Distributed Scheduling of Workflow Computations[*]

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

singh@ncsu.edu

September 12, 1996

Abstract

Workflows are composite activities that achieve interoperation of a variety of system and human tasks. Workflows must satisfy subtle domain-specific semantic and organizational requirements. Consequently, flexibility in execution is crucial. We address the problem of scheduling workflows from declarative specifications in terms of intertask dependencies and event attributes. Our approach goes beyond previous approaches in being generic and distributed, and having a rigorous formal semantics. It involves distributed events, which are automatically set up to exchange the necessary messages. Our approach uses symbolic reasoning to (a) determine the initial constraints on events or *guards*, (b) preprocess the guards, and (c) execute the events. This approach has been implemented in an actor language.

[*]This is a greatly extended and revised version of a paper that appears in the Proceedings of the International Conference on Data Engineering (ICDE), held in New Orleans in February 1996.

[†]Partially supported by the NCSU College of Engineering and by the National Science Foundation under grant IRI-9529179 and IRI-9624425.

Contents

1 Introduction

Heterogeneous systems consist of a number of differently constructed applications and information resources that must interoperate meaningfully. *Workflows* are composite activities that achieve interoperation of a variety of system and human tasks [Dayal *et al.*, 1993; Georgakopoulos *et al.*, 1995]. Because of their importance to heterogeneous environments—both in legacy and in modern, open settings—workflows are garnering much attention in the research community. Simultaneously, a number of workflow products have merged—recent estimates of their number go up to 250. Current products are limited in several respects: they (a) are centralized, (b) provide limited database support, (c) are often restricted to forwarding forms via email, (d) require procedural encoding of workflows, and (e) assume homogeneous PC platforms.

To remove these limitations requires extensions along a number of dimensions. We envision workflows as incorporating both semantic and organizational requirements. The former category includes abstractions from extended transaction models that relate to correctness, concurrency control, failure handling, and recovery. The latter category includes abstractions from business modeling that relate to organizational structure, roles of participants, and coordination among them and the information system. These topics are being actively researched, e.g., [Bußler & Jablonski, 1994; Ramamritham & Chrysanthis, 1996], but they are not the subject of this paper. However, it is significant that these efforts would be greatly facilitated by an infrastructure that includes a generic and rigorous approach to workflow specification and scheduling. We provide such an infrastructure. Although our approach applies to activities whose component tasks are not restricted in any way, our primary interest is in workflows with a databases and information systems focus. Some of the constituent tasks may be database transactions; some may involve human interaction; some may be other kinds of computations.

For motivation, consider the transaction concept in traditional databases. Transactions provide a high level of support for traditional single-server applications. By following the transaction discipline, application programmers can isolate their programs from any undetermined set of other programs that might happen to be executing concurrently. The burden on the application programmer is to make simple system calls to initiate and commit or abort transactions; the database system does the rest. The transaction concept is thus immensely successful. Unfortunately, the transaction concept applies primarily to centralized homogeneous database systems. With some effort, it can be extended to certain distributed environments, but proves quite limiting. This is because the very features of the transaction concept, the so-called ACID properties [Gray & Reuter, 1993], become a hindrance in heterogeneous settings. The ACID properties are:

- *Atomicity:* All of a transaction happens or none.
- *Consistency:* Any concurrent execution of the transactions preserves consistency, provided each transaction would if executed exclusively on the system.
- *Isolation:* The intermediate results of a transaction are not visible to any other transaction.
- *Durability:* If a transaction commits, its results are permanent.

These properties can be ensured through serializable executions of transactions [Bernstein *et al.*, 1987]. However, although serializable executions are acceptable in a single-site system, they can be unacceptable in a distributed heterogeneous system. This is because serializability requires that the subtransactions at each site have the same serialization order, which effectively precludes autonomy. Further, atomicity requires that the subtransactions run mutual commit protocols, which can cause resources to be locked up for long periods.

For these reasons, the traditional ACID transaction model has been found unacceptable for modern heterogeneous systems. Consequently, a large number of extended transaction models have been proposed recently, which relax the ACID model in various ways [Elmagarmid, 1992; Hsu, 1993]. However, whereas the ACID model comes with a scheduling discipline, namely, serializability, the newer extended transaction models do not typically have as clear a scheduling approach associated with them. This has led to calls for generic facilities for specifying and scheduling extended transaction models, e.g., [Kaiser, 1994, pp. 428–429].

Contributions

We present an approach through which workflows can be efficiently scheduled in a distributed manner. We base our approach on an algebra of events that we previously developed for representing and reasoning about *intertask dependencies.*[1] Formal specifications yield predictable behavior and enable efficiencies in implementation without jeopardizing soundness.

The present paper develops a temporal language with a novel formal semantics. *Guards* on events are compiled from the workflow specifications. The guard on each event is localized on that event and used to control the workflow 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 compilation of the guards. Further, declarative primitives also facilitate run-time modifications of workflows, e.g., in response to exception conditions. We

[1]The present paper is self-contained; details of the algebra are available in [Singh, 1996].

now highlight some key features of our approach relative to recent research (section 8 has details).

- We provide a formal semantics and closely relate it to how workflows are scheduled in our approach.
- Previous approaches apply only to component tasks that are loop-free [Attie *et al.*, 1993; Klein, 1991a; Günthör, 1993; Chrysanthis & Ramamritham, 1994]. Our approach, by contrast, makes no assumptions about the structure of the tasks. Our solution is conceptually simple, but involves some technical subtlety. It is essential to be able to handle ongoing or iterative activities in most settings where workflows are of interest.
- Most existing scheduling approaches are centralized. [Klein, 1991a] claims a purely distributed scheduler, but gives few details even in the longer article [Klein, 1991b]. Recently, some distributed prototypes have been proposed, e.g., [Sheth *et al.*, 1996], but these are not given a formal basis.

Philosophical Remarks

This paper seeks to better delineate the fundamental limits of workflow scheduling. Although we propose advanced techniques for scheduling, we obviously cannot go beyond what distributed computing allows, e.g., in terms of the necessary message exchanges [Chandy & Misra, 1986]. Although we cannot violate the laws of distributed computing, we do enable customizable, declarative specifications through which only the essential restrictions are stated, and the computations are decoupled as much as possible.

It is instructive to recall the history of traditional transaction theory. A significant research effort by the database community converged to the conclusion that strict and serializable transactions are often the most practicable option [Gray & Reuter, 1993]. Intuitively, strictness and serializability imply that activities that share resources should access those resources one by one. In retrospect, this may be obvious, but it took considerable effort to determine so. Similarly, as our approach has evolved, it has tended toward increased conceptual simplicity captured through increasing technical subtlety. We attempt to explain our technical motivations along with our results.

In effect, we show how different activities can proceed concurrently taking into account intertask dependencies and the inherent attributes of their events. These activities exchange relevant information as it becomes available. However, when the dependencies require some of the activities to be ordered, the appropriate activities execute one at a time—just as one would desire.

Organization

The rest of the paper is organized as follows. Section 2 describes our system model, its relation to heterogeneous environments, and its implementation. Section 3 summarizes our algebra for specifying workflows. Section 4 introduces our temporal logic, gives the conceptual intuitions behind *guards*—the cornerstone of our approach—and shows how they can automatically be compiled. Section 5 describes how guards can be used to execute workflows in a naturally distributed manner that respects the attributes of the events. Section 6 presents the main technical results of our approach, establishing its correctness and justifying a series of efficiency-enhancing transformations. Section 7 discusses some enhancements, including how to handle arbitrary tasks. Section 8 reviews the relevant literature.

2 Execution Model and Implementation

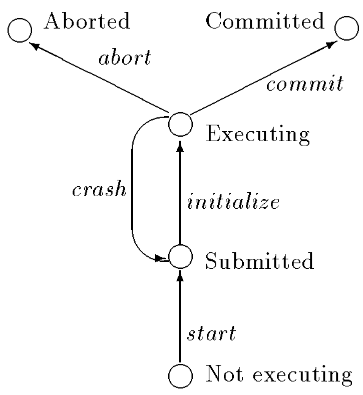


Figure 1: Task agent for an IMS transaction [Rusinkiewicz & Sheth, 1994]

Our execution model is designed to maximize distribution and autonomy. Centralization is clearly undesirable in distributed, heterogeneous environments. Often distribution is an inherent feature of the environments in which workflows are needed, e.g., [Schwenkreis, 1996]. Our approach associates an *agent* with each task or transaction.[2] The agent embodies a coarse description of the task, including only states and transitions (or *events*) that are significant for coordination [Chrysanthis & Ramamritham, 1994]. Our approach applies to arbitrary agents and to any set of significant events.

[2]This is the common database usage of *agent*. There appear to be deeper connections between our approach and agents more generally understood, but those are beyond the scope of this paper.

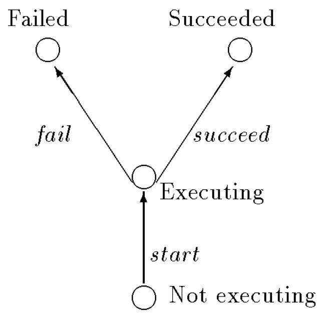


Figure 2: Task agent for a typical application [Rusinkiewicz & Sheth, 1994]

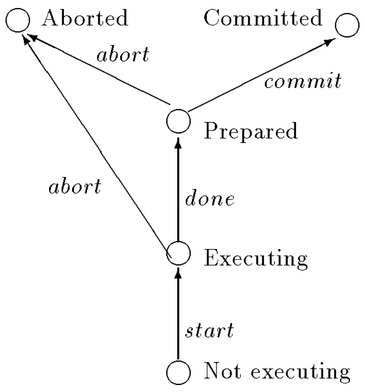


Figure 3: Task agent for a typical transaction [Elmasri & Navathe, 1994, p. 535]

Figure 1 shows an example task agent suitable for an IMS transaction modeled as follows. An IMS database transaction when submitted enters the executing state, from where it may proceed to abort or commit. If it crashes during processing, it may be resubmitted. Figure 2 shows a task agent suitable for a typical application (e.g., as initiated through a shell call), which enters its executing state when started, from where it may fail or succeed. Failure or success is usually associated with a return code. Figure 2 shows a task agent suitable for a transaction that has an explicit prepared state, which enables mutual commit protocols across transactions. This is a simplified version of the one presented in [Gray & Reuter, 1993, figure 10.20, p. 573]. Our approach is not limited to the above agents.

The agent performs an important function: it interfaces 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. Local autonomy is maximized by the above approach, 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 thus the task's interface is not violated. This approach lends itself to distribution, since the agents can be naturally placed close to their respective tasks.

To achieve distribution, we instantiate an *actor* [Agha, 1986] for each event. Actors are lightweight threads that communicate through symbolic addresses and live in an object-oriented execution environment [Tomlinson *et al.*, 1993]. At most one copy of the execution environment is needed at each host: this exists as a single process that manages low-level communications and data resources. Modules to access various database management systems have been implemented.

The actor for an event maintains its current *guard* and manages 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 receiving actor's guard becomes true, then its parked event, if any, is enabled. In our implementation, the `attempt` method on events is *reflective* in that the calling context is passed as an explicit argument. The context can be stored in a parked event and resumed when appropriate. Thus events can be attempted without having to break up the source code to use callbacks.

Additional complexity is involved in handling several important situations. One, the events may have mutual constraints, which must be processed to allow effective progress. Two, the events may have to be proactively triggered, or may be such that they cannot be delayed or rejected. Three, some events may be instantiated multiple times, and we must ensure that all and only the right event instances occur. We discuss these variations below. Our approach has the following main steps:

1. the events are created (as objects)
2. the dependencies are asserted (from which the guards can be automatically compiled)
3. the guards are installed
4. the guards link the events into a graph, which is preprocessed to detect mutual constraints and messages set up to achieve correct behavior
5. the events are activated and made to listen on their input ports.

As the task agents execute, they attempt different events, leading to other agents being executed, thereby enacting the workflow.

Our execution model relates to the Workflow Management Coalition (WfMC) reference model [WfMC, 1995]. Our system can be viewed as providing a "workflow enactment service." Our algebraic language provides a rudimentary means to formally specify processes into which high-level languages may be translated. Based on the specifications, our run-time environment deals with "workflow client applications" that can attempt events in the enactment service. The run-time environment can also trigger "invoked applications," which can be arbitrary programs. We believe that a rigorous approach such as ours will facilitate interoperation among workflow management systems. However, we emphasize that although our execution model has been prototyped, it is by no means an implementation of the reference model. Our focus is on the theoretical problems and we have not invested the system-building effort necessary to make a real product. However, as of July 1996, there is a possibility that our research will be commercialized by a major software vendor.

3 Background: Event Algebra

Our formal language, $\mathcal{E}$, algebraically specifies acceptable computations or *traces* through intertask dependencies [Singh, 1996]. A *dependency*, D, is an expression of $\mathcal{E}$. A *workflow*, $\mathcal{W}$, is a set of dependencies.

3.1 Syntax and Semantics

Event symbols are the atoms of $\mathcal{E}$.[3] We introduce for each event symbol e a symbol $\overline{e}$ corresponding to its complement ($\overline{\overline{e}} = e$). $\mathcal{E}$ contains constants 0 and $\top$, and operators for choice ($\vee$), interleaving ($\wedge$), and sequence ($\cdot$). Roughly, e (resp. $\overline{e}$) means that event e (resp. $\overline{e}$) should occur; $E \vee F$ (resp. $E \wedge F$) means that one of (resp. both) E and F should hold; and, $E \cdot F$ means that E should hold on the initial part of a trace and F on the remainder. Each expression or dependency identifies a set of event traces, namely, those of which it is a true description. 0 is true of no trace, and $\top$ is true of all. $\Sigma \neq \emptyset$ is the set of significant events and Γ is the *alphabet*.

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

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

Traces are finite or infinite sequences of events, which describe a fragment of a possible computation in the system. Traces are written as event sequences enclosed in $\langle$ and $\rangle$ (for convenience, we overload event symbols with the events they denote, but our usage is unambiguous). $u, \ldots$ refer to traces and $i, \ldots$ to indices. For $1 \leq i \leq |u|$, u_i denotes the ith event in u; it is undefined otherwise. Thus for $v = \langle e\overline{f}\rangle$, $v_1 = e$ and $v_2 = \overline{f}$. $\lambda \triangleq \langle\,\rangle$ is the empty trace.

Definition 1 The universe, $\mathbf{U}_{\mathcal{E}}$, is the set of traces on Γ such that (a) no trace contains both an event and its complement and (b) no event is repeated on any trace.

Our semantics (formally $\models$) 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. For a trace u and an expression E, $u \models E$ means that u satisfies E.

Semantics 1 $u \models f$ iff $(\exists j : u_j = f)$, where $f \in \Gamma$

Semantics 2 $u \models E_1 \vee E_2$ iff $u \models E_1$ and $u \models E_2$

Semantics 3 $u \models E_1 \cdot E_2$ iff $(\exists v, w : u = vw$ and $v \models E_1$ and $w \models E_2)$

[3]For ease of exposition and to facilitate comparison with the literature, we initially assume that the symbols denote unique event instances. In section 7.3 we allow multiple instantiation.

Semantics 4 $u \models E_1 \wedge E_2$ iff $u \models E_1$ and $u \models E_2$

Semantics 5 $u \models \top$

For convenience, we define the *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. $E_1 \vee E_2$ denotes the union of the sets for E_1 and E_2. Lastly, $E_1 \wedge E_2$ denotes the intersection of the sets for E_1 and E_2.

As running examples, we use two dependencies due to [Klein, 1991a], which are closely related to the primitives in [Attie *et al.*, 1993; Chrysanthis & Ramamritham, 1994; Günthör, 1993]. $e < f$ means that if both events e and f happen, then e precedes f; $e \rightarrow f$ means that if e occurs then f also occurs (before or after e). Examples 1 and 2 show how to formalize these.

Example 1 Let $D_< = \overline{e} \vee \overline{f} \vee e \cdot f$. Let $\tau \in \mathbf{U}_\mathcal{E}$ satisfy $D_<$. If τ satisfies both e and f, then e and f occur on τ. Thus, neither $\overline{e}$ nor $\overline{f}$ can occur on τ. Hence, τ must satisfy $e \cdot f$, which requires that an initial part of τ satisfy e and the remainder satisfy f. That is, e must precede f on τ. ▮

Example 2 Let $D_\rightarrow = \overline{e} \vee f$. Let $\tau \in \mathbf{U}_\mathcal{E}$ satisfy $D_\rightarrow$. If τ satisfies e, then e occurs on τ. Thus, $\overline{e}$ cannot occur on τ. Hence, f must occur somewhere on τ. ▮

Now we define a workflow, simplified for expository ease. This indicates how the various requirements may be expressed as dependencies.

Example 3 Consider a workflow which attempts to *buy* an airline ticket and *book* a car for a traveler, such that both or neither task should have an effect. Mutual commit protocols cannot be executed, since the airline and car rental agency are different enterprises and their databases may not have a visible precommit state.

Assume that (a) the booking can be canceled: thus *cancel* compensates for *book*, and (b) the ticket is nonrefundable: *buy* cannot be compensated. Assume all subtasks have at least *start*, *commit*, and *abort* events, as in Figure 3. For simplicity, assume that *book* and *cancel* always commit. Now the desired workflow may be specified as follows: (D_1) $\overline{s_{buy}} \vee s_{book}$ (initiate *book* upon starting *buy*), (D_2) $\overline{c_{buy}} \vee c_{book} \cdot c_{buy}$ (if *buy* commits, it commits after *book*—this is reasonable since *buy* cannot be compensated and commitment of *buy* effectively commits the entire workflow), (D_3) $\overline{c_{book}} \vee c_{buy} \vee s_{cancel}$ (compensate *book* by *cancel*), and (D_4) $\overline{s_{cancel}} \vee c_{book} \wedge \overline{c_{buy}}$ (start *cancel* only if necessary).

