponds to a new evaluation strategy, $\mathsf{S}_\div$. This strategy permits possible runs on which the guards are initially set according to the original definition, but *may* be updated in response to expressions verified at previous indices. $\mathsf{S}_\div$ does not require that every M that is true be used in reducing the guard. Lemma 5 enables us to accommodate message delay, because notifications need not be incorporated immediately. This is because when $\Box e$ and $\Diamond e$ hold at an index, they hold on all future indices.

DEFINITION 6. $\mathsf{S}_\div(\mathcal{W}, e, u, 0) \ \underline{\triangle}\ \mathsf{G}_b(\mathcal{W}, e)$.
$\mathsf{S}_\div(\mathcal{W}, e, u, i+1) \neq \mathsf{S}_\div(\mathcal{W}, e, u, i) \Rightarrow (\exists k \leq i : u \models_k M$ *and* $\mathsf{S}_\div(\mathcal{W}, e, u, i+1) = (\mathsf{S}_\div(\mathcal{W}, e, u, i) \div M))$.

Theorem 6 establishes that the evaluation of the guards according to $\div$ is sound and complete. All runs that could be generated by the original guards are generated when the guards are updated (completeness) and that any runs generated through the modified guards could be generated from the original guards (soundness).

THEOREM 6. *Replacing* S_b *by* $\mathsf{S}_\div$ *preserves correctness, i.e.,* $\mathsf{S}_\div(\mathcal{W}) \leadsto u$ *iff* $\mathsf{S}_b(\mathcal{W}) \leadsto u$.

PROOF. *From Definition 6, it is possible to have a run u, such that* $(\forall i : \mathsf{S}_\div(\mathcal{W}, e, u, i) = \mathsf{G}_b(\mathcal{W}, e))$. *Therefore,* $\mathsf{S}_\div(\mathcal{W})$ *generates all the runs that* $\mathsf{S}_b(\mathcal{W})$ *generates. Thus completeness is established. Soundness is established inductively since an acceptable run is computed incrementally.* ■

The main motivation for performing guard evaluation as above is that it enables us to incrementally collect the information necessary to make a local decision on each event. Theorem 6 establishes that any such modified execution is correct. However, executability requires in addition that we can take decisions without having to look into the future. The above theorem does not establish that the guards for each event will be reduced to $\Box$ and $\neg$ expressions (which require no information about the future). That depends on how effectively the guards are processed.

4.2 Simplification

We now show how guards as given in Definition 1 can be computed more efficiently. Theorems 7 and 8 show that the computations during guard compilation can be distributed over the conjuncts and disjuncts of a dependency. Since our dependencies are limited to having sequences of two literals, this means the constituent sequence terms can be processed independently of each other. Theorem 8 is also important for another reason, namely, that in essence it equates a workflow with the conjunction of the dependencies in it.

The next theorems rely on additional auxiliary definitions and results. $I(w, \Gamma')$ gives all the superpaths of w that include all interleavings of w with the events in Γ'. We assume that $(\forall e : e \in \Gamma'$ iff $\overline{e} \in \Gamma')$. First we show that the guard contributed by a path w equals the sum of the contributions of the paths that extend w, provided all possible extensions relative to some Γ' are considered. For each event e in Γ', e and $\overline{e}$ can occur anywhere relative to w, and thus they essentially factor out. Using this, we show that the guards are well-behaved with respect to the denotations of the dependencies, i.e., $D \equiv E \Rightarrow \mathsf{G}_b(D, e) = \mathsf{G}_b(E, e)$.

THEOREM 7. $\mathsf{G}_b(D \vee E, e) = \mathsf{G}_b(D, e) \vee \mathsf{G}_b(E, e)$.

PROOF. *The implication from left to right is trivial. In the opposite direction, starting with a path* w *in* $\Pi(D)$, *we can interleave* w *safely with events in* $\Gamma_{D \vee E} \setminus \Gamma_w$. *Likewise for* $\Pi(E)$. *Thus the contribution of* w *to the right hand side is included in the left hand side.* ■

THEOREM 8. $\mathsf{G}_b(D \wedge E, e) = \mathsf{G}_b(D, e) \wedge \mathsf{G}_b(E, e)$.