Note that D_2 explicitly orders c_{book} before c_{buy}—as in Example 1 above. However, D_1, D_3, and D_4 do not order any events—see Example 2 above. However, to be triggered, the events should have the attribute *triggerable*—introduced in section 5.2. The scheduler causes the events to occur when necessary, and may order them before or after other events as it sees fit. ▌

The above workflow is informally discussed in Example 6 and formally scheduled in section 6.6.

3.2 Enforcing Dependencies

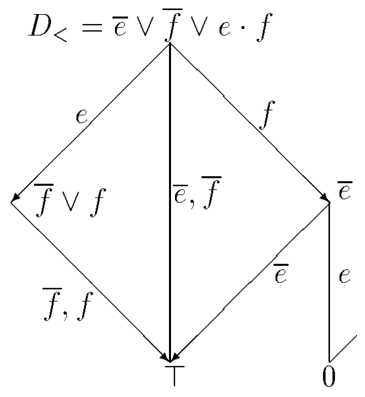


Figure 4: Symbolic representations of scheduler states and transitions for $D_<$

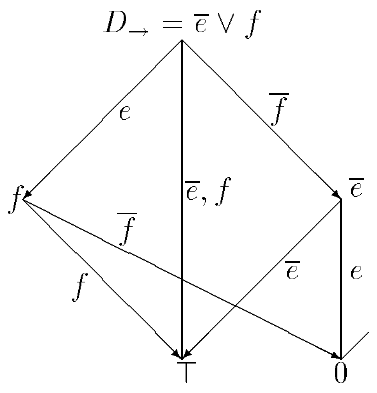


Figure 5: Symbolic representations of scheduler states and transitions for $D_\rightarrow$

The scheduler's behavior is determined by the traces it can allow. Initially, these are given by the stated dependencies. As events occur, the possible traces get narrowed down. Intuitively, the state after each event equals the remnant of the dependency yet to be enforced.

Example 4 (Figure 4) 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). Similarly, (Figure 5) for $D_\rightarrow$ $\overline{e}$ or f may happen right away; if e happens first, it must be followed by f and if $\overline{f}$ happens first, it must be followed by $\overline{e}$. ▮

Importantly, the state changes illustrated above can be symbolically computed using the algebraic operation of *residuation* (/). Semantics 6 means that / yields the most general set of traces for the destination state given the source state and the event transition. Note that the operator / does not occur in $\mathcal{E}$ and cannot be used in the specifications.

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

Interestingly, / can be computed symbolically by the following set of equations. $\vee$ is nested within $\wedge$ and $\cdot$ is nested within $\vee$. CNF can be obtained by repeated application of the distribution laws. Thus E in equations 5, 7, and 8 must be an atom or a sequence expression. 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 $(E_1 \wedge E_2)/e = ((E_1/e) \wedge (E_2/e))$

Residuation 4 $(E_1 \vee e_2)/e = (E_1/e \vee E_2/e)$

Residuation 5 $(e \cdot E)/e = E$

Residuation 6 $E/e = E$, if $e \not\in \Gamma_E$

Residuation 7 $(e' \cdot E)/e = 0$, if $e \in \Gamma_E$

Residuation 8 $(\overline{e} \cdot E)/e = 0$

Example 5 In Figures 4 and 5, we can verify that residuating each state label by any of its out-edge labels yields the label of the next state. For instance, $(\overline{e} \vee \overline{f} \vee e \cdot f)/e = ((\overline{e}/e) \vee (\overline{f}/e) \vee (e \cdot f/e)) = (0 \vee \overline{f} \vee f) = (\overline{f} \vee f)$. Similarly, $(\overline{e} \vee f)/\overline{f} = \overline{e}$. ▮

Theorem 1 ([Singh, 1996]) Equations 1 through 8 are sound and complete. ▮

4 Guards on Events

The preceding 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 and implemented a distributed *event-centric* one. This enables the information pertinent to decisions about an event e to be localized on e. As events prepare to happen and happen, messages are sent to ensure that other events learn of their recent or imminent occurrence. Thus only essential messages need be transmitted. Before we introduce additional theoretical concepts, we give an informal example of how our approach applies to the workflow of Example 3.

Example 6 For simplicity, we show only the events explicitly mentioned in the dependencies of Example 3. Assume s_{buy} is initiated from outside. Then, this event is allowed, but (due to D_1) a message is sent to s_{book} to trigger it. Let *book* now attempt to commit—this is allowed because s_{cancel} can be triggered if necessary. A message is sent to s_{cancel}, which receives the message but is not yet triggered. Suppose *buy* aborts (the scheduler cannot prevent an abort). Another message is sent to s_{cancel}, which this time is triggered. ▮

Intuitively, an event e occurs when the scheduler (a) accepts e if requested by e's task agent, (b) triggers e, or (c) rejects $\overline{e}$ if $\overline{e}$ is requested. The scheduler has no choice but to accept nonrejectable events like *abort*. Our, or any, implementation of the above 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. Intuitively, the guard is the weakest condition that guarantees correctness if the event occurs. 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

The guards on events are temporal expressions. This is necessary so that decisions made on different events can be sensitive to the state of the system, particularly with regard to which events have occurred, which have not occurred but are expected to occur, and which will never occur. Representing these conditions is essential to properly enforcing the dependencies. We propose a temporal language that has the necessary expressiveness. We give it a novel formal semantics that yields a number of important properties, in particular allowing stability of events, yet preserving event orderings (these are explained below).

$\mathcal{T}$ is the formal language in which the guards are expressed. This language captures the above distinctions. Intuitively, $\Box E$ 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. $E_1 \cdot E_2$ means that E_2 has occurred preceded by E_1.

Syntax 5 $\mathcal{E} \subseteq \mathcal{T}$

Syntax 6 $E_1, E_2 \in \mathcal{T}$ implies that $E_1 \vee E_2$, $E_1 \wedge E_2 \in \mathcal{T}$

Syntax 7 $E_1, E_2 \in \mathcal{T}$ implies that $\Box E_1$, $\Diamond E_1$, $E_1 \cdot E_2 \in \mathcal{T}$

Syntax 8 $E \in \mathcal{T}$ implies that $\neg E_1 \in \mathcal{T}$

For simplicity, we assume that the above operators have the following binding precedence (in decreasing order): $\neg$, $\cdot$, $\Box$ and $\Diamond$, $\wedge$, $\vee$.

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. In some respects, our semantics is intuitively similar to the standard one for linear temporal logics [Emerson, 1990], but there are important differences. One, our traces are sequences of events, not of states. Two, most of our semantic definitions are given in terms of a pair of indices (hence they implicitly involve intervals, rather than points). 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.

Definition 2 A trace, u, is *maximal* iff for each event, either the event or its complement occurs on u.

Definition 3 $\mathbf{U}_{\mathcal{T}} \triangleq \{u : u \in \mathbf{U}_{\mathcal{E}} \text{ and } (\forall e \in \Gamma : (\exists j : u_j = e) \text{ or } (\exists j : u_j = \overline{e}))\}$

Observation 2 All traces in $\mathbf{U}_{\mathcal{T}}$ are maximal. ∎

Observation 3 $\lambda \not\in \mathbf{U}_{\mathcal{T}}$, for $\Gamma \neq \emptyset$. ∎

As stated in section 3, we assume $\Gamma \neq \emptyset$. The following semantic definitions capture useful intuitions about $\mathcal{T}$. Semantics 7, which involves just one index, invokes the semantics with the entire trace until the present moment. In the remaining definitions, which involve pairs of indices, the second index is interpreted as the present moment. Semantics 9, 10, 12, and 13 are as in traditional formal semantics. Semantics 14 and 15 involve looking into the future of the present moment. Semantics 8 and 11 capture the dependence of an expression on the immediate past, bounded by the first index of the semantic definition. Semantics 11 is the only definition that introduces a nonzero first index.

Semantics 7 $u \models_i E$ iff $u \models_{0,i} E$

Semantics 8 $u \models_{i,k} f$ iff $(\exists j : i \leq j \leq k$ and $u_j = f)$, where $f \in \Gamma$

Semantics 9 $u \models_{i,k} E_1 \vee E_2$ iff $u \models_{i,k} E_1$ or $u \models_{i,k} E_2$

Semantics 10 $u \models_{i,k} E_1 \wedge E_2$ iff $u \models_{i,k} E_1$ and $u \models_{i,k} E_2$

Semantics 11 $u \models_{i,k} E_1 \cdot E_2$ iff $(\exists j : i \leq j \leq k$ and $u \models_{i,j} E_1$ and $u \models_{j+1,k} E_2)$

Semantics 12 $u \models_{i,k} \top$

Semantics 13 $u \models_{i,k} \neg E$ iff $u \not\models_{i,k} E$

Semantics 14 $u \models_{i,k} \Box E$ iff $(\forall j : k \leq j \Rightarrow u \models_{i,j} E)$

Semantics 15 $u \models_{i,k} \Diamond E$ iff $(\exists j : k \leq j$ and $u \models_{i,j} E)$

Example 7 Let $u = \langle efg \ldots \rangle$ be a trace in $\mathbf{U}_{\mathcal{T}}$. One can verify that

- $u \models_0 \Diamond g$
- $u \models_0 \neg e \wedge \neg f \wedge \neg g$
- $u \models_0 \Diamond(f \cdot g)$
- $u \not\models_0 \Diamond(g \cdot f)$
- $u \models_1 \Box e \wedge \neg f \wedge \neg g$
- $u \not\models_1 (e \cdot g)$
- $u \models_3 (e \cdot g)$

Thus the semantics behaves quite as expected. In particular, $(e \cdot g)$ is satisfied when it is in the past of the given index. The order of the arguments to $\cdot$ remains important even in the context of a $\Diamond$. ∎

4.2 Important Properties

Definition 4 $E \cong F \triangleq (\forall i,k : u \models_{i,k} E \text{ iff } u \models_{i,k} F)$, where $E, F \in \mathcal{T}$.

Definition 5 $E \equiv F \triangleq u \models E \text{ iff } u \models F$, where $E, F \in \mathcal{E}$.

Thus, $\cong$ means equivalence for temporal expressions, whereas $\equiv$ means equivalence for algebraic expressions. Both relations are abbreviations, and not in our formal object languages. Observations 4 and 5 result from a fundamental difference in our semantics for $\mathcal{E}$ and $\mathcal{T}$.

Observation 4 $(\forall e : \top \ncong e \vee \overline{e})$ and $(\forall e : 0 \cong e \wedge \overline{e})$. Also, $(\forall e : \top \cong \Diamond e \vee \Diamond \overline{e})$. ∎

Observation 5 $(\forall e : \top \not\equiv e \vee \overline{e})$ and $(\forall e : 0 \not\equiv e \wedge \overline{e})$. ∎

Definition 6 An event e is *stable* iff if e is satisfied at a given index, then it is satisfied at all future indices as well.

Observation 6 means that Semantics 8 validates the *stability* of all events. However, Observation 7 shows that the temporal operators cannot always be eliminated. Intuitively, $\neg e$ means "not yet e."

Observation 6 $\Box e \cong e$. ∎

Observation 7 $\Box \neg e \ncong \neg e$. ∎

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

Table 1: Temporal operators for $\Gamma = \{e, \overline{e}\}$

Example 8 The possible maximal traces for $\Gamma = \{e, \overline{e}\}$ are $\{\langle e \rangle, \langle \overline{e} \rangle\}$. 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. Table 1 illustrates the following results:

- $\Box e \vee \Box \overline{e} \ncong \top$—neither e nor $\overline{e}$ may have occurred at certain times, e.g., initially
- $\Diamond e \vee \Diamond \overline{e} \cong \top$—eventually either e or $\overline{e}$ will occur
- $\Diamond e \wedge \Diamond \overline{e} \cong 0$—both e and $\overline{e}$ will not occur
- $\Diamond e \vee \Box \overline{e} \ncong \top$—initially, $\overline{e}$ has not happened, but e may not be guaranteed
- $\neg e \vee \Box e \cong \top$ and $\neg e \wedge \Box e \cong 0$—$\neg e$ is the boolean complement of $\Box e$
- $\neg e \vee \Box \overline{e} \cong \neg e$—$\Box \overline{e}$ entails $\neg e$.

The above and allied results were our main motivation in designing the formal semantics of $\mathcal{T}$. For larger alphabets, the set of traces is larger, but there is no conceptual difference, and the above results hold in general. ∎

Intuitively, there are two candidate meanings of an event atom e:

- e is true precisely when e has occurred on the given trace and will remain occurred forever
- e is true precisely when it is definite that e will occur on the given trace, even if e has not occurred yet.

The former corresponds to the $\Box$ operator under stability. The latter corresponds to the $\Diamond$ operator. By using the language $\mathcal{T}$ instead of $\mathcal{E}$ for internal reasoning, we can make the above distinction explicit.

Observations 8, 9, 10, and 11 follow from the above semantics. Observation 8 justifies writing $E_1 \cdot (E_2 \cdot E_3)$ as $(E_1 \cdot E_2 \cdot E_3)$. $\mathcal{P}$ is the negation-free fragment of $\mathcal{T}$. Stability and additional distribution laws (Observations 12, 13, and 14) hold for $\mathcal{P}$. Observations 15 and 16 help simplify long sequence expressions.

Observation 8 $E_1 \cdot (E_2 \cdot E_3) \cong (E_1 \cdot E_2) \cdot E_3$. ∎

Observation 9 $e_1 \cdot \ldots \cdot e_n \cong \bigwedge_{1 \leq i < n} e_i \cdot e_{i+1}$. ∎

Observation 10 $\Box(E_1 \wedge E_2) \cong \Box E_1 \wedge \Box E_2$. ∎

Observation 11 $\Diamond(E_1 \vee E_2) \cong \Diamond E_1 \vee \Diamond E_2$. ∎

Definition 7 $\mathcal{P} \triangleq$ the language obtained using only the syntax rules 5, 6, and 7.

Observation 12 $\Box E \cong E$, where $E \in \mathcal{P}$. ∎

Observation 13 $\Box(E_1 \vee E_2) \cong \Box E_1 \vee \Box E_2$, where $E_1, E_2 \in \mathcal{P}$. ∎

Observation 14 $\Diamond(E_1 \wedge E_2) \cong \Diamond E_1 \wedge \Diamond E_2$, where $E_1, E_2 \in \mathcal{P}$. ∎

Observation 15 $\Diamond(e_1 \cdot \ldots \cdot e_n) \cong \bigwedge_{1 \leq i < n} \Diamond(e_i \cdot e_{i+1})$. ∎

Observation 16 $\Box(e_1 \cdot \ldots \cdot e_n) \cong \bigwedge_{1 \leq i < n} \Box(e_i \cdot e_{i+1})$. ∎

4.3 Compiling Guards

We now use $\mathcal{T}$ to compile guards from dependencies. For expository ease, we begin with a straightforward approach, which is intuitively correct, but not very effective. Section 5.2 discusses additional features. Section 6.3 exploits the special properties of our formal approach to give a series of formal results that improve elegance and efficiency.

The key idea behind guards is that they precisely capture the requirements of the dependencies. Thus they must permit precisely the set of traces that satisfy the given dependencies. In order to capture this intuition, we associate a set of *paths* with each dependency D. A path ρ is a sequence of event symbols that residuate D to $\top$. In other words, the dependency is satisfied if the events in the path occur in the designated order. We require that $\Gamma_\rho \supseteq \Gamma_D$, i.e., all the events in D feature (possibly complemented) in ρ. This is to make sure that all the relevant information is considered and declarative specifications (dependencies) can be converted into correct executions (abstractly given by paths). A path corresponds to a unique trace consisting of the events named in the path. For expository ease, we shall overload paths to refer to their corresponding traces. Although a path may not be maximal, maximal traces can be derived from it.

Recall that dependencies are expressions in $\mathcal{E}$. 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

- the events on ρ up to e have occurred in the right sequence (this is given by $pre(\rho, e)$, the *pre-state* of e in ρ), and
- the events of ρ after e have not occurred, but will occur in the right sequence (this is given by $post(\rho, e)$, the *post-state* of e in ρ).

A path is a consistent sequence of event symbols. Paths thus correspond to members of $\mathbf{U}_\mathcal{E}$; the traces that concern us here are members of $\mathbf{U}_\mathcal{T} \subseteq \mathbf{U}_\mathcal{E}$. $\Pi(D)$ is the set of paths that satisfy D. In the following, let $\rho = \langle e_1 \ldots e_n \rangle$.

Definition 8 $D/\rho \triangleq ((D/e_1)/\ldots)/e_n$.

Definition 9 ρ is a path iff

- the events e_i are distinct
- no event and its complement both occur in ρ.

Definition 10 $\Pi(D) \triangleq \{\rho : \rho \text{ is a path and } \Gamma_\rho \supseteq \Gamma_D \text{ and } D/\rho = \top\}$.

Lemma 17 means that oaths correspond to traces that satisfy the given dependency. Lemma 21 essentially establishes that there is a unique dependency corresponding to any set of paths.

Lemma 17 $\rho \in \Pi(D)$ iff ρ is a path and $\Gamma_\rho \supseteq \Gamma_D$ and $\rho \models D$. ∎

Definition 11 $(v \sqsupseteq w) \triangleq v$ contains the events of w in the same relative order.

Observation 18 If $v \sqsupseteq w$, then $w \in \Pi(D) \Rightarrow v \in \Pi(D)$. ∎

Observation 19 $\Pi(D \wedge E) = \Pi(D) \cap \Pi(E)$. ∎

Observation 20 $D \equiv E \not\Rightarrow \Pi(D) = \Pi(E)$. ∎

Lemma 21 $D \equiv \bigvee_{\rho \in \Pi(D)} \rho$. ∎

Definition 12 $\mathit{pre}(\rho, e) \triangleq$ if $e = e_i$, then $\Box(e_1 \cdot \ldots \cdot e_{i-1})$, else 0.

Definition 13 $\mathit{post}(\rho, e) \triangleq$ if $e = e_i$, then $\neg e_{i+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond(e_{i+1} \cdot \ldots \cdot e_n)$, else 0.

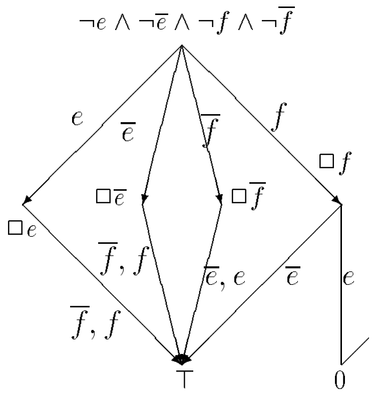
Observation 22 $\mathit{post}(\rho, e_i) = \neg e_{i+1} \wedge \neg \overline{e_{i+1}} \wedge \ldots \wedge \neg e_n \wedge \neg \overline{e_n} \wedge \Diamond(e_{i+1} \cdot \ldots \cdot e_n)$. ∎

Now we can formally define guards. $\mathsf{G}_b(\rho, e)$ denotes the guard on e due to path ρ (b is mnemonic 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 understood. It is perhaps apparent that this definition is redundant, essentially repeating information about the entire path on each event. Later, we shall remove this redundancy to obtain an approach that is equivalent, but far superior.

Definition 14 $\mathsf{G}_b(\rho, e) \triangleq \mathit{pre}(\rho, e) \wedge \mathit{post}(\rho, e)$.
$\mathsf{G}_b(D, e) \triangleq \bigvee_{\rho \in \Pi(D)} \mathsf{G}_b(\rho, e)$.
$\mathsf{G}_b(\mathcal{W}, e) \triangleq \bigwedge_{D \in \mathcal{W}} \mathsf{G}_b(D, e)$.

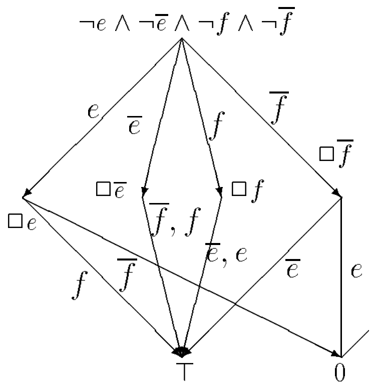
Observation 23 $u \models_k \mathsf{G}_b(D, e) \Rightarrow (\exists \rho \in \Pi(D) : u \models_k \mathsf{G}_b(\rho, e))$. ∎

Figures 6 and 7 illustrate our procedure for the dependencies introduced above. Each figure encodes all traces on $\Gamma = \{e, \overline{e}, f, \overline{f}\}$. The traces that end in a state labeled 0 are not paths in the given dependency (as defined above), and therefore contribute 0. The traces ending in $\top$ are paths in the given dependency. Their contribution is the pre-state for the event conjoined with the label of the post-state. In the figures, we combine the paths into a graph to reflect the "state of the scheduler" intuition



$D_< = \overline{e} \vee \overline{f} \vee e \cdot f$

Figure 6: Guards with respect to $D_<$



$D_\rightarrow = \overline{e} \vee f$

Figure 7: Guards with respect to $D_\rightarrow$

described in section 3.2 and to relate better to the results of section 6. The initial node in the dependency representation is labeled $\neg e \wedge \neg\overline{e} \wedge \neg f \wedge \neg\overline{f}$ to indicate that no event has occurred yet—this relies on observation 22. 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 9 Using Figure 6, we can compute the guards for the events in $D_<$.

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

- $\mathsf{G}_b(D_<, \overline{f}) = \top$.
- $\mathsf{G}_b(D_<, f) = (\neg e \wedge \neg \overline{e} \wedge \Diamond \overline{e}) \vee \Box e \vee \Box \overline{e}$, which simplifies to $\Diamond \overline{e} \vee \Box e$.

Consequently, $\overline{e}$ can occur at any point in the computation, 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. ▌

Example 10 Using Figure 7, we can compute the guards for the events in $D_\rightarrow$.

- $\mathsf{G}_b(D_\rightarrow, e) = (\neg f \wedge \neg \overline{f} \wedge \Diamond f) \vee (\Box f \wedge \top)$. Since $\Diamond f$ entails $\neg \overline{f}$, this reduces to $(\neg f \wedge \Diamond f) \vee \Box f$. Since $(\Box f \wedge \neg f \cong 0$, $\mathsf{G}_b(D_\rightarrow, e) = \Diamond f \vee \Box f$, which equals $\Diamond f$.
- $\mathsf{G}_b(D_\rightarrow, \overline{f}) = (\neg e \wedge \neg \overline{e} \wedge \Diamond \overline{e}) \vee \Box \overline{e}$, which simplifies to $\Diamond \overline{e}$.
- $\mathsf{G}_b(D_\rightarrow, \overline{e}) = (\neg f \wedge \neg \overline{f} \wedge \Diamond \top) \vee (\Box f \wedge \top) \vee (\Box \overline{f} \wedge \top)$, which reduces to $\top$.
- $\mathsf{G}_b(D_\rightarrow, f) = \top$.

Consequently, $\overline{e}$ and f can occur at any time; e can occur if f has happened or will happen; and $\overline{f}$ can occur if $\overline{e}$ has happened or will happen. ▌

5 Distributed Workflow Scheduling

Execution with guards is straightforward. When an event e is attempted, its guard is evaluated. Since guards are updated whenever an event mentioned in them occurs, evaluation usually means checking if the guard evaluates to $\top$. If e's guard is satisfied, e is executed; if it is 0, it is rejected; else it it made to wait. Whenever an event occurs, a notification is sent to each pertinent event f, whose guards are updated accordingly. If f's guard becomes $\top$, f is allowed; if it becomes 0, f is rejected; otherwise, f is made to wait some more. This is the simplest case. More complex cases are discussed in subsequent subsections.

Examples 11 and 12 give a flavor of distributed scheduling using guards. The remainder of this section takes care of some more complex cases.

Example 11 Using the guards from Example 9, 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.

Events $\overline{e}$ and $\overline{f}$ have their guards equal to $\top$, so they can happen at any time. ▌

Example 12 Using the guards from Example 10, if e is attempted first, e's guard evaluates to $\Diamond f$. Consequently, it is made to wait. However, f's guard is $\top$, so it can happen when attempted. In that case, a notification $\Box f$ is sent to e. Recall that in our logic, $\Box f \Rightarrow \Diamond f$. Thus, upon receipt of this notification, e's guard is simplified to $\top$, enabling e to happen. ▌

5.1 Mutual Constraints Among Events

The execution mechanism should avoid potential race conditions and deadlocks. It should also ensure that the necessary information flows to a guard when needed. Certain problems that may arise with the above naive approach can be averted through preprocessing the guards so as to detect and resolve potential deadlocks. We discuss these issues conceptually here; we formalize them in section 6.

Prohibitory Relationships

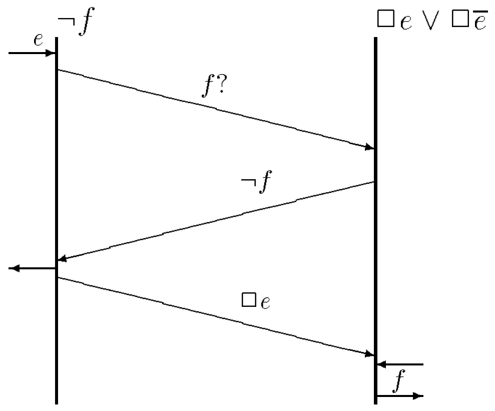


Figure 8: Prohibitory message flows

Let e be the event whose guard is being evaluated. Roughly, a subexpression of the form $\neg f$ evaluates to $\top$, since it means that f has not yet happened: if f had happened, $\neg f$ would have been reduced to 0. But such subexpressions are potentially problematic, since the message announcing f could be in transit when $\neg f$ is evaluated, leading to an inconsistent evaluation. Thus, a message exchange with f's actor is essential to ensure that f has not happened and is not happening. This is an example of a *prohibitory* relationship between events, since f's occurrence can possibly disable e (depending on the rest of the guard of e). Figure 8 diagrams the message flow in this case. Theorem 49 shows how prohibitory message flows can sometimes be avoided in our approach.

Promissory Relationships

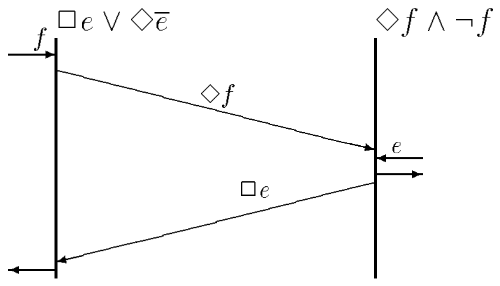

$\Box e \vee \Diamond \overline{e}$
$\Diamond f \wedge \neg f$
f
$\Diamond f$
e
$\Box e$

Figure 9: Promissory message flows

If the guard on an event is neither $\top$ nor 0, then the decision on it can be deferred. The execution scheme must be enhanced to prevent mutual waits in situations where progress can be consistently made.

Example 13 Consider $\mathcal{W}_1 = \{D_<, D_\rightarrow\}$. $\mathsf{G}_b(\mathcal{W}_1, e) = \Diamond f \wedge \neg f$ and $\mathsf{G}_b(\mathcal{W}_1, f) = \Box e \vee \Diamond \overline{e}$. Roughly, this means that e waits for $\Diamond f$, while f waits for $\Box e$. ▌

The guards given in Example 13 do not reflect an inconsistency, since f is allowed to occur after e. One way in which to resolve this kind of wait is through *promises* (this appears related to Klein's approach [1991a]). This relationship is recognized during preprocessing. The events are set up so that when f is attempted, it promises to happen if e occurs. Since e's guard only requires that f occur sometimes, before or after e, e is then enabled and can happen as soon as it is attempted. When news of e's occurrence reaches f, f discharges its promise by occurring. Figure 9 diagrams the message flow in this case.

Example 14 Consider dependency $D_\rightarrow$ of Figure 5 and its "transpose" $D_\rightarrow^T = (\overline{f} \vee e)$. These result in e's guard being $\Diamond f$ and f's guard being $\Diamond e$. Thus, e waits for $\Diamond f$, while f waits for $\Diamond e$. This kind of a cyclic wait can also be handled through promising; either temporal order of e and f is correct. ▌

5.2 Incorporating Event Attributes

Any acceptable schedule must respect the intrinsic properties or *attributes* of the events that occur in it. We now show how event attributes can be naturally incorporated into our approach. The following attributes were introduced in [Attie *et al.*, 1993]: (a) *forcible:* events that the system can initiate; (b) *rejectable:* events that the system can prevent; and (c) *delayable:* events that the system can delay.

A nondelayable event must also be nonrejectable, because it happens before the system learns of it. Intuitively, such an event is not attempted: the scheduler is notified of its occurrence after the fact. It is possible to have nonrejectable, but delayable, events. Also, an event can be forcible only if its complement is rejectable. For many forcible events, the complement might not actually be attempted, so the point is moot from a scheduling viewpoint. However, from the standpoint of declaratively specifying the traces, the scheduler can cause e to occur in some state only if it can prevent $\overline{e}$ from occurring.

In order to capture the above restriction and to reason more easily about the attributes, it is extremely useful to introduce the attributes *immediate* and *inevitable* as combinations of the above. We also believe that *triggerable* is a more appropriate name for forcible events, because of its actual effect during execution. Thus our attributes are as follows. (Triggerability is treated orthogonally to the other attributes, which are easily seen to be mutually exclusive.)

- *Normal:* events that are delayable and rejectable;
- *Inevitable:* events that are delayable and nonrejectable;
- *Immediate:* events that are nondelayable and nonrejectable; and
- *Triggerable:* events that are forcible.

Example 15 The scheduler may trigger a *start*, but not a *commit* (see Figure 3). However, it can reject or delay a *commit*, but can neither delay nor reject an *abort* (a task may unilaterally abort). The scheduler can delay but not reject a *forget* (not shown), in which a task clears its bookkeeping data and releases its locks. Timer events (not shown) are not delayable, rejectable, or triggerable. In other words, *commit* is normal; *start* is triggerable; *forget* is inevitable; *abort* is immediate and triggerable; and timer events are immediate and not triggerable. ▮

Definition 15 Σ_n, Σ_v, Σ_m, and Σ_t are the sets of normal, inevitable, immediate, and triggerable events, respectively.

The correctness and executability of the schedules generated depends crucially on the attributes of the involved events. It turns out that for triggerable events, the dependencies do not need to be modified, although the execution mechanism must be made proactive. This is achieved through setting up appropriate message flows.

By contrast, for other attributes, the dependencies must be modified, although the execution mechanism remains unchanged. The key intuition is that traces that violate some event attribute must be eliminated. A dependency excludes certain computations as illegal; additional knowledge of event attributes further restricts the set of allowed computations. Thus event attributes can be seen as a means of strengthening a given dependency. The approach of section 4.3 applies to normal events; we consider inevitable and immediate events below.

5.2.1 Inevitable Events

An inevitable event must remain permissible on every trace until it or its complement has happened. (The complement may occur if the given task agent attempts the complement.) Possibly, the event may be delayed—i.e., permitted only after some other events have occurred. Thus the scheduler must realize only those traces on which the given event can occur (sooner or later). The basic approach is as described in section 4.3, except that it is applied to a dependency representation from which certain paths are removed. Intuitively, these are the paths on which there is a risk of violating the inevitability of the given event.

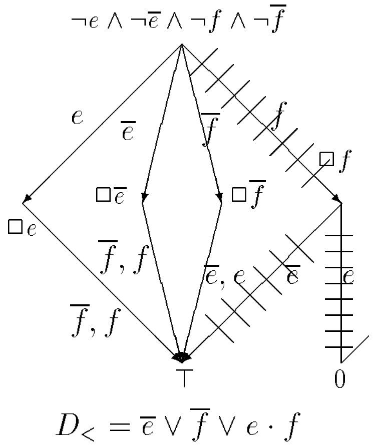
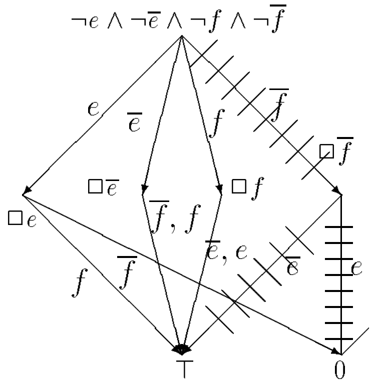


Figure 10: Guards from $D_<$ assuming e is inevitable

Figures 10 and 11 shows the dependencies of Figures 6 and 7, respectively, with unsafe paths deleted to reflect the fact that event e is inevitable. Using this representation, we can readily determine the guards for the various events.

Example 16 Using the fact that event e is inevitable, we have

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



$D_{\rightarrow} = \overline{e} \vee f$

Figure 11: Guards from $D_{\rightarrow}$ assuming e is inevitable

Thus, $\mathsf{G}_b(D_<, e)$ 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. ▌

Example 17 Again referring to Figure 11, we can compute the following:

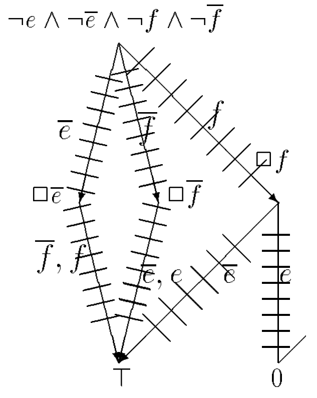
- $\mathsf{G}_b(D_{\rightarrow}, e) = \Diamond f$
- $\mathsf{G}_b(D_{\rightarrow}, \overline{f}) = \Box\overline{e}$
- $\mathsf{G}_b(D_{\rightarrow}, \overline{e}) = (\neg f \wedge \neg\overline{f}) \vee \Box f \cong \neg\overline{f} \vee \Box f \cong \neg\overline{f}$
- $\mathsf{G}_b(D_{\rightarrow},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

Thus the guards on $\overline{e}$ and $\overline{f}$ are strengthened. ▌

5.2.2 Immediate Events

An event that is immediate must be permissible in every state of the scheduler. Thus the scheduling system must never take an action that leaves it in a state where the given event cannot occur immediately. (Of course, as for other events, the complement may occur if the given task agent attempts the complement.) Thus, in essence, the guard of any such event is set to $\top$ by fiat. This category appears to correspond to *uncontrollable* events of [Klein, 1991a] and [Günthör, 1993].

Example 18 Figures 10 and 11 still hold with the assumption that e is not delayable or rejectable. Thus the same guards are obtained as in Examples 16 and 17. ▌



$D_6 = \overline{e} \vee \overline{f} \cdot e$

Figure 12: Extreme example of an immediate event (e)

However, the guards obtained under different attribute assumptions can be different. An extreme case is Example 19. Example dependencies where the guards turn out to be different for immediate than for inevitable events involve more than two events—these are conceptually simple and are left to the reader.