PROOF. *The implication from left to right is trivial. In the opposite direction, consider the contribution of a pair of paths* $v \in \Pi(D)$ *and* $w \in \Pi(E)$ *to* $\mathsf{G}_b(D, e) \wedge \mathsf{G}_b(E, e)$. *If* e *does not occur on both* v *and* w, *the contribution is* 0. *Otherwise, construct a path in* $\Pi(D \wedge E)$, *which has the combined contribution of* v *and* w. ■

We introduce the *basis* of a set of paths as a means to show how guards can be compiled purely symbolically, and to establish some critical results regarding the independence of events from dependencies that do not mention them. Intuitively, Ψ, the basis of $\Pi(D)$, is the subset of $\Pi(D)$ that involves only those events that are mentioned in D. $\Pi(D)$ can be determined from $\Psi(D)$ and vice versa. Theorem 9 shows that the guard on an event e due to a dependency D not mentioning e is simply $\Diamond D$. This means that, for most dependencies and events, guards can be quickly compiled into a succinct expression.

THEOREM 9. $\mathsf{G}_b(D, e) = \Diamond D$, *if* $e \notin \Gamma_D$.

PROOF. *Apply induction on the structure of dependencies. For the base case, consider* 0, $\top$, *and event* f, *where* $f \neq e$. *For the inductive step, let* $D = e_1 \cdot \ldots \cdot e_n$. *Separately establish that* $\mathsf{G}_b(D, e) = \Diamond D$. *Since* $D \in \mathcal{I}$, *we can show* $\Diamond(D_1 \vee D_2) \cong \Diamond D_1 \vee \Diamond D_2$, *and* $\Diamond(D_1 \wedge D_2) \cong \Diamond D_1 \wedge \Diamond D_2$. ■

4.2.1 Symbolically Calculating Guards

We now define a symbolic calculation for guards as below. These cases cover all of the syntactic possibilities of $\mathcal{I}$. Importantly, our definition distributes over $\wedge$ and $\vee$: using our normalization requirement, each sequence subexpression can be treated separately. Thus the guards are quite succinct for the common cases, such as the relationships of Section 2.1.

DEFINITION 7. *The guards are given by the operator* $\mathsf{G} : \mathcal{I} \times \Xi \mapsto \mathcal{T}$:

(a) $\mathsf{G}(D_1 \vee D_2, e) \ \underline{\triangle}\ \mathsf{G}(D_1, e) \vee \mathsf{G}(D_2, e)$
(b) $\mathsf{G}(D_1 \wedge D_2, e) \ \underline{\triangle}\ \mathsf{G}(D_1, e) \wedge \mathsf{G}(D_2, e)$
(c) $\mathsf{G}(e_1 \cdot \ldots \cdot e_i \cdot \ldots \cdot e_n, e_i) \ \underline{\triangle}\ \Box e_1 \wedge \ldots \wedge \Box e_{i-1} \wedge \neg e_{i+1} \wedge \ldots \neg e_n \wedge \Diamond(e_{i+1} \cdot e_{i+2}) \wedge \ldots \wedge \Diamond(e_{n-1} \cdot e_n)$
(d) $\mathsf{G}(e_1 \cdot \ldots \cdot e_n, e) \ \underline{\triangle}\ \Diamond(e_1 \cdot \ldots \cdot e_n)$, *if* $\{e, \overline{e}\} \not\subseteq \{e_1, \overline{e_1}, \ldots, e_n, \overline{e_n}\}$
(e) $\mathsf{G}(e_1 \cdot \ldots \cdot e_i \cdot \ldots \cdot e_n, \overline{e_i}) \ \underline{\triangle}\ 0$
(f) $\mathsf{G}(0, e) \ \underline{\triangle}\ 0$
(g) $\mathsf{G}(\top, e) \ \underline{\triangle}\ \top$

EXAMPLE 8. *We compute the guards for the events in* $D_<$:

- $\mathsf{G}(D_<, e) = (\Diamond \overline{f} \vee (\neg f \wedge \Diamond f)) \cong \neg f$
- $\mathsf{G}(D_<, \overline{e}) = \top$
- $\mathsf{G}(D_<, f) = \Diamond \overline{e} \vee \Box e$
- $\mathsf{G}(D_<, \overline{f}) = \top$

Thus $\overline{f}$ and $\overline{e}$ can occur at any time. However, f can occur only if e has occurred or never will. Similarly, e can occur only if f has not yet occurred (it may or may not occur in the future).

The symbolic calculation of guards corresponds to a new evaluation strategy, S_s, which begins with guards initialized according to Definition 7 and updates guards through $\div$.