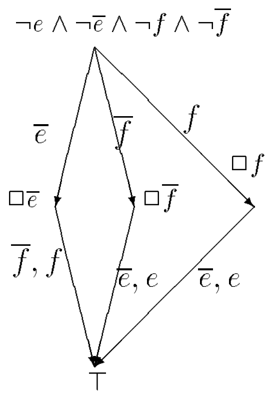
Example 19 Referring to Figure 12, we can readily see that the guards for all the events are 0!
However, if e is inevitable, then the guards are nonzero, as can be readily checked (see Figure 14 below). ▮

5.2.3 Enforceability of Dependencies

A dependency is enforceable if, under the appropriate event attributes, its denotation is nonempty. Dependencies can be enforced during workflow execution as long as appropriate guards for events can be found. Enforceability must be checked when dependencies are asserted. Individual dependencies may become unenforceable if the asserted attributes are overconstraining. Examples 20 and 21 show how the enforceability of dependencies can vary based on the event attributes that are asserted.

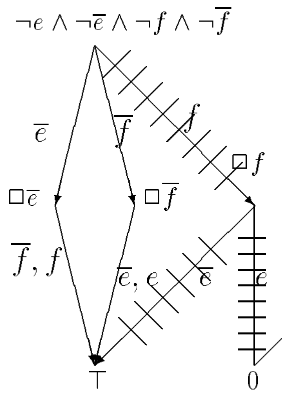
Example 20 Dependency $D_<$ of Figure 10 is enforceable if e is immediate and f is inevitable, because f can be delayed until e or $\overline{e}$ occurs. However, $D_<$ is unenforceable if e is inevitable and f is immediate, because the inevitability of e removes the path beginning with f from the initial state. ▮

Example 21 Figure 13 diagrams a dependency, $D_5 = \overline{e} \vee \overline{f} \cdot e \vee f \cdot e$, which is enforceable when e is inevitable (because e can occur after f or $\overline{f}$), but unenforceable when



$$D_5 = \overline{e} \vee \overline{f} \cdot e \vee f \cdot e$$

Figure 13: Enforceable when e is inevitable; unenforceable when e is immediate



$$D_6 = \overline{e} \vee \overline{f} \cdot e$$

Figure 14: Unenforceable when e and f are inevitable

e is immediate (because e cannot occur right away). D_5 continues to be enforceable when e is inevitable and f is immediate. By contrast, $D_6 = \overline{e} \vee \overline{f} \cdot e$ (diagramed in Figure 14) is unenforceable when e and f are both inevitable, because there is no path on which they both occur. ▮

However, it is possible that dependencies are individually, but not jointly, enforceable. This can be detected when the guards on different events as contributed by different dependencies are conjoined and clash with the attributes of those events.

Example 22 Let $D_7 = \overline{f} \vee \overline{e}$. The dependencies $D_{\rightarrow} = \overline{e} \vee f$ and D_7 are jointly enforceable, although $\mathsf{G}_b(D_<, e) \wedge \mathsf{G}_b(D_7, e) = \Diamond f \wedge \Diamond \overline{f}$, which reduces to 0. This just means that e must always be rejected. However, if we are given the fact that e

is inevitable, then the dependencies are jointly unenforceable, since the guard of an inevitable event must be an expression that will eventually evaluate to $\top$. ▮

5.2.4 Triggerable Events

Triggerable events do not strengthen a dependency or cause any modification in the guards, but affect scheduling in a direct way.

Example 23 Assume we are given $D_{\rightharpoondown}$ with f triggerable. Then the approach of Example 12 will fail because e will wait forever for f. ▮

We must set up the information flow so that f is triggered. A principled way to achieve this is to set up a promissory message from the potential trigger event to the triggerable event, and to arrange the execution mechanism so that triggerable events can be scheduled even though they are not explicitly attempted.

Example 24 Continuing with Example 23, when e is attempted, it promises $\Diamond e$ to f. Since f's guard was already $\top$, it remains $\top$, but the interrupt, i.e., the receipt of a message from e, serves to trigger f. f produces a notification to e, which causes e's guard to become $\top$. Thus e can also happen. ▮

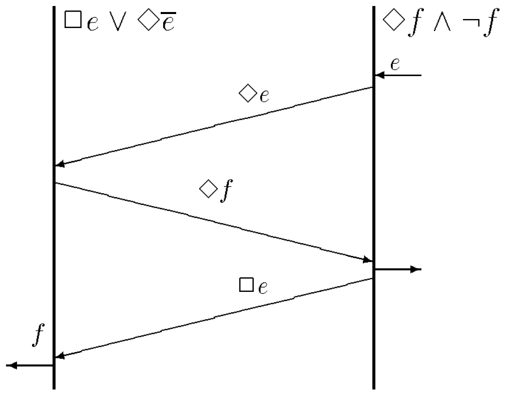

$\Box e \vee \Diamond \overline{e}$
$\Diamond f \wedge \neg f$
e
$\Diamond e$
$\Diamond f$
$\Box e$
f

Figure 15: Triggering a later event

A more complex case arises when the guards on the events are as in Example 13 and Figure 9, but the later event is to be triggered.

Example 25 Let $\mathsf{G}_b(e) = \Diamond f \wedge \neg f$ and $\mathsf{G}_b(f) = \Box e \vee \Diamond \overline{e}$. Let f be triggerable. When e is attempted, it sends a promise of $\Diamond e$ to f. f's guard can change to $\Box e$ as a result—still not enough to execute f, so f promises back. e's guard can change to $\top$, so it happens. Its notification satisfies the condition in f's promise, so f happens. Figure 15 diagrams the message flow in this case. ▮

Sometimes the system may need to trigger an event directly, e.g., when the workflow contains a dependency e. For such situations, we recall that an event or its complement must eventually occur, and that they both cannot occur. Thus if e is triggerable and $\mathsf{G}_b(e) = \top$ and $\mathsf{G}_b(\overline{e}) = 0$, then e is executed, even if no message is received.

6 Formalization: Correctness and Simplification

A careful formalization can help in proving the correctness and acquiring a deeper understanding of the computations being studied. In the present case, we wish to establish that only correct executions can be produced by our approach. In addition, the formalization helps us prove that certain elegant and efficient variations are valid, thereby justifying our use of them.

Correctness in our approach is a concern at three different points, namely:

- When guards are compiled from the stated dependencies
- When guards are preprocessed
- When events are executed.

It is clear that soundness is essential for all three points, because we surely do not wish to allow any unacceptable computations. However, completeness, although desirable, is not essential—a scheduler that may miss out on some acceptable computations can still be useful. This is analogous to the situation in theorem proving—a reasoning system may be incomplete, but it should not be unsound. We address the above three facets of correctness below.

Correctness in our approach depends on the *evaluation strategy* used, which determines how events are scheduled. This is so even for the guard compilation, because we can show that the guards are correct only under the assumption that they will be used to produce executions in a specific manner. We begin with a strategy that is simple but correct and produce a series of more sophisticated strategies that are semantically equivalent to it.

Given a workflow $\mathcal{W}$, S yields a function $\mathsf{S}(\mathcal{W})$ that assign to each event e a temporal expression at each index of each trace. This expression, in $\mathcal{T}$, corresponds to the evolving guard on e with the initial guards based on $\mathcal{W}$. We adopt the convention that metatheory expressions of the form $u \models_i \mathsf{S}(\mathcal{W}, e, v, j)$ can be abbreviated to $u \models_i \mathsf{S}(\mathcal{W}, e)$, wherein we implicitly set $v = u$ and $j = i$. We use $\mathsf{S}(D)$ and $\mathsf{S}(w)$ to relativize S to a dependency D and path w, respectively.

Definition 16 An evaluation strategy is a function $\mathsf{S} : 2^{\mathcal{E}} \mapsto (\Gamma \times \mathbf{U}_{\mathcal{T}} \times \mathbf{N} \mapsto \mathcal{T})$.

We state that an evaluation strategy $\mathsf{S}(\mathcal{W})$ *generates* trace $u \in \mathbf{U}_{\mathcal{T}}$ 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 $\mathsf{S}(\mathcal{W}) \rightsquigarrow u$. Notice that although the guard is verified at the designated index on the trace, its verification might involve future indices on that trace. That is, the guard may involve $\Diamond$ expressions that happen to be true on the given trace at the index of e's occurrence. Because generation assumes looking ahead into the future, it is intuitively more abstract than execution.

Definition 17 $\mathsf{S}(\mathcal{W}) \rightsquigarrow_i u$ iff $i = 0$ or $(\forall j : 1 \leq j \leq i \Rightarrow u \models_{j-1} \mathsf{S}(\mathcal{W}, u_j))$.

Definition 18 $\mathsf{S}(\mathcal{W}) \rightsquigarrow u \triangleq (\forall i : 0 \leq i \leq |u| \Rightarrow \mathsf{S}(\mathcal{W}) \rightsquigarrow_i u)$.

The idea behind generation is to abstractly characterize the traces that can result from the guards under some evaluation strategy. The objective of the exercise is to prove that these traces are in the denotation of the dependencies from which the guards were compiled. We introduce $u \sim_i v$ to mean that trace u agrees with trace v up to index i. $u \sim_i D$ means that a trace u agrees with dependency D up to index i. We overload the relation $\sim$ because the meanings in the two cases are closely related. Lemma 25 relies on the maximality of u.

Definition 19 $u \sim_i v \triangleq i \leq |u|$ and $i \leq |v|$ and $(\forall j : 1 \leq j \leq i \Rightarrow u_j = v_j)$.

Definition 20 $u \sim_i D \triangleq (\exists v \in [\![D]\!] : u \sim_i v)$.

Observation 24 If $[\![D]\!] \neq \emptyset$, then $(\forall u : u \sim_0 D)$. ∎

Lemma 25 $u \sim_{|u|} D \Rightarrow u \in [\![D]\!]$. ∎

6.1 Compilation

In order to establish model-theoretic correctness of the initial compilation procedure given by Definition 14, we begin with a trivial strategy, S_b. S_b sets the guards using G_b and never modifies them. Theorem 28 establishes the soundness of the guard compilation procedure. It asserts that traces generated from guards do not violate the stated dependencies. Theorem 30 establishes the completeness of the guard compilation procedure. It asserts that every trace that satisfies the stated dependencies may be generated by the guards.

Definition 21 $(\forall v, j : \mathsf{S}_b(\mathcal{W}, e, v, j) = \mathsf{G}_b(\mathcal{W}, e))$.

Observation 26 $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$ iff $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(\mathcal{W}, u_j))$. ∎

Lemma 27 If $u \models_{k-1} \mathsf{G}_b(w, u_k)$, then $u \sqsupseteq w$. ∎

Theorem 28 Let $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$. Then, $(\forall D \in \mathcal{W} : u \models D)$. ∎

Observation 29 $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(u, u_j))$. ∎

Theorem 30 Let $(\forall D \in \mathcal{W} : u \models D)$. Then, $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$. ∎

6.2 Evaluation

The guards are initially set according to Definition 14 (or something equivalent to it described later). We make use of Observations 15 and 16 to simplify long sequences of events into conjunctions of sequences of length 2. As events happen, new information is assimilated by updating the guards repeatedly through symbolic reasoning. The operator $\div$ captures the processing required when different messages are assimilated into a guard to yield a modified guard. This operator embodies a set of "proof rules" to reduce guards when an event occurs or is promised. This operator is well-defined because whenever more than one case applies, e.g., because $\Box e$ entails $\Diamond e$, the same results are obtained.

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 new. This is because the stability of events is in tension with ordering. If $e_1 \cdot e_2$ is specified, we wish to refer to the first occurrences of e_1 and e_2—otherwise, we would end up allowing $\langle e_2 e_1 \rangle$, and thereby $e_1 \cdot e_2$ would be violated. For this reason, the updates in those cases are more complex. Lemma 32 means that the operator $\div$ preserves the truth of the original guards.

Definition 22 $G \div M$ is as given by Table 2.

Observation 31 If $\Gamma_G \cap \Gamma_M = \emptyset$, then $G = G \div M$. ∎

Lemma 32 $(\exists k \leq j : u \models_k M$ and $u \models_j G \div M) \Rightarrow u \models_j G$. ∎

The repeated application of $\div$ to update the guards of events corresponds to a new evaluation strategy, $\mathsf{S}_{\div}$. This strategy permits possible traces on which the guards are initially set according to the original definition, but may be updated in response to expressions verified at previous indices.

Definition 23 $\mathsf{S}_{\div}(\mathcal{W}, e, u, 0) \triangleq \mathsf{G}_b(\mathcal{W}, e)$.
$\mathsf{S}_{\div}(\mathcal{W}, e, u, i+1) \neq \mathsf{S}_{\div}(\mathcal{W}, e, u, i) \Rightarrow$
$(\exists k \leq i : u \models_k M$ and $\mathsf{S}_{\div}(\mathcal{W}, e, u, i+1) = (\mathsf{S}_{\div}(\mathcal{W}, e, u, i) \div M))$.

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	$G_1 \div M \wedge G_2 \div M$
$\Box e$	$\Box e$	$\top$
$\Box \overline{e}$	$\Box e$	0
$\Diamond e$	$\Box e$	$\top$
$\Diamond \overline{e}$	$\Box e$	0
$\Box(e_1 \cdot e_2)$	$\Box e_1 \wedge \neg e_2$	$\Box e_2$
$\Box(e_1 \cdot e_2)$	$\Box e_2 \wedge \neg e_1$	0
$\Box(e_1 \cdot e_2)$	$\Box \overline{e_i}$	0
$\Diamond(e_1 \cdot e_2)$	$\Box e_1 \wedge \neg e_2$	$\Diamond e_2$
$\Diamond(e_1 \cdot e_2)$	$\Box e_2 \wedge \neg e_1$	0
$\Diamond(e_1 \cdot e_2)$	$\Box \overline{e_i}$	0
$\neg e$	$\Box e$	0
$\neg \overline{e}$	$\Box e$	$\top$
$\Box(e_1 \cdot e_2)$	$\Diamond \overline{e_i}$	0
$\Box \overline{e}$	$\Diamond e$	0
$\neg \overline{e}$	$\Diamond e$	$\top$
G	M	G, otherwise

Table 2: Guard Evaluation

$\mathsf{S}_{\div}$ does not require that every M that is true be used in reducing the guard. Except for sequence expressions, this enables us to accommodate message delay, because notifications need not be incorporated immediately. Recall that $\Box e$ and $\Diamond e$ are members of $\mathcal{P}$, for which stability holds by Observation 12.

The rules in Table 2 describe how the guards may be updated as information is received about other events. Theorem 33 establishes that the evaluation of the guards according to these rules is sound and complete. All traces that could be generated by the original guards are generated when the guards are updated (completeness) and that any traces generated through the modified guards would have been generated from the original guards (soundness).

The main motivation for performing guard evaluation as above is that it enables us to collect the information necessary to make a local decision on each event. Theorem 33 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 effective the preprocessing is.

Theorem 33 Replacing S_b by $\mathsf{S}_\div$ preserves correctness, i.e., $\mathsf{S}_\div(\mathcal{W}) \leadsto u$ iff $\mathsf{S}_b(\mathcal{W}) \leadsto u$. ∎

6.3 Simplification

We now show how guards as given in Definition 14 can be computed more efficiently. Theorems 36 and 38 show that the computations during guard compilation can be distributed over the conjuncts and disjuncts of a dependency. Since our dependencies are in CNF, this means the constituent sequence terms can be processed independently of each other. Theorem 38 is also important for another reason, namely, because it essentially equates a workflow with the conjunction of the dependencies in it.

These theorems rely on some 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 $\overline{e} \in \Gamma')$. Lemma 34 states 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. Lemma 35 shows that the guards are well-behaved with respect to denotations.

Definition 24 $I(w, \Gamma') \triangleq \{v : \Gamma_v = \Gamma_w \cup \Gamma'$ and $v \sqsupseteq w\}$.

Lemma 34 If $e \in \Gamma_w$, then $\mathsf{G}_b(w, e) = \bigvee_{v \in I(w,\Gamma')} \mathsf{G}_b(v, e)$. ∎

Lemma 35 $D \equiv E \Rightarrow \mathsf{G}_b(D, e) = \mathsf{G}_b(E, e)$. ∎

Theorem 36 $\mathsf{G}_b(D \vee E, e) = \mathsf{G}_b(D, e) \vee \mathsf{G}_b(E, e)$. ∎

Observation 37 $\mathsf{G}_b(e \vee \overline{e}, f) = \top$. ∎

Theorem 38 $\mathsf{G}_b(D \wedge E, e) = \mathsf{G}_b(D, e) \wedge \mathsf{G}_b(E, e)$. ∎

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 essential 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 occur in D. $\Pi(D)$ can be determined from $\Psi(D)$ and vice versa. Lemma 40 means that the guard compilation essentially uses $\Psi(D)$—the other paths in $\Pi(D)$ can be safely ignored. Theorem 43 shows that the guard on an event e due to a dependency D is simply $\Diamond D$. This means that, for most pairs of dependencies and events, guards can be quickly compiled into a succinct expression.

Definition 25 $\Psi(D) \triangleq \{w : w \in \Pi(D), \Gamma_w = \Gamma_D\}$.

Lemma 39 $D \equiv \bigvee_{\rho \in \Psi(D)} \rho$. ∎

Lemma 40 If $e \in \Gamma_D$, then $\mathsf{G}_b(D, e) = \bigvee_{w \in \Psi(D)} \mathsf{G}_b(w, e)$. ∎

Lemma 41 $\mathsf{G}_b(D, e) = \mathsf{G}_b(D \wedge e, e)$. ∎

Lemma 42 $\mathsf{G}_b(e_1 \cdot \ldots \cdot e_n, e) = \Diamond(e_1 \cdot \ldots \cdot e_n)$, if $e \notin \Gamma_{e_1 \cdot \ldots \cdot e_n}$. ∎

Theorem 43 $\mathsf{G}_b(D, e) = \Diamond D$, if $e \notin \Gamma_D$. ∎

6.3.1 Relaxing the Past

Recall that Definition 14 states that for each trace in the denotation of a given dependency, 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. Definition 27 shows that we can replace certain sequences in the past component of the guard definition by conjunctions. Intuitively, the original guards place redundant information on each event, even when the other events would ensure that no spurious traces are realized. G_p^{Δ} gives a relaxed definition of guards, where Δ is a set of events for which the non-relaxed definition is applied (Δ is used when the event attributes are formalized). We omit Δ when it equals $\emptyset$. S_p^{Δ} is the evaluation strategy corresponding to G_p^{Δ}. Theorem 44 establishes the correctness of S_p^{Δ}. As before, $\rho = \langle e_1 \ldots e_n \rangle$.

Definition 26 $\mathit{pre}_p(\rho, e) \triangleq$ if $e = e_i$, then $\Box e_1 \wedge \ldots \wedge \Box e_{i-1}$, else 0.

Definition 27 $\mathsf{G}_p^{\Delta}(\mathcal{W}, e) \triangleq \bigvee_{D \in \mathcal{W}} \mathsf{G}_p^{\Delta}(D, e)$.
$\mathsf{G}_p^{\Delta}(D, e) \triangleq \bigvee_{w \in \Pi(D)} \mathsf{G}_p^{\Delta}(w, e)$.
$\mathsf{G}_p^{\Delta}(\rho, e_i) \triangleq$ if $\{e_1 \ldots e_{i-1}\} \cap \Delta \neq \emptyset$, then $\mathit{pre}(\rho, e_i) \wedge \mathit{post}(\rho, e_i)$, else $\mathit{pre}_p(\rho, e_i) \wedge \mathit{post}(\rho, e_i)$.

Definition 28 $\mathsf{S}_p^{\Delta}(\mathcal{W}, e, u, 0) = \mathsf{G}_p^{\Delta}(\mathcal{W}, e)$.
$\mathsf{S}_p^{\Delta}(\mathcal{W}, e, u, i+1) \neq \mathsf{S}_p^{\Delta}(\mathcal{W}, e, u, i) \Rightarrow$
$e \notin \Delta$ and $(\exists k \leq i : u \models_k M$ and $\mathsf{S}_p^{\Delta}(\mathcal{W}, e, u, i+1) = (\mathsf{S}_p^{\Delta}(\mathcal{W}, e, u, i) \div M))$.

Theorem 44 For all Δ, replacing $\mathsf{S}_{\div}(\mathcal{W})$ by $\mathsf{S}_p^{\Delta}(\mathcal{W})$ preserves correctness. ∎

6.3.2 Relaxing the Past and the Future

We now consider another formulation, which is of interest because it works in several cases of practical interest, although it fails in general. This formulation replaces all sequences in guards with conjunctions. The idea is that each event would locally capture what comes before and what comes after. Interestingly, Lemma 45 shows that this works when individual paths are considered. However, Lemma 46 shows that it fails for dependencies in general. The problem is that when a dependency includes multiple paths, Definition 30 allows the traces generated from the paths to be spuriously combined into traces the original paths would not generate. However, as shown below, we can obtain a positive result by restricting the syntax of dependencies. $\mathsf{S}^{\Delta}_{\wedge}$ is the evaluation strategy corresponding to $\mathsf{G}^{\Delta}_{\wedge}$. As before, $\rho = \langle e_1 \ldots e_n \rangle$.

Definition 29 $post_f(\rho, e) \triangleq$ if $e = e_i$, then $\neg e_{i+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond e_{i+1} \wedge \ldots \wedge \Diamond e_n$, else 0.

Definition 30 $\mathsf{G}^{\Delta}_{\wedge}(\mathcal{W}, e) \triangleq \bigvee_{D \in \mathcal{W}} \mathsf{G}^{\Delta}_{\wedge}(D, e)$.
$\mathsf{G}^{\Delta}_{\wedge}(D, e) \triangleq \bigvee_{w \in \Pi(D)} \mathsf{G}^{\Delta}_{\wedge}(w, e)$.
$\mathsf{G}^{\Delta}_{\wedge}(\rho, e_k) \triangleq pre_p(\rho, e) \wedge post_f(\rho, e)$.

Definition 31 $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, 0) = \mathsf{G}^{\Delta}_{\wedge}(\mathcal{W}, e)$.
$\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, i+1) \neq \mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, i) \Rightarrow$
$(\exists k \leq i : u \models_k M$ and $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, i+1) = (\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, i) \div M))$.

Lemma 45 $\mathsf{S}_{\div}(w) \rightsquigarrow u$ iff $\mathsf{S}^{\Delta}_{\wedge}(w) \rightsquigarrow u$. ∎

Lemma 46 Replacing $\mathsf{S}_{\div}(\mathcal{W})$ by $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W})$ does not preserve correctness. ∎

A *3-dependency* is a member of $\mathcal{E}_3$. Every sequence term in a 3-dependency—which is in CNF—must have 3 or fewer events. A *3-workflow* is a workflow in which every dependency is a 3-dependency. Lemma 47 shows the present formulation applies for 3-workflows, which cover many of the cases of interest.

Syntax 9 $e_1 \cdot \ldots \cdot e_n \in \mathcal{D}_3$ if $n \leq 3$

Syntax 10 $E_1, E_2 \in \mathcal{D}_3$ implies that $E_1 \vee E_2 \in \mathcal{D}_3$

Syntax 11 $\mathcal{D}_3 \subseteq \mathcal{E}_3$

Syntax 12 $E_1, E_2 \in \mathcal{E}_3$ implies that $E_1 \wedge E_2 \in \mathcal{E}_3$

Syntax 13 $0, \top \in \mathcal{E}_3$

Lemma 47 If $\mathcal{W}$ is a 3-workflow, then replacing $\mathsf{S}_{\div}(\mathcal{W})$ by $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W})$ preserves correctness. ∎

6.3.3 Eliminating Irrelevant Guards

We establish an important result in this section, to do with the guards of events not occurring in a dependency. Theorem 48 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 in the given workflow. Thus dependencies in the workflow that don't mention an event can be safely ignored. Intuitively, this makes sense because the events mentioned in D will ensure that D is satisfied in any generated trace. Thus at all indices of any generated trace, we will have $\Diamond D$ anyway. Below, $\mathsf{G}^{\triangle}_{\top}$ and $\mathsf{S}^{\triangle}_{\top}$ have the obvious meanings.

Definition 32 $\mathsf{G}^{\triangle}_{\top}(D, e) = \top$, if $e \not\in \Gamma_D$ and $D \in \mathcal{W}$;
$\mathsf{G}^{\triangle}_{\top}(D, e) = \mathsf{G}^{\triangle}_{p}(D, e)$ otherwise.

Definition 33 $\mathsf{S}^{\triangle}_{\top}(\mathcal{W}, e, u, 0) = \mathsf{G}^{\triangle}_{\top}(\mathcal{W}, e)$.
$\mathsf{S}^{\triangle}_{\top}(\mathcal{W}, e, u, i+1) \neq \mathsf{S}^{\triangle}_{\top}(\mathcal{W}, e, u, i) \Rightarrow$
$(\exists k \leq i : u \models_k M$ and $\mathsf{S}^{\triangle}_{\top}(\mathcal{W}, e, u, i+1) = (\mathsf{S}^{\triangle}_{\top}(\mathcal{W}, e, u, i) \div M))$.

Theorem 48 Replacing $\mathsf{S}^{\triangle}_{p}(\mathcal{W})$ by $\mathsf{S}^{\triangle}_{\top}(\mathcal{W})$ does not violate correctness. ∎

Theorems 38 and 48 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 in a modular fashion and simultaneously obtain simpler expressions that are more amenable to processing. These results are significant because they show how we can evaluate the guards purely symbolically, without ever having to expand a dependency into the paths that validate it.

Importantly, our approach obviates prohibitory message flows in many cases (the exceptions are discussed in section 6.4). This is because expressions of the forms $\Box f$ and $\neg f$, when encountered at run-time, can be evaluated as 0 and $\top$, respectively. Intuitively, this can be thought of as similar to negation as failure—if an event e has not heard of f one way or the other, it can assume that f has not yet happened. One would think such that reasoning would be risky. However, Theorem 49 shows that this simplification does not violate correctness. Thus events can be executed with greater decoupling than one might expect. Note that although $\neg$ expressions are evaluated as $\top$, they cannot be removed from the guard itself. This is because the receipt of a $\Box f$ message can cause them to be reduced to 0. For example, if the guard on e is $\Box f \wedge \Box g \vee \neg f \wedge \neg g$, we would enable e if it is attempted first, but reduce the guard to $\Box g$ upon the receipt of $\Box f$. $\mathsf{S}^{\triangle}_{\neg}$ is the evaluation strategy that reduces $\Box f$ to 0 and $\neg f$ to $\top$, provided $f \not\in \Delta$. The guards update according to $\mathsf{S}^{\triangle}_{\top}$—the reduction kicks in independently at each moment.

Definition 34 $\mathsf{S}^{\Delta}_{\neg}(\mathcal{W}, e, u, i) = \mathsf{S}^{\Delta}_{\top}(\mathcal{W}, e, u, i)|_{(\forall f \in \Gamma \backslash \Delta : \Box f ::= 0, \neg f ::= \top)}$.

Theorem 49 For all Δ, replacing $\mathsf{S}^{\Delta}_{\top}$ by $\mathsf{S}^{\Delta}_{\neg}$ does not violate correctness. ∎

6.4 Formalizing Event Attributes

Now we can formalize the event attributes *inevitable* and *immediate* in terms of how they strengthen a given dependency. We define a transformation, σ, of a set of paths into a set of "safe" paths that satisfy the event attributes. In other words, σ eliminates paths that violate one of the attributes. Since this depends on what other paths are legal, we must apply σ iteratively in order to obtain the desired set. The number of iterations depend on the length of the longest path in the input set. For this reason, we begin the process from $\Psi(D)$, in which the paths are limited to length $|\Gamma_D|/2$.

Definition 35 $\sigma^0(D) \triangleq \Psi(D)$.
$\sigma^{i+1}(D) \triangleq \sigma^i(D) \cap \{w : \beta(w, \sigma^i(D))\}$,
 where $\beta(w, P)$ holds iff

- if $e \in \Gamma_D \cap \Sigma_v$, then $(\forall w_1, w_2 : w = w_1 w_2 \Rightarrow$
  - $e \in w_1$, or
  - $\overline{e} \in w_1$, or
  - $(\exists w_3, w_4 : w_1 w_3 e w_4 \in P))$.
- if $e \in \Gamma_D \cap \Sigma_m$, then $(\forall w_1, w_2 : w = w_1 w_2 \Rightarrow$
  - $e \in w_1$, or
  - $\overline{e} \in w_1$, or
  - $(\exists w_3 : w_1 e w_3 \in P))$.

Definition 36 $\Phi(D) \triangleq \sigma^{|\Gamma_D|/2}(D)$.

$\Phi(D)$ gives the set of safe paths in D that do not risk violating the given attributes. $\alpha(D)$ is D strengthened to accommodate the event attributes. Once the safe paths in a dependency relative to the event attributes have been identified, we can compute $\alpha(D)$ in two different ways. One, (since $\Phi(D)$ is a set of paths), using Lemma 39, we can determine $\alpha(D)$ as the disjunction of the safe paths. Two, we can conjoin D with the complement of the paths that are eliminated. This formally establishes our intuition that event attributes merely strengthen the given dependencies. The rest of the processing can proceed as before. Lemmas 50 and 51 show that the formal definition behaves as desired.

Definition 37 $\alpha(D) \triangleq \bigvee_{w \in \Phi(D)} w$.
$\alpha(\mathcal{W}) = \{\alpha(D) : D \in \mathcal{W}\}$.

Lemma 50 If $[\![\alpha(D)]\!] \neq \emptyset$ and $u \sim_k \alpha(D)$ and $e \in \Sigma_m$ and
$(\forall j : 1 \leq j \leq k \Rightarrow u_j \neq e)$, then $(\exists v : u \sim_k v$ and $v_{k+1} = e$ and $v \sim_{k+1} \alpha(D))$. ∎

Lemma 51 If $[\![\alpha(D)]\!] \neq \emptyset$ and $u \sim_k \alpha(D)$ and $e \in \Sigma_v$ and
$(\forall j : 1 \leq j \leq k \Rightarrow u_j \neq e)$, then $(\exists v, l : u \sim_k v$ and $v_l = e$ and $v \sim_l \alpha(D))$. ∎

$\mathsf{C}(e_1 \cdot \ldots \cdot e_n)$ to mean the complement of $e_1 \cdot \ldots \cdot e_n$. The complement of a sequence expression E is satisfied on any trace on which E is falsified, which means that either (a) one of the events in the sequence does not occur, or (b) the order among events is violated.

Definition 38 $\mathsf{C}(e_1 \cdot \ldots \cdot e_n) \triangleq \bigvee_{1 \leq i \leq n} \overline{e_i} \vee \bigvee_{2 \leq i \leq n} (e_n \cdot e_{n-1})$.

Lemma 52 $\alpha(D) \equiv D \wedge \bigwedge_{w \in \Psi(D) \setminus \Phi(D)} \mathsf{C}(w)$. ∎

Example 26 Referring to Figure 11, we can determine that if $\Sigma_v = \{e\}$, then $\alpha(\overline{e} \vee f)$ is equivalent 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 $\overline{e} \cdot \overline{f} \vee f$, slightly stronger than the original dependency.

Similarly, referring to Figure 12, we can check that if $\Sigma_m = \{e\}$, then $\alpha(\overline{e} \vee \overline{f} \cdot e) = 0$. ∎

Scheduling with Event Attributes

Given a dependency $\alpha(D)$ that is modified because of the attributes, we can compute the guards on each event as above. Lemmas 50 and 51 mean that if $\alpha(D) \neq 0$, each inevitable event will remain possible along each path in $\alpha(D)$ and each immediate event will remain possible at each index of each path in $\alpha(D)$.

For inevitable events, no special treatment is necessary. If an inevitable event is attempted, it is made to wait until the opportune moment. When its guard becomes $\top$, it can happen. If $e \in \Sigma_v$ is attempted, $\Diamond e$ must hold on any generated trace, because only traces with e can be allowed. Thus, $\Diamond e$ is communicated to the relevant events so that spurious traces can be prevented.

The situation is more complex for immediate events. The guard on an immediate event may turn out not to be $\top$. Yet, such an event must be allowed immediately. This proves not to be a problem, however. Intuitively, our approach redundantly attaches information about event ordering on both events in any pair of events that are mutually ordered. We exploit this redundancy in $\mathsf{S}_{\neg}^{\triangle}$ by treating events that have not been heard from as not having occurred. This works because the event not heard from is made responsible for ensuring the mutual ordering. The above strategy is desirable because events can proceed in a more decoupled manner and send messages to each other only when necessary. For immediate events, however, the situation must be reversed. Since an immediate event f can and must happen without regard to its guard, it cannot be made responsible for ensuring its mutual ordering with any other event. Any other event e that relies on f's non-occurrence, must explicitly check if f has not yet occurred. If not, e can proceed; otherwise not. (If e is also immediate, $\alpha(D)$ must contain a path for each order of e and f.) S_m is the strategy that enables immediate events to happen immediately.

Definition 39 $\mathsf{S}_m(\mathcal{W}, e, u, i) \triangleq$ (if $e \in \Sigma_m$, then $\top$, else $\mathsf{S}_\neg^{\Sigma_m}(\mathcal{W}, e, u, i)$).

Theorem 53 Replacing $\mathsf{S}_\neg^\triangle$ by S_m does not violate correctness. ∎

The above simplifications lead to guard expressions that are easier to manipulate. However, the point is not to obtain the shortest guard representation, but to obtain the simplest representation with which we can reason symbolically. For instance, one might consider just using the past and doing away with the future information altogether. As can be readily checked, this would not work. If we use only the past information, i.e., knowledge of the events that precede the given event on different paths, we would obtain guards that involved only $\Box$ expressions. It would not be possible to execute such guards without making additional assumptions beyond the formal semantics. The present approach works because it unifies formal theory and the execution of guards.

6.5 Preprocessing and Execution

As we explained above, *generation* is an abstract process in that a trace may be generated based on the guards of the events on it evaluating to $\top$, even if they refer to the future. In actual execution, we do not have the luxury of looking into the future. For this reason, various heuristics are needed at execution to ensure that the necessary information flows, so that each event can locally decide to proceed or not. Sections 5.1 and 5.2.4 discussed some approaches through which the desired computations may be realized. We refer to these as *preprocessing*, because they involve setting up message flows according to different heuristics when guards are installed.

These heuristics set up the exchange of certain messages among events based on relationships among the events. These heuristics are obviously sound, because they merely call for the exchange of messages that contain true assertions. Thus these messages will cause no more traces to be generated than are generated by the previous evaluation strategies. Completeness, however, requires that if the guards are mutually satisfiable, then at least one event must have a guard of $\top$. We defer completeness issues to future research.