DEFINITION 8. $\mathsf{S}_s(\mathcal{W}, e, u, 0) \triangleq \mathsf{G}(\mathcal{W}, e)$.
$\mathsf{S}_s(\mathcal{W}, e, u, i+1) \neq \mathsf{S}_s(\mathcal{W}, e, u, i) \Rightarrow (\exists k \leq i : u \models_k M$ *and* $\mathsf{S}_s(\mathcal{W}, e, u, i+1) = (\mathsf{S}_s(\mathcal{W}, e, u, i) \div M))$.

Theorem 10 establishes that the evaluation of the guards according to $\div$ is sound and complete. All runs that could be generated by the original guards are generated when the guards are updated (completeness) and that any runs generated through the modified guards could be generated from the original guards (soundness).

THEOREM 10. *Replacing* $\mathsf{S}_\div$ *by* S_s *preserves correctness, i.e.,* $\mathsf{S}_s(\mathcal{W}) \rightsquigarrow u$ *iff* $\mathsf{S}_\div(\mathcal{W}) \rightsquigarrow u$.

PROOF. *Follows from Theorems 7, 8, 9, which are in essence encoded in Definition 7.* ■

4.2.2 Eliminating Irrelevant Guards

Theorem 11 shows that the guard on an event e due to a dependency D in which e does not occur can be set to $\top$, provided D is entailed by the given workflow—an easy test of entailment is that D is in the workflow. Thus dependencies in the workflow that do not mention an event can be safely ignored for that event. This makes sense because the events mentioned in D will ensure that D is satisfied in any generated run. Thus at all indices of any generated run, we will have $\Diamond D$ anyway. Below, $\mathsf{G}^{\Delta}_{\top}$ replaces the irrelevant guards for events not in Δ; $\mathsf{S}^{\Delta}_{\top}$ is the corresponding strategy.

THEOREM 11. *Replacing* $\mathsf{S}_s(\mathcal{W})$ *by* $\mathsf{S}^{\Delta}_{\top}(\mathcal{W})$ *does not violate correctness.*

PROOF. *Since* $\mathsf{S}^{\Delta}_{\top}(\mathcal{W})$ *is weaker than* $\mathsf{S}_s(\mathcal{W})$*, completeness is preserved.*
Consider $D \in \mathcal{W}$ *and* $e \notin \Gamma_D$*. Let* $f \in \Gamma_D$*. Consequently,* e *and* $\overline{e}$ *do not occur in* $\mathsf{G}(D, f)$*. Thus the occurrence or nonoccurrence of* e *or* $\overline{e}$ *has no effect upon* f*.*
Let $\mathsf{S}^{\Delta}_{\top}(\mathcal{W}) \rightsquigarrow u$*. If* $u \not\models_j \mathsf{S}_s(D, u_{j+1})$ *and* $u \models_j \mathsf{S}^{\Delta}_{\top}(D, u_{j+1})$*, then clearly* $u_{j+1} \notin \Gamma_D$*. Let* $B(u) = \{u_i : u \not\models_{i-1} \mathsf{S}_s(D, u_i)\}$*.*
Let v *be such that* $u \sqsupseteq v$ *and* $\Gamma_v = \Gamma_u \setminus B(u)$*. Since the guards for events in* Γ_D *do not depend on* u_{j+1}*, we have that*
$(\forall k, l : 1 \leq k$ *and* $1 \leq l$ *and* $u_k = v_l \Rightarrow u \models_{k-1} \mathsf{G}(D, u_k)$ *iff*
$v \models_{l-1} \mathsf{G}(D, v_l))$*. Hence,* $\mathsf{S}_s(\mathcal{W}) \rightsquigarrow v$*. By Theorem 4,* $v \models D$
and thus $u \models D$*.* ■

Theorems 8 and 11 establish that the guard on an event e due to a conjunction of dependencies is the conjunction of the guards due to the individual dependencies that mention e. Thus, we can compile the guards modularly and obtain expressions that are more amenable to processing.

4.3 Formalizing Event Classes

Not all events are alike. We consider four classes of events, which have different properties with respect to coordination. Because the significant events are visible, the service always knows of their occurrence (some of our results are about distributing this knowledge requirement over the agents). When an agent is willing to delay or omit an event, it is on the basis of constraints provided by the service, which in turn involve the occurrence or nonoccurrence of other events.

- *Flexible*, which the agent is willing to delay or omit
- *Inevitable*, which the agent is willing only to delay
- *Immediate*, which the agent performs unilaterally, i.e., is willing neither to delay nor to omit
- *Triggerable*, which the agent is willing to perform if requested.