Heuristic 1 If e is triggerable, but not submitted, then execute e at any index where $\mathsf{G}(e) = \top$. In other words, a triggerable event may proactively be executed when it meets the usual requirements for execution, i.e., when its guard is $\top$. $\bullet$

Heuristic 2 If $\mathsf{G}(e) \neq \top$, $\mathsf{G}(e) \div \Box f = \top$, $\mathsf{G}(f) \neq \top$, $\mathsf{G}(f) \div \Diamond e = \top$, then e and f will send a promise to the other if they are the first to be attempted. $\bullet$

Heuristic 3 Let $\{e_0, \ldots, e_{n-1}\}$ be a set of events such that $(\forall i : 0 \leq i < n \Rightarrow \mathsf{G}(e_i) \div \Diamond(e_{i+1 \mathrm{mod} n}) = \top)$, then set up a promise from e_i to $e_{i+1 \mathrm{mod} n}$ when e_i is attempted and a promise has not already been received that updates its guard to $\top$. $\bullet$

Heuristic 4 If $\mathsf{G}(e) \neq \top$, $\mathsf{G}(e) \div \Box f = 0$, $\mathsf{G}(f) \neq \top$, $\mathsf{G}(f) \div \Box e = 0$, then e and f will prohibit the other before proceeding. $\bullet$

Theorem 54 Heuristics 1, 2, 3, and 4 are sound. ∎

Since we are operating at the level of workflows, we assume that an underlying message transport layer is available that guarantees message ordering. We also assume that the events are ordered at the desired granularity of time.

6.6 Examples

Following Example 3, we can compile and preprocess the guards on the different events in the workflow to achieve an effect similar to Example 6. In the following, we assume that the abort events, $\overline{c_{book}}$ and $\overline{c_{buy}}$, are immediate. All other events, in particular, the commit events, c_{book} and c_{buy}, are normal. The start events, s_{buy}, s_{book}, and s_{cancel}, are triggerable. For simplicity, Example 27 considers the workflow $\{D_1, D_2, D_3\}$; Example 28 adds back D_4.

Example 27 Because of the above attributes, dependency D_3 of Example 3 is effectively strengthened to $\alpha(D_3) = \overline{c_{book}} \vee s_{cancel} \vee c_{buy} \cdot c_{book} \cdot \overline{s_{cancel}} \vee c_{book} \cdot c_{buy} \cdot \overline{s_{cancel}} \vee c_{buy} \cdot \overline{s_{cancel}} \cdot c_{book} \vee \overline{s_{cancel}} \cdot c_{buy} \cdot c_{book}$. There is no change in D_1 or D_2.

We obtain the following guards from the workflow $\{D_1, D_2, \alpha(D_3)\}$:

- $\mathsf{G}(s_{buy}) = \Diamond s_{book}$
- $\mathsf{G}(\overline{s_{buy}}) = \top$
- $\mathsf{G}(s_{book}) = \top$
- $\mathsf{G}(\overline{s_{book}}) = \Diamond \overline{s_{buy}}$
- $\mathsf{G}(c_{book}) = \neg c_{buy} \wedge (\Diamond s_{cancel} \vee \Box c_{buy} \vee \neg \overline{s_{cancel}} \wedge \Diamond c_{buy})$

- $\mathsf{G}(\overline{c_{book}}) = \top$ (instead of $\Diamond\overline{c_{buy}}$)
- $\mathsf{G}(c_{buy}) = \Box c_{book} \wedge \neg\overline{s_{cancel}} \wedge \overline{s_{cancel}}$
- $\mathsf{G}(\overline{c_{buy}}) = \top$ (instead of $\Diamond\overline{c_{book}} \vee \Diamond s_{cancel}$)
- $\mathsf{G}(s_{cancel}) = \top$
- $\mathsf{G}(\overline{s_{cancel}}) = \Diamond\overline{c_{book}} \vee \Box c_{buy} \vee \neg c_{book} \wedge \Diamond c_{buy}$.

Following Example 6, we can enact this workflow as follows:

1. Suppose s_{buy} is attempted. It is accepted immediately, because its guard would be satisfied by a triggerable event with a true guard.
2. s_{book} is triggered.
3. Suppose c_{book} is attempted. It too is accepted because its guard would be satisfied by a triggerable event with a true guard. $\mathsf{G}(\overline{s_{cancel}}) \div \Box c_{book} = \Box c_{buy}$.
4. Suppose $\overline{c_{buy}}$ is attempted. It is too is accepted (being an immediate event). Since $\mathsf{G}(s_{cancel}) \div \Box\overline{c_{buy}} = top$, s_{cancel} must be triggered now.

In this way, a workflow may be enacted in a distributed manner. ▌

Example 28 In Example 27, s_{cancel} may be triggered unnecessarily, because its guard is $\top$. This means the specification allows superfluous events. To make the workflow more realistic, we add dependency D_4 back, which ensures that s_{cancel} executes only if necessary. Under the above

attributes, this strengthens to $\alpha(D_4) = \overline{s_{cancel}} \vee c_{book} \cdot s_{cancel} \cdot \overline{c_{buy}} \vee c_{book} \cdot \overline{c_{buy}} \cdot s_{cancel} \vee \overline{c_{buy}} \cdot c_{book} \cdot s_{cancel}$. We obtain different guards on the following events.

- $\mathsf{G}(c_{book}) = \neg c_{buy} \wedge (\Diamond s_{cancel} \vee \neg\overline{s_{cancel}} \wedge \Diamond c_{buy}) \wedge (\Diamond\overline{s_{cancel}} \vee \neg s_{cancel} \wedge \Diamond\overline{c_{buy}})$
- $\mathsf{G}(\overline{c_{book}}) = \top$ (instead of $\Diamond\overline{c_{buy}} \wedge \Diamond s_{cancel}$)
- $\mathsf{G}(c_{buy}) = \Box c_{book} \wedge \neg\overline{s_{cancel}} \wedge \Diamond s_{cancel}$
- $\mathsf{G}(\overline{c_{buy}}) = \top$ (instead of $(\Diamond\overline{c_{book}} \vee \Diamond s_{cancel}) \wedge (\Diamond\overline{s_{cancel}} \vee \Box c_{book} \wedge \Box s_{cancel})$)
- $\mathsf{G}(s_{cancel}) = \Box c_{book} \wedge \Diamond\overline{c_{buy}}$
- $\mathsf{G}(\overline{s_{cancel}}) = \Diamond\overline{c_{book}} \vee \Box c_{buy} \vee (\neg c_{book} \wedge \Diamond c_{buy})$.

This workflow proceeds similarly to Example 27. The main difference is in the guard of the triggerable s_{cancel} not being $\top$.

- When c_{book} is attempted, it is accepted, because if the immediate events referred to in its guard occurs, its guard reduces to $\Diamond s_{cancel}$, which can be triggered.
- When $\overline{c_{buy}}$ is attempted, it is accepted, because it is immediate.
- Now the guard on s_{cancel} is $\top$. Thus, s_{cancel} is triggered.

Thus the desired behavior is obtained. ▮

7 Enhancements

We now introduce some enhancements to the approach developed above.

7.1 Simple Real-Time Aspects

Our approach can accommodate real-time dependencies in a simple sense. We model clock values or timer interrupts simply as events that are immediate. Thus, we can strengthen the dependencies involving such events, and compile guards as above. Timer events affect the enforceability of dependencies in the same manner as other immediate events.

Example 29 Consider the dependency $D_8 = \overline{e} \vee \overline{f} \vee e \cdot f$. If f is a timer event, e.g., "today at noon," D_8 can be enforced only if e is not inevitable or immediate. Thus if e is attempted before noon, it is allowed, otherwise it is rejected.

If e is "today at noon," then D_8 can be enforced only if f is not immediate. If f is immediate and occurs before noon, D_8 would be violated. However, if f is an inevitable or normal event, then it can be delayed until after noon. ▮

In a more general sense, real-time scheduling involves reasoning about timer events before they occur, so as to allow enough time for various activities. We imagine that such reasoning might be performed while designing effective workflows. This is an important topic, but beyond the scope of this paper.

7.2 Dynamic Modification of Workflows

We assumed that a workflow is given at the outset, and said nothing about its being modified. However, workflows often need to be modified. In our approach, modifications can take the form of dependencies being added or removed, or attributes being modified. Checking if any dependency or attribute has been violated requires maintaining the entire history of the computation, and may not be practical.

No special treatment is necessary to remove dependencies, although it makes sense to remove their contributions to the guards of various events. This prevents the workflow from being over-constrained, leading to better schedules and also enabling additional dependencies to be added later. For this purpose, we can maintain the guard for each event as a conjunction, with each conjunct compiled from a different dependency. When a dependency is removed, the corresponding conjuncts are removed from each event mentioned in the dependency.

When a dependency D is added, D's contribution to the guard on each event is compiled as usual, but before it is assigned to the event, it is updated using the $\div$ operator. The final guard reflects all the events that have occurred or were promised along the computation so far, taking into account the ordering information.

Modifying event attributes is unintuitive, since it means a change in the inherent structure of the component tasks of a workflow, and should not be occurring while a workflow is in execution. However, adding or removing the attributes of normal, immediate, and inevitable translates into corresponding updates on the dependencies. Triggerable events can be triggered whenever their guard becomes true. If we care about the past, we must check not only the dependencies, but also whether an inevitable event has not already been rejected, an immediate event rejected or delayed, or a non-triggerable event triggered.

7.3 Parametrization

So far, we considered specific event instances. Following [Singh, 1996], we modify the syntax of $\mathcal{E}$ and $\mathcal{T}$ to interpret events as types, and parametrize them to create unique event instances. When dependencies are stated, some of the parameters can be variables, which are implicitly universally quantified. When events are scheduled, all parameters must be constants.

Common parameters include task ids, database keys, and other unique ids. Importantly, the event IDs need *not* depend on the structure of the associated task. Thus, our scheduler does not need to know the internal structure of a task agent. An agent may have arbitrary loops and branches and may exercise them in any order as required by the underlying task. Hence, we can handle arbitrary tasks correctly!

We replace Syntax 2 by the rules below. Here $\mathcal{V}$ is a set of variables and $\mathcal{C}$ a set of constants. Members of $\mathcal{V}$ and $\mathcal{C}$ can be used as parameters. Intuitively, Γ includes all (ground) event literals and Ξ includes all event atoms. Since the universe depends on Γ, it is automatically redefined to include all traces formed from all possible event instances. Let $\delta(e)$ give the *degree* of e, i.e., the number of parameters e needs to become an event instance.

Syntax 14 $e \in \Sigma$, $\delta(e) = m$, $p_1, \ldots, p_m \in \mathcal{C}$ implies $e[p_1 \ldots p_m], \overline{e}[p_1 \ldots p_m] \in \Gamma$

Syntax 15 $e \in \Sigma$, $\delta(e) = m$, $p_1, \ldots, p_m \in \mathcal{V} \cup \mathcal{C}$ implies $e[p_1 \ldots p_m], \overline{e}[p_1 \ldots p_m] \in \Xi$

Syntax 16 $\Xi \subseteq \mathcal{E}$

Semantics 1 continues to hold for members of Γ. We add Semantics 16 to interpret variables as universally quantified. Here $E(v)$ refers to an expression free in variable v (it may also be free in other variables). $E(v ::= c)$ refers to the expression obtained from $E(v)$ by substituting every occurrence of v by constant c.

Semantics 16 $u \models E(v)$ iff $(\forall c \in \mathcal{C} : u \models E(v ::= c))$

Events from the same task must have the same variable parameters. Further, because our focus is on intertask dependencies, different references to an event type involve the same tuple of parameters (i.e., variables and constants). This restriction forces all reasoning pertinent to individual tasks to be performed by the tasks or their agents. The scheduler handles only the intertask aspects. Thus the events and their parameters define a clean interface between the tasks and the scheduler.

We define residuation for parametrized dependencies. Here, $\vec{v}$ is a tuple of variables that parametrize the occurrences of e in E. This tuple must be unique within each dependency. Similarly, $\vec{c}$ is a tuple of constants with which the putative instance of e is instantiated.

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

We add a semantic rule for $\mathcal{T}$ as well:

Semantics 17 $u \models_{i,k} E(v)$ iff $(\forall c \in \mathcal{C} : u \models_{i,k} E(v ::= c))$

The guard definitions remain unchanged. The variables are instantiated to the appropriate constants when an event is attempted or triggered. We now consider two different ways of scheduling parametrized dependencies to handle scheduling of related and unrelated activities, using the same or different variable parameters, respectively. The updates on guards due to event occurrences of promises are still performed using the $\div$ operator, which is enhanced to accommodate parameters.

Definition 40 $G(\vec{v}) \div M(\vec{c}) \triangleq G(\vec{v}) \wedge G(\vec{v} ::= \vec{c}) \div M(\vec{c})$.

Parametrized Workflows

In the simplest case, parameters are used *within* a given workflow to relate events in different tasks. Typically, the same variables are used in parameters on events of different tasks. Attempting some key event binds the parameters of all events, thus instantiating the workflow afresh. The workflow is then scheduled as described in previous sections. We redo Example 3 below.

Example 30 Now we use t as the trip or reservation id to parametrize the workflow. The parameter t is bound when the *buy* task is begun. The explanations are as before—now we are explicit that the same customer features throughout the workflow. The resulting guards are as in Example 27, but with an explicit parameter, which is bound when the event is attempted or a message received from another event. Some of the guards are

- $\mathsf{G}(s_{buy}[t]) = \Diamond s_{book}[t]$
- $\mathsf{G}(\overline{s_{buy}}[t]) = \top$
- $\mathsf{G}(s_{book}[t]) = \top$
- $\mathsf{G}(\overline{s_{book}}[t]) = \Diamond \overline{s_{buy}}[t]$

Suppose $s_{buy}[33]$ is attempted. As before, it is accepted, and a notification of $\Box s_{buy}[33]$ is produced, which triggers $s_{book}[33]$, and so on. Thus the bound parameter is passed along as the workflow is enacted. ▌

Arbitrary Interacting Tasks

In the second class of problems, the different events may have unrelated variable parameters. Such cases occur in the specification of concurrency control requirements *across* workflows or transactions. Example 31 shows how mutual exclusion may be captured in our approach.

Example 31 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 captured by stating that if both b_1 and B_2 occur, then if b_1 precedes b_2, e_1 must also precede b_2. This holds for independent variable parameters x and y on T_1 and T_2, respectively. Thus, we have $D_{10}(x, y) = (b_2[y] \cdot b_1[x] \vee \overline{b_1[x]} \vee \overline{b_2[y]} \vee e_1[x] \cdot b_2[y])$ and $D_{11}(x, y) = (b_1[x] \cdot b_2[y] \vee \overline{b_1[x]} \vee \overline{b_2[y]} \vee e_2[y] \cdot b_1[x])$.

We observe that e_1 and e_2 are immediate events, since the tasks may unilaterally exit their critical sections. Interestingly, $\alpha(D_{10}) = D_{10}$ and $\alpha(D_{11}) = D_{11}$. Thus, from the workflow $\{D_{10}, D_{11}\}$, we obtain the following guards:

- $\mathsf{G}(b_1[x]) = (\Box b_2[y] \vee \Diamond\overline{b_2[y]} \vee \Diamond(e_1[x] \cdot b_2[y])) \wedge (\neg b_2[y] \vee \Box e_2[y])$
- $\mathsf{G}(e_1[x]) = \top$
- $\mathsf{G}(b_2[y]) = (\Box b_1[x] \vee \Diamond\overline{b_1[x]} \vee \Diamond(e_2[y] \cdot b_1[x])) \wedge (\neg b_1[x] \vee \Box e_1[x])$
- $\mathsf{G}(e_2[y]) = \top$

When $b_1[\hat{x}]$ is attempted for a constant $\hat{x}$, $\mathsf{G}(b_1[x])$ is instantiated to $\mathsf{G}(b_1[\hat{x}])$. Intuitively, $\mathsf{G}(b_1[\hat{x}])$ means that for all y (a) either $b_2[y]$ has not occurred or $e_2[y]$ has occurred, and (b) either $b_2[y]$ has occurred, $b_2[y]$ will not occurred, or $b_2[y]$ will occur after $e_1[\hat{x}]$. These are precisely the conditions under which $b_1[\hat{x}]$ can be allowed to proceed. The presence of the variable parameters leads to an interesting phenomenon during execution:

- Assume that initially, none of the events has occurred. When $b_1[\hat{x}]$ is attempted, it prohibits $b_2[y]$ for all y, and is accepted.
- $\mathsf{G}(b_2[y])$ is updated to $\mathsf{G}(b_2[y]) \wedge (\mathsf{G}(b_2[y]) \div \Box b_1[\hat{x}])$. Since $\mathsf{G}(b_2[y]) \div \Box b_1[\hat{x}] = \Box e_1[x]$, $\mathsf{G}(b_2[y])$ becomes $(\Box b_1[x] \vee \Diamond\overline{b_1[x]} \vee \Diamond(e_2[y] \cdot b_1[x])) \wedge (\neg b_1[x] \vee \Box e_1[x]) \wedge \Box e_1[\hat{x}]$.
- Suppose $b_2[\hat{y}]$ is attempted. Because its guard includes $\Box e_1[\hat{x}]$, it is made to wait.
- Later, when $e_2[\hat{x}]$ occurs, $\mathsf{G}(b_2[y])$ is updated to its original form. Now $b_2[\hat{y}]$ can be accepted, appropriately prohibiting $b_1[x]$ for all x.

In this manner, the guard expression can grow to accommodate the relevant instances, and later shrink when those instances are no longer relevant. ▌

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

8 Overview of the Literature

Several extended transaction models have been proposed [Barghouti & Kaiser, 1991; Kaiser, 1994; Elmagarmid, 1992]. These are typically nested but *open* in that results of subtransactions are visible before the global activity terminates. *Sagas* are composed of several subtransactions or *steps* [Garcia-Molina & Salem, 1987], which can be optimistically committed, thereby increasing concurrency but reducing isolation. If a step fails, consistency can be restored by compensating the previously committed steps in the reverse order. [Levy, 1991] formalizes some important restrictions on compensation. The *NT/PV* (nested transactions with predicates and views) model is based on multiple coexisting versions of a database [Korth & Speegle, 1994]. Transactions are modeled as binary relations on database states, rather than as functions. Specifications for transactions are given as precondition and postcondition pairs. Several transaction models can be expressed as special cases of the NT/PV model. [Kamath & Ramamritham, 1996] classify the correctness criteria that characterize several different transaction models.

ConTracts group a set of transactions into a multitransaction activity [Wächter & Reuter, 1992]. Each ConTract consists of (a) a set of steps: sequential ACID transactions that define the algorithmic aspects of the ConTract, and (b) a script or execution plan: a (possibly parallel) program invoking the steps that defines the structural aspects of the ConTract. ConTracts are forward-recoverable through failures and interruptions. A *long-running activity* is a set of transactions (possibly nested) and other activities [Dayal *et al.*, 1991]. Control and data flow may be specified in a script or with event-condition-action rules. Other important models include *flex transactions* [Bukhres *et al.*, 1993], *cooperating transactions* [Nodine, 1993], *split-and-join transactions* [Kaiser & Pu, 1992], and *cooperating activities* [Rusinkiewicz *et al.*, 1995].

There is also increasing interest in the organizational aspects of workflow management. [Joosten, 1994] proposes "trigger modeling" as a technique to capture some of the interrelationships among components of a workflow from an organizational perspective. This model comprises the concepts of activities, events, and actors, and relates them to each other. [Bußler & Jablonski, 1994] also relate workflows with organizational modeling. We believe the above approaches are at a higher level than our approach, which could provide a rigorous infrastructure in which to realize them.

Several execution environments have been proposed, which support specification and execution of transactions to varying degrees. An actor-based environment is developed in [Haghjoo *et al.*, 1993]. The DOM project includes a programmable environment (TSME) in which several transaction models can be specified [Georgakopoulos *et al.*, 1994]. However, TSME defines correctness criteria based on transaction histories, like in traditional approaches. The task specification languages for *interactions* [Nodine *et al.*, 1994] and METEOR [Krishnakumar & Sheth, 1994] are similar in intent. [Biliris *et al.*, 1994] have developed a programming facility for specifying transactions in the Ode environment. This facility borrows intuitions, but not the formalism, from ACTA (discussed below). The proclamations approach allows transactions to write (i.e., proclaim) sets of values prior to commitment [Jagadish

& Shmueli, 1992]. Transactions that read a proclaimed set carry out all possible computations and may generate further proclamations. Only single values are finally committed.

ACTA was the first serious attempt at formalizing the semantics of different extended transaction models [Chrysanthis & Ramamritham, 1994]. The notion of significant events of database transactions was introduced there, although the logics of programs community had developed similar ideas before. ACTA provides a history-based formalism for specifying intertask dependencies. It is similar in spirit to [Singh, 1996], although the latter also develops equations and model-theory for residuation, which characterizes the most general transitions in an abstract scheduler. The latter idea is important as a basis for the present paper. Intuitively, we can think of [Chrysanthis & Ramamritham, 1994], [Singh, 1996], and the present paper as forming a continuum analogous to the traditional one of logic, reasoning, and reasoning engine.

Active databases have a rule-triggering mechanism that can be used to maintain various constraints. Triggers can also be used to coordinate complex activities [Dayal *et al.*, 1990]. We too have developed and used a distributed expert system shell to encode workflows, including their contingency conditions [Singh & Huhns, 1994]. It is important to capture the contingencies declaratively, since they can otherwise get very complex.

Klein's approach, which is only sketchily described in [Klein, 1991a], 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 generically. Klein assumes that the structure of tasks can be expressed in a star-free regular expression—hence the restriction to loop-free events. Also, he does not allow prohibitory message flows, and must achieve their effect through complex promises, which seems unintuitive. Günthör's approach is based on temporal logic, but is centralized [1993]. These approaches are somewhat *ad hoc* in their details and do not properly handle complementary events. Also, they do not consider all the attribute combinations that we motivated above. Lastly, our previous approach, which constructs finite automata for dependencies, is centralized [Attie *et al.*, 1993]. It uses pathset search to avoid generating product automata, but the individual automata can be quite large. Neither of the above approaches can express or process complex dependencies as easily as the present approach.

In our system, events variously wait (at their actors), send messages to each other, and thereby enable or trigger each other. This appears intuitively similar to Petri nets in some respects. Petri nets are indeed an important formalism in which to express computations and can be applied to workflows, e.g., [van der Aalst, 1996]. However, our goal in this paper was to find a way to characterize workflows that may be weaker than general Petri nets, but which has just enough power to do what we need and is declaratively specified. Indeed, in a sense we "synthesize" Petri nets automatically by setting up the appropriate message flows. By symbolic reasoning during preprocessing, we also ensure that the "net" will operate correctly, e.g., by not deadlocking at mutual waits, but generating appropriate promissory messages instead.

It has been recently argued that various transaction models can be expressed in existing workflow products, and therefore existing workflow products are sufficient [Alonso *et al.*, 1996]. This argument is weak. Compilability into machine code does not make higher-level programming languages irrelevant! Extended transaction and workflow models provide a programming discipline through which computations can be structured. If they could not be translated to lower-level representations, they would not be useful! Although we did not propose any extended transaction or workflow models in this paper, we agree with [Wiederhold *et al.*, 1992] that higher-level abstractions of some form are necessary for programming in complex environments. The present work will facilitate the development of the appropriate abstractions. This paper discusses the underlying computations at the event-semantics level, which must be realized no matter what lies above.

9 Conclusions and Future Work

A prototype of our system has been implemented in an actor language. It is now being reimplemented in Java. 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 run-time. Although we begin with specifications that characterize entire traces as acceptable or unacceptable, we set up our computations so that information flows as soon as it is available, and activities are not unnecessarily delayed.

We showed how we can handle parametrized dependencies within our theory. Importantly, although individual workflows can be parametrized in a straightforward manner, interactions between independent workflows require greater subtlety. For these, it is essential to have a formal semantics. Although it took no significant effort in our approach to do so, this is conceptually an important step—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 logically would be a challenge for them.

In future work, we plan to explore connections with constraint languages so as to restrict the parameters in useful ways. Other potential extensions to the present work include research into the real-time aspects of workflow scheduling, improved heuristics for preprocessing, and a direct closed form representation for dependencies incorporating event attributes.

Our approach formalizes some of the reasoning required in scheduling workflows. It assumes intertask dependencies as given. An important problem that is beyond the scope of this paper is how may one actually come up with the necessary intertask dependencies to capture some desired workflowΓ This is the focus of a follow on research project we are now initiating.

Acknowledgments

Some of the preliminary work of this paper was performed in the Carnot Project at MCC from March 1993 to December 1994. The author participated in several discussions with Paul Attie, Greg Meredith, and Chris Tomlinson, especially in March 1993. The author benefited from the design and implementation of a previous approach [Attie *et al.*, 1993].

Many significant revisions and enhancements—including a new semantics, all proofs, most definitions, and most theorems—were made at NCSU in the period August 1995 through September 1996. The author has benefited from discussions with Neerja Bhatt, Anuj Jain, and Chubin Lin. This work has been partially supported by the NCSU College of Engineering and by the National Science Foundation under grant IRI-9529179 and IRI-9624425.

References

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

[Alonso *et al.*, 1996] Alonso, G.; Agrawal, D.; Abbadi, A. El; Kamath, M.; Günthör, R.; and Mohan, C.; 1996. Advanced transaction models in workflow contexts. In *International Conference on Data Engineering.*

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

[Barghouti & Kaiser, 1991] Barghouti, Naser S. and Kaiser, Gail E.; 1991. Concurrency control in advanced database applications. *ACM Computing Surveys* 23(3):269–317.

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

[Biliris *et al.*, 1994] Biliris, A.; Dar, S.; Gehani, N.; Jagadish, H. V.; and Ramamritham, K.; 1994. ASSET: A system for supporting extended transactions. In *Proceedings of the ACM SIGMOD Conference on Management of Data.*

[Bukhres *et al.*, 1993] Bukhres, Omran A.; Chen, Jiansan; Du, Weimin; Elmagarmid, Ahmed K.; and Pezzoli, Robert; 1993. InterBase: An execution environment for heterogeneous software systems. *IEEE Computer* 26(8):57–69.

[Bußler & Jablonski, 1994] Bußler, Christoph and Jablonski, Stefan; 1994. An approach to integrated workflow modeling and organization modeling in an enterprise. In *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises.* IEEE Computer Society Press.

[Chandy & Misra, 1986] Chandy, K. M. and Misra, Jayadev; 1986. How processes learn. *Distributed Computing* 1:40–52.

[Chrysanthis & Ramamritham, 1994] Chrysanthis, Panos K. and Ramamritham, Krithi; 1994. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems* 19(3):450–491.

[Dayal *et al.*, 1990] Dayal, U.; Hsu, M.; and Ladin, R.; 1990. Organizing long-running activities with triggers and transactions. In *Proceedings of ACM SIGMOD Conference on Management of Data.*

[Dayal *et al.*, 1991] Dayal, U.; Hsu, M.; and Ladin, R.; 1991. A transactional model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Data Bases.*

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

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

[Elmasri & Navathe, 1994] Elmasri, Ramez and Navathe, Shamkant; 1994. *Fundamental of Database Systems.* Benjamin Cummings Publishing Company, Redwood City, California, second edition.

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

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

[Georgakopoulos *et al.*, 1994] Georgakopoulos, Dimitrios; Hornick, Mark; Krychniak, Piotr; and Manola, Frank; 1994. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings of the International Conference on Data Engineering.*

[Georgakopoulos *et al.*, 1995] Georgakopoulos, Dimitrios; Hornick, Mark; and Sheth, Amit; 1995. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases* 3(2).

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

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

[Haghjoo *et al.*, 1993] Haghjoo, Mostafa S.; Papazoglou, Mike P.; and Schmidt, Heinz W.; 1993. A semantics-based nested transaction model for intelligent and cooperative information systems. In *International Conference on Intelligent and Cooperative Information Systems (CoopIS).*

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

[Jagadish & Shmueli, 1992] Jagadish, H. V. and Shmueli, Oded; 1992. A proclamation based model for cooperating transactions. In *Proceedings of the 18th International Conference on Very Large Data Bases.*

[Joosten, 1994] Joosten, Stef; 1994. Trigger modelling for workflow analysis. In *Proceedings of CON: Workflow Management.* R. Oldenbourg Verlag, Munich.

[Kaiser & Pu, 1992] Kaiser, Gail E. and Pu, Calton; 1992. Dynamic restructuring of transactions. In *[Elmagarmid, 1992].* Chapter 8.

[Kaiser, 1994] Kaiser, Gail E.; 1994. Cooperative transactions for multi-user environments. In *[Kim, 1994].* Chapter 20.

[Kamath & Ramamritham, 1996] Kamath, Mohan and Ramamritham, Krithi; 1996. Bridging the gap between transaction management and workflow management. In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions.* `http:// optimus.cs.uga.edu:5080/ activities/NSF-workflow/ kamath.html`.

[Kim, 1994] Kim, Won, editor. *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press (Addison-Wesley), New York, NY. Reprinted with corrections, 1995.

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

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

[Korth & Speegle, 1994] Korth, Henry F. and Speegle, Greg; 1994. Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *ACM Transactions on Database Systems* 19(3):492–535.

[Krishnakumar & Sheth, 1994] Krishnakumar, Narayanan and Sheth, Amit; 1994. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases.*

[Levy, 1991] Levy, Eliezer; 1991. *Semantics-Based Recovery in Transaction Management Systems.* Ph.D. Dissertation, Department of Computer Sciences, University of Texas, Austin, TX. Technical Report TR-91-29.

[Nodine *et al.*, 1994] Nodine, Marian H.; Nakos, Noela; and Zdonik, Stanley B.; 1994. Specifying flexible tasks in a multidatabase. In *Proceedings of the 2nd International Conference on Cooperative Information Systems (CoopIS).*

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

[Ramamritham & Chrysanthis, 1996] Ramamritham, Krithi and Chrysanthis, Panos K.; 1996. A taxonomy of correctness criteria in database applications. *The VLDB Journal* 5(1):85–97.

[Rusinkiewicz & Sheth, 1994] Rusinkiewicz, Marek and Sheth, Amit; 1994. Specification and execution of transactional workflows. In *[Kim, 1994].* Chapter 29.

[Rusinkiewicz *et al.*, 1995] Rusinkiewicz, Marek; Klas, Wolfgang; Tesch, Thomas; Wäsch, Jürgen; and Muth, Peter; 1995. Towards a cooperative transaction model — the cooperative activity model. In *Proceedings of the 19th International Conference on Very Large Data Bases.*

[Schwenkreis, 1996] Schwenkreis, Friedemann; 1996. Workflow for the German federal government – a position paper. In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions.* `http://optimus.cs.uga.edu:5080/activities/NSF-workflow/schwenk.html`.

[Sheth *et al.*, 1996] Sheth, Amit; Kochut, Krys; Miller, John; Worah, Devashish; Das, Souvik; Lin, Chenye; Palaniswami, Devanand; Lynch, John; and Shevchenko, Ivan; 1996. Supporting state-wide immunization tracking using multi-paradigm workflow technology. In *Proceedings of the 22nd VLDB Conference.*

[Singh & Huhns, 1994] Singh, Munindar P. and Huhns, Michael N.; 1994. Automating workflows for service provisioning: Integrating AI and database technologies. *IEEE Expert* 9(5). Special issue on *The Best of CAIA'94* with selected papers from Proceedings of the 10th IEEE Conference on Artificial Intelligence for Applications, March 1994.

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

[Singh, 1996] Singh, Munindar P.; 1996. Formal semantics for workflow computations. Technical Report TR-96-08, Department of Computer Science, North Carolina State University, Raleigh, NC. Extended version of [Singh, 1995]. Also available at `http://www4.ncsu.edu/eos/info/ dblab/www/ mpsingh/ papers/ databases/ semantics-jan96.ps`.

[Tomlinson *et al.*, 1993] Tomlinson, Christine; Cannata, Philip E.; Meredith, Greg; and Woelk, Darrell; 1993. The extensible services switch in Carnot. *IEEE Parallel and Distributed Technology* 16–20.

[van der Aalst, 1996] Aalst, W. M. P.van der; 1996. Petri-net-based workflow management software. In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions.* `http:// optimus. cs.uga.edu:5080/ activities/NSF-workflow/ wfm.html`.

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

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

[Wiederhold *et al.*, 1992] Wiederhold, Gio; Wegner, Peter; and Ceri, Stefano; 1992. Toward megaprogramming. *Communications of the ACM* 35(11):89–99.

A Proofs of Important Results

Lemma 27

If $u \models_{k-1} \mathsf{G}_b(w, u_k)$, then $u \sqsupseteq w$.

Proof.

Let $u_k = w_l$ (for otherwise, $\mathsf{G}_b(w, u_k) = 0$). Then, $pre(u, u_k) \Rightarrow pre(w, u_k)$ and $post(u, u_k) \Rightarrow post(w, u_k)$. Therefore, $\langle u_1, \ldots, u_{k-1}\rangle \sqsupseteq \langle v_1, \ldots, v_{l-1}\rangle$ and $\langle u_{k+1}, \ldots\rangle \sqsupseteq \langle v_{l+1}, \ldots\rangle$. Hence, $u \sqsupseteq w$.

Theorem 28

Let $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$. Then, $(\forall D \in \mathcal{W} : u \models D)$.

Proof.

Consider any dependency $E \in \mathcal{W}$. By Observation 26, $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(E, u_j))$.

By Observation 23, $u \models_0 \mathsf{G}_b(E, u_1)$ implies that $(\exists w : w \in \Pi(E)$ and $u \models_0 \mathsf{G}_b(w, u_1))$. By Lemma 27, $u \sqsupseteq w$. By Observation 18, $u \in \Pi(E)$. Therefore, by Lemma 17, $u \models E$. This holds for all dependencies in $\mathcal{W}$.

Theorem 30

Let $(\forall D \in \mathcal{W} : u \models D)$. Then, $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$.

Proof.

Consider a dependency $D \in \mathcal{W}$. Since $u \models D$ and $\Gamma_u \supseteq \Gamma_D$ (because u is maximal), $u \in \Pi(D)$. Thus, $(\forall j : 1 \leq j \leq |u| \Rightarrow (\mathsf{G}_b(u, u_j) \Rightarrow \mathsf{G}_b(D, u_j)))$. Using Observation 29, we have $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(D, u_j))$. Hence, by Observation 26, $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$.

Lemma 32

$(\exists k \leq j : u \models_k M$ 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.

Theorem 33

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

Proof.

From Definition 23, it is possible to have a trace 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 traces that $\mathsf{S}_b(\mathcal{W})$ generates. Thus completeness is established.

Let $\mathsf{S}_\div(\mathcal{W}) \rightsquigarrow_i u$. Then, $(\forall j : 1 \leq j \leq i \Rightarrow u \models_{j-1} \mathsf{S}_\div(\mathcal{W}, u_j))$. We prove by mathematical induction that $(\forall j : 1 \leq j \leq i \Rightarrow u \models_{j-1} \mathsf{S}_b(\mathcal{W}, u_j))$. Since $\mathsf{S}_\div(\mathcal{W}, u_1, u, 0) = \mathsf{S}_b(\mathcal{W}, u_1, u, 0)$, we have $u \models_0 \mathsf{S}_b(\mathcal{W}, u_1)$. Thus, $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow_1 u$.

Assume that the inductive hypothesis holds for $1 \leq l \leq i$, i.e., $(\forall j : 1 \leq j \leq l \Rightarrow u \models_{j-1} \mathsf{S}_b(\mathcal{W}, u_j))$. We show that $u \models_l \mathsf{S}_b(\mathcal{W}, u_{l+1})$. $\mathsf{S}_\div(\mathcal{W}) \rightsquigarrow_{l+1} u$ holds only if $u \models_l \mathsf{S}_\div(\mathcal{W}, u_{l+1}, u, l+1)$.