The first three classes are mutually exclusive and exhaustive; each can be conjoined with triggerability. We do not have a category where an agent will entertain omitting an event, but not delaying it, because unless the agent performs the event unilaterally, there must be some delay in receiving a response from the service. This is because the above event classes apply to the interface between an agent and a logical coordinator.

Event classes do not replace the dependencies, which specify the constraints among different agents. For example, it is possible that a postal service agent may offer to deliver at two addresses in a particular order, but let the residents at the two addresses decide whether delivery should be made there at all. Thus at the conceptual level, it is possible that an agent may appear to another agent to be willing to cancel an action but not to delay it. However, this requirement is captured through dependencies. If p_1 and p_2 are the postal deliveries and a_1 and a_2 are the addressees' permissions then, following Examples 1 and 2, we would have $\overline{p_1} \vee \overline{p_2} \vee p_1 \cdot p_2$ (if both deliveries occur, they are ordered), and $\overline{p_1} \vee a_1$ and $\overline{p_2} \vee a_2$ (deliveries are only made if permitted).

For inevitable and immediate events, the dependencies must be strengthened. The basic idea is to eliminate paths whose prefixes lead to a state where an inevitable event may have to be denied, or an immediate event denied or delayed. An algorithm to derive strengthened dependencies proceeds by iteratively removing unsafe paths; it is iterative because removing one path to make a dependency safe for one event can make it unsafe for another.

The strengthened dependencies are then used in all reasoning, e.g., in computing the guards. Below, we show revised representations for some relationships.

EXAMPLE 9. *We can verify that if* e *is inevitable, then* $\overline{e} \vee f$ *strengthens to* $e \cdot f \vee \overline{e} \cdot f \vee \overline{e} \cdot \overline{f} \vee f \cdot e \vee f \cdot \overline{e}$*, which simplifies to* $\overline{e} \cdot \overline{f} \vee f$*, i.e., slightly stronger than the original dependency. Similarly, referring to Figure 4, we can check that if* e *is immediate, then* $\overline{e} \vee \overline{f} \cdot e$ *strengthens to* 0*.*

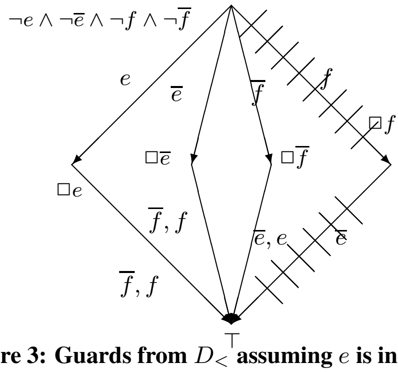

Figure 3: Guards from $D_<$ assuming e is inevitable

$\overline{e} \vee \overline{f} \vee e \cdot f$

EXAMPLE 10. *Figure 3 shows the dependency of Figure 2. The path* $\langle f\overline{e}\rangle$ *is deleted because if* f *occurs first,* e *must not occur. We can verify that* $\mathsf{G}_b(D_<, e)$ *is unchanged but* $\mathsf{G}_b(D_<, f)$ *is stronger: since* e *cannot be rejected, we cannot let* f *happen unless* e *or* $\overline{e}$ *has already happened. Figure 3 still holds when* e *is immediate. Thus the same guards are obtained as before.*

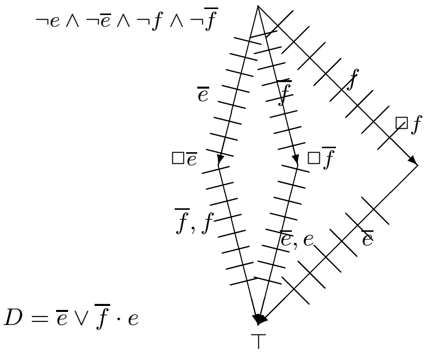

Figure 4: Extreme example of an immediate event (e)

EXAMPLE 11. *Referring to Figure 4, we can readily see that the guards for all the events are* 0*. However, if* e *is inevitable, then the guards are nonzero, as can be readily checked.*

5. DISCUSSION

We presented a generic approach for building multiagent systems to achieve service composition. Our approach hones in on the structure of the composed computations. It can thus facilitate the design and enactment of coordinated behavior by hiding low-level details. By separating the specification and the internal languages, we can begin with declarative specifications and derive operational decision procedures from them. The semantics of $\mathcal{I}$ considers entire runs, to reflect the intuition that the specifier only cares about good versus bad runs, not how to achieve them. By contrast, the semantics of $\mathcal{T}$ involves indices into the runs, to reflect the intuition that decisions on events are sensitive to what exactly has transpired when a decision is taken. Our approach requires neither unnecessary control over agents' actions, nor detailed knowledge of the agents' construction.

Some relevant ideas exist in the literature on extended transactions. We consider two of the recent contributions geared for Web service composition. One is the business transaction protocol, which provides primitives through which a composition can be carried out by a central coordinator [5]. The composed services are all modeled by a fixed set of significant events, along the lines of the two-phase commit protocol: start, prepare to commit, fail, or commit. The protocol describes how the coordinator communicates with the services. By contrast, our approach enables the significant events to be selected and dependencies among them to be stated in an application-specific manner, and the execution is distributed among the services.

The other approach is TxA (née WSTx), which defines three provider transactional *attitudes*. The attitudes specify how a given service may participate in a transaction [12]: by exposing a prepared to commit state for one or a group of operations or by allowing compensation. The transactions are coordinated through an external coordinator, which tracks the success or failure of the individual services, and reports them to the application.

Concurrent METATEM, a executable temporal logic, has been used to specify and build agents and multiagent systems [15]. It enables the specification of the behavior of the various agents, somewhat like reactive systems in traditional logics of programs. This is a major difference from our approach, because we only formalize the coordination requirements in our logic, and leave the internal details to the implementors. However, both approaches use linear models with a bounded past and infinite future. Our $\cdot$ (before) operator is related to the *until* and *since* operators. Concurrent METATEM has a greater range of operators. Wooldridge assumes that the agents execute in lock-step synchronization, and that they always choose the "right" path, which would lead to the stated rules being satisfied (assuming the rules are consistent) [15]. These assumptions are relaxed in our approach. Our agents can execute asynchronously, and must be serialized only when necessary.

The present paper does not address the challenge of how the required declarative specifications are created in the first place. To this end, we previously proposed a methodology based on an analysis of example conversations [13]. This methodology produces specifications that capture the essential constraints and no more.

We lack the space to include complexity results here. In general, the approach is efficient because it is limited; yet it is expressive enough for practical service composition needs. Proofs of theorems and additional motivation and results for the technical development of Sections 3 and 4 are available in [14].

6. REFERENCES

[1] A. Ankolekar, et al. DAML-S: Semantic markup for Web services. *Proc. Intl. Semantic Web Working Symp.*, Jul. 2001.

[2] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. *Proc. ACM Symp. Theory of Computing*, pp 51–63. 1984.

[3] Business process execution language for web services, version 1.0, July 2002. www-106.ibm.com/developerworks/webservices/library/ws-bpel.

[4] F. Curbera, et al. Unraveling the Web services Web. *IEEE Internet Computing*, 6:86–93, 2002.

[5] S. Dalal, S. Temel, M. Little, M. Potts, and J. Webber. Coordinating business transactions on the Web, *IEEE Internet Computing*, 7(1):30–39, Jan. 2003.

[6] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[7] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, vol. B, pages 995–1072. North-Holland, 1990.

[8] N. Francez and I. R. Forman. *Interacting Processes*. ACM Press, New York, 1996.

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

[10] R. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[11] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic Web services. *IEEE Intelligent Systems*, 16(2):46–53, Mar. 2001.

[12] T. Mikalsen, S. Tai, and I. Rouvellou. Transactional Attitudes. *Proc. Workshop Dependable Middleware-Based Systems* at Dependable Systems & Networks Conf., 2002.

[13] M. P. Singh. Synthesizing coordination requirements for heterogeneous autonomous agents. *Autonomous Agents and Multi-Agent Systems*, 3(2):107–132, Jun. 2000.

[14] M. P. Singh. Distributed enactment of multiagent flows, TR 2003-12, Dept of Computer Science, NCSU, 2003.

[15] M. Wooldridge. A knowledge-theoretic semantics for concurrent METATEM. *Agent Theories, Architectures, and Languages*, LNAI 1193, pp. 357–374, Springer, 1997.

[16] J. Xing and M. P. Singh. Engineering Commitment-Based Multiagent Systems: A Temporal Logic Approach, *Proc. Intl. Joint Conf. Autonomous Agents and MultiAgent Systems*, ACM Press, 2003, this volume.

[17] P. Yolum and M. P. Singh. Flexible Protocol Specification and Execution, *Proc. Intl. Joint Conf. Autonomous Agents and MultiAgent Systems*, pp. 527–534, ACM Press, 2002.