If $\mathsf{S}_\div(\mathcal{W}, u_{l+1}, u, l{+}1) = \mathsf{S}_\div(\mathcal{W}, u_{l+1}, u, l)$, then $\mathsf{S}_\div(\mathcal{W}, u_{l+1}, u, l{+}1) = \mathsf{S}_b(\mathcal{W}, u_{l+1}, u, l) = \mathsf{S}_b(\mathcal{W}, u_{l+1}, u, l+1)$. Therefore, $u \models_l \mathsf{S}_b(\mathcal{W}, u_{l+1}, u, l+1)$, which (since $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow_l u$) holds iff $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow_{l+1} u$, as desired.

If $\mathsf{S}_\div(\mathcal{W}, u_{l+1}, u, l{+}1) \neq \mathsf{S}_\div(\mathcal{W}, u_{l+1}, u, l)$, then $(\exists k \leq l : u \models_k M$ and $\mathsf{S}_\div(\mathcal{W}, e, u, l{+}1) = (\mathsf{S}_\div(\mathcal{W}, e, u, l) \div M))$. By Lemma 32, $(\exists k \leq l : u \models_k M$ and $u \models_l \mathsf{S}_\div(\mathcal{W}, e, u, l) \div M)$ implies that $u \models_l \mathsf{S}_\div(\mathcal{W}, e, u, l)$. Thus, $u \models_l \mathsf{S}_b(\mathcal{W}, u_{l+1}, u, l+1)$, which holds iff $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow_{l+1} u$, as desired.

Theorem 36

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

Proof.

$\mathsf{G}_b(D \vee E, e) = \bigvee_{w \in \Pi(D \vee E)} \mathsf{G}_b(w, e)$. Since $\Pi(D \vee E) \subseteq \Pi(D)$ and $\Pi(D \vee E) \subseteq \Pi(E)$, $\mathsf{G}_b(D \vee E, e) \Rightarrow \bigvee_{w \in \Pi(D)} \mathsf{G}_b(w, e) \vee \bigvee_{w \in \Pi(E)} \mathsf{G}_b(w, e)$, which equals $\mathsf{G}_b(D, e) \vee \mathsf{G}_b(E, e)$.

In the opposite direction, let $w \in \Pi(D)$. w contributes to $\mathsf{G}_b(w, e)$ only if e occurs in w. Instantiate Definition 24 as $I(w, \Gamma_{D \vee E} \setminus \Gamma_w)$, which contains all interleavings of w with events in $\Gamma_{D \vee E}$ that aren't in Γ_w. By Observation 18, $I(w) \subseteq \Pi(D)$. Also, $I(w) \subseteq \Pi(D \vee E)$. By Lemma 34, $\mathsf{G}_b(w, e) = \bigvee_{v \in I(w, \Gamma_{D \vee E} \setminus \Gamma_w)} \mathsf{G}_b(v, e)$. Thus the contribution of w to $\mathsf{G}_b(w, e)$ is covered by paths in $\Pi(D \vee E)$.

Theorem 38

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

Proof.

$\mathsf{G}_b(D \wedge E, e) = \bigvee_{w \in \Pi(D \wedge E)} \mathsf{G}_b(w, e)$. By Observation 19, $\Pi(D \wedge E) = \Pi(D) \cap \Pi(E)$. Therefore, $\mathsf{G}_b(D \wedge E, e) \Rightarrow \bigvee_{w \in \Pi(D)} \mathsf{G}_b(w, e) \wedge \bigvee_{w \in \Pi(E)} \mathsf{G}_b(w, e)$, which equals $\mathsf{G}_b(D, e) \wedge \mathsf{G}_b(E, e)$.

In the opposite direction, consider the contribution of a pair $v \in \Pi(D)$ 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. Let $e = v_i = w_j$. Then $\mathsf{G}_b(v,e) = \Box(e_1 \cdot \ldots \cdot e_{i-1}) \wedge \neg e_{i+1} \wedge \ldots \wedge \neg e_{|v|} \wedge \Diamond(e_{i+1} \cdot \ldots \cdot e_{|v|})$ and $\mathsf{G}_b(w,e) = \Box(e_1 \cdot \ldots \cdot e_{j-1}) \wedge \neg e_{j+1} \wedge \ldots \wedge \neg e_{|w|} \wedge \Diamond(e_{j+1} \cdot \ldots \cdot e_{|w|})$.

$\mathsf{G}_b(v,e) \wedge \mathsf{G}_b(w,e) \neq 0$ implies that there is a path x, such that $x \sqsupseteq v$ and $x \sqsupseteq v$ and $\mathsf{G}_b(x,e) = \mathsf{G}_b(v,e) \wedge \mathsf{G}_b(w,e)$. By Observation 18, $x \in \Pi(D) \cap \Pi(E)$. By Observation 19, $x \in \Pi(D \wedge E)$. Thus, $x \in \Pi(D) \cap \Pi(E)$. Hence, any contribution $\mathsf{G}_b(v,e) \wedge \mathsf{G}_b(w,e)$ to $\mathsf{G}_b(D,e) \wedge \mathsf{G}_b(E,e)$ due to paths in $\Pi(D)$ and $\Pi(E)$ is matched by a contribution by a path in $\Pi(D \wedge E)$. Therefore, $\mathsf{G}_b(D,e) \wedge \mathsf{G}_b(E,e) \Rightarrow \mathsf{G}_b(D \wedge E, e)$.

Lemma 42

$\mathsf{G}_b(e_1 \cdot \ldots \cdot e_n, e) = \Diamond(e_1 \cdot \ldots \cdot e_n)$, if $e \not\in \Gamma_{e_1 \cdot \ldots \cdot e_n}$.

Proof.

Let $D^e = (e_1 \cdot \ldots \cdot e_n) \wedge e$. By Lemmas 40 and 41, $\mathsf{G}_b(D,e) = \bigvee_{w \in \Psi(D^e)} \mathsf{G}_b(w,e)$. Let $w = \langle e_1, \ldots, e_k, e, e_{k+1}, \ldots, e_n \rangle \in \Psi(D^e)$. Then $\mathsf{G}_b(w,e) = \Box(e_1 \cdot \cdots \cdot \cdots \cdot e_k) \wedge \neg e_{k+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond(e_{k+1} \cdot \ldots \cdot e_n)$. There is one such w for each position of e, i.e., for $0 \leq k \leq n$. Thus, $\mathsf{G}_b(D,e) = \bigvee_{0 \leq k \leq n} \Box(e_1 \cdot \cdots \cdot \cdots \cdot e_k) \wedge \neg e_{k+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond(e_{k+1} \cdot \ldots \cdot e_n)$. It is easy to verify that for any trace u and index i, $u \models_i \mathsf{G}_b(D,e)$ iff $u \models_i \Diamond D$.

Theorem 43

$\mathsf{G}_b(D,e) = \Diamond D$, if $e \not\in \Gamma_D$.

Proof.

The proof is by induction on the structure of dependencies. For the base case, consider 0, $\top$, and event f, where $f \neq e$. For the inductive step, consider dependencies of the form $e_1 \cdot \ldots \cdot e_n$. By Lemma 42, $\mathsf{G}_b(D,e) = \Diamond D$. Consider dependencies of the form $D_1 \vee D_2$. By Observation 11, $\Diamond(D_1 \vee D_2) \cong \Diamond D_1 \vee \Diamond D_2$. For dependencies of the form $D_1 \wedge D_2$, use Observation 14 and the fact that $\mathcal{E} \subseteq \mathcal{P}$ to obtain $\Diamond(D_1 \wedge D_2) \cong \Diamond D_1 \wedge \Diamond D_2$.

Theorem 44

For all Δ, replacing $\mathsf{S}_{\div}(\mathcal{W})$ by $\mathsf{S}_p^{\Delta}(\mathcal{W})$ preserves correctness.

Proof.

Because $\mathsf{S}_p^{\Delta}(\mathcal{W}) \Rightarrow \mathsf{S}_{\div}(\mathcal{W})$ (i.e., S_p^{Δ} weakens the guards), it does not prevent any traces that would have been generated. Thus it preserves completeness.

Let $\mathsf{S}_p^{\Delta}(\mathcal{W}) \rightsquigarrow u$, such that $\mathsf{S}_{\div}(\mathcal{W}) \not\rightsquigarrow u$. Clearly, $\mathsf{S}_{\div}(\mathcal{W}) \rightsquigarrow_1 u$. Let $D \in \mathcal{W}$. Thus, there exists $w \in \Pi(D)$, such that $u \models_0 \mathsf{G}_b(w, u_1)$. By Lemma 27, $u \sqsupseteq w$. Thus, $u \models D$. Thus no additional spurious traces are allowed by $\mathsf{S}_p^{\Delta}(\mathcal{W})$.

Lemma 45

$\mathsf{S}_{\div}(w) \rightsquigarrow u$ iff $\mathsf{S}^{\Delta}_{\wedge}(w) \rightsquigarrow u$.

Proof.

Since $\mathsf{S}^{\Delta}_{\wedge}(w)$ is weaker than $\mathsf{S}_{\div}(w)$, $\mathsf{S}_{\div}(w) \rightsquigarrow u$ implies $\mathsf{S}^{\Delta}_{\wedge}(w) \rightsquigarrow u$.

Assume that $\mathsf{S}^{\Delta}_{\wedge}(w) \rightsquigarrow u$. Thus, $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}^{\Delta}_{\wedge}(w, u_j))$. Our proof uses mathematical induction. We first establish that $\mathsf{S}_{\div}(w) \rightsquigarrow_1 u$. We claim that $u \models_0 \mathsf{G}^{\Delta}_{b}(w, w_1)$. Consider the terms in $\mathsf{G}^{\Delta}_{b}(w, w_1)$ one by one.

- There are no $\Box$ terms in $\mathsf{G}^{\Delta}_{b}(w, w_1)$. Therefore, they are trivially satisfied by any trace at index 0.

- All $\neg$ terms are trivially satisfied on any trace at index 0.

- Suppose $u \not\models_0 \Diamond(w_2 \cdot \ldots \cdot w_{|w|})$. However, since $\mathsf{S}^{\Delta}_{\wedge}(w) \rightsquigarrow u$, $u \models_0 (\Diamond w_2 \wedge \ldots \wedge \Diamond w_{|w|})$. Thus, all the events on w occur on u. Let i be least index such that $u \models_0 \Diamond(w_2 \cdot \ldots \cdot w_i)$ and $u \not\models_0 \Diamond(w_2 \cdot \ldots \cdot w_{i+1})$. Since $u \models_0 \Diamond w_2$, $i \geq 2$. We know that $\mathsf{G}^{\Delta}_{\wedge}(w, w_{i+1})$ includes a term $\Box w_i$. Let $u_j = w_i$ and $u_k = w_{i+1}$. Then $k < j$, because otherwise, $u \models_0 \Diamond(w_2 \cdot \ldots \cdot w_i) \Rightarrow u \models_0 \Diamond(w_2 \cdot \ldots \cdot w_{i+1})$. However, $u \not\models_{k-1} \Box w_i$. Hence, $u \not\models_{k-1} \mathsf{G}^{\Delta}_{\wedge}(w, w_{i+1})$. Thus, $\mathsf{S}^{\Delta}_{\wedge}(w) \not\rightsquigarrow u$, which is a contradiction.

Thus, $u \models_0 \mathsf{G}^{\Delta}_{b}(w, w_1)$ or $\mathsf{S}_{\div}(w) \rightsquigarrow_1 u$.

Assume $\mathsf{S}_{\div}(w) \rightsquigarrow_k u$ and $\mathsf{S}_{\div}(w) \not\rightsquigarrow_{k+1} u$. Then, $u \not\models_k \mathsf{S}_{\div}(w, u_{k+1})$. There are three cases:

- $u \not\models_k \Box(w_1 \cdot \ldots \cdot w_{k-1} \cdot w_k)$, but $u \models_{k-1} \Box(w_1 \cdot \ldots \cdot w_{k-1})$ and $u \models_k \Box w_1 \wedge \ldots \wedge \Box w_k$. This is impossible.

- $u \not\models_k \neg w_{k+i}$, for some $i > 0$. This is impossible because $w_{k+i} \not\in \{w_1, \ldots, w_k\}$.

- $u \not\models_k \Diamond(w_{k+2} \cdot \ldots \cdot w_{|w|})$. This is impossible because $u \models_0 \Diamond(w_2 \cdot \ldots \cdot w_{|w|})$ (as established above).

Thus, $\mathsf{S}_{\div}(w) \rightsquigarrow_{k+1} u$ follows from $\mathsf{S}_{\div}(w) \rightsquigarrow_k u$. Consequently, $\mathsf{S}_{\div}(w) \rightsquigarrow u$.

Lemma 46

Replacing $\mathsf{S}_{\div}(\mathcal{W})$ by $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W})$ does not preserve correctness.

Proof.

Consider a dependency $D = (e \cdot f \cdot g \cdot h) \vee (f \cdot e \cdot h \cdot g)$. $\mathsf{S}^{\Delta}_{\wedge}(D) \rightsquigarrow \langle fegh \rangle$, which is not in $[\![D]\!]$. $\mathsf{S}_{\div}(D) \not\rightsquigarrow \langle fegh \rangle$.

Lemma 47

If $\mathcal{W}$ is a 3-workflow, then replacing $\mathsf{S}_{\div}(\mathcal{W})$ by $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W})$ preserves correctness.

Proof.

If trace u containing a subsequence $\langle efg \rangle$ is generated (with $f = u_k$), then there must be a path w, such that $u \models_{k-1} \mathsf{G}^{\Delta}_{\wedge}(w, f)$. This entails that if either e and g occur on w, they are in the correct order with respect to f. Thus $\langle efg \rangle$ does not represent a violation of any dependency in $\mathcal{W}$.

Theorem 48

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

Proof.

Since $\mathsf{S}^{\Delta}_{\top}(\mathcal{W})$ is weaker than $\mathsf{S}^{\Delta}_{p}(\mathcal{W})$, completeness is preserved.

Consider $D \in \mathcal{W}$ and $e \notin \Gamma_D$. Let $f \in \Gamma_D$. By Definition 32 and Lemma 40, $\mathsf{G}^{\Delta}_{\top}(D, f) = \bigvee_{w \in \Psi(D)} \mathsf{G}^{\Delta}_{p}(w, f)$. Consequently, e and $\overline{e}$ do not occur in $\mathsf{G}_p(D, f)$. Thus the occurrence or non-occurrence of e or $\overline{e}$ has no effect upon f.

Let $\mathsf{S}^{\Delta}_{\top}(\mathcal{W}) \leadsto u$. If $u \not\models_j \mathsf{S}^{\Delta}_{p}(D, u_{j+1})$ and $u \models_j \mathsf{S}^{\Delta}_{\top}(D, u_{j+1})$, then $u_{j+1} \notin \Gamma_D$. Let $B(u) = \{u_i : u \not\models_{i-1} \mathsf{S}^{\Delta}_{p}(D, u_i)\}$. 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 $u_k = v_l \Rightarrow u \models_{k-1} \mathsf{G}^{\Delta}_{p}(D, u_k)$ iff $v \models_{l-1} \mathsf{G}^{\Delta}_{p}(D, v_l))$. Hence, $\mathsf{S}^{\Delta}_{p}(\mathcal{W}) \leadsto v$. By Theorem 44, $v \models D$. By Observation 18, $u \models D$.

Theorem 49

For all Δ, replacing $\mathsf{S}^{\Delta}_{\top}$ by $\mathsf{S}^{\Delta}_{\neg}$ does not violate correctness.

Proof.

Because $\mathsf{S}^{\Delta}_{\neg}(\mathcal{W}) \Rightarrow \mathsf{S}^{\Delta}_{\top}(\mathcal{W})$ (i.e., $\mathsf{S}^{\Delta}_{\neg}$ weakens the guards), it does not prevent any traces that would have been generated. Thus it preserves completeness.

For soundness, our proof obligation is $\mathsf{S}^{\Delta}_{\neg} \leadsto u \Rightarrow \mathsf{S}^{\Delta}_{\top} \leadsto u$. We establish this by induction. Clearly, $\mathsf{S}^{\Delta}_{\neg} \leadsto_1 u \Rightarrow \mathsf{S}^{\Delta}_{\top} \leadsto_1 u$. Let $D \in \mathcal{W}$. By the inductive hypothesis, assume $\mathsf{S}^{\Delta}_{\neg} \leadsto_k u$ and $\mathsf{S}^{\Delta}_{\top} \leadsto_k u$ and $u \not\models_k \mathsf{S}^{\Delta}_{\top}(\mathcal{W}, u_{k+1}))$.

Assume $\mathsf{S}^{\Delta}_{\neg} \leadsto_{k+1} u$. Therefore, $(\exists w : w \in \Pi(D)$ and $u \models_k \mathsf{S}^{\Delta}_{\neg}(w, u_{k+1})$ and $u \not\models_k \mathsf{S}^{\Delta}_{\top}(w, u_{k+1}))$. If u_{k+1} does not occur on w, then $\mathsf{S}^{\Delta}_{\neg}(w, u_{k+1}) = 0$, i.e., $u \not\models_k \mathsf{S}^{\Delta}_{\neg}(w, u_{k+1})$. Let $u_{k+1} = w_l$. Then, there must be a $\neg f$ or $\Box f$ term in $\mathsf{S}^{\Delta}_{\top}(w, u_{k+1})$, such that $\neg f$ is spuriously evaluated as $\top$ (or $\Box f$ as 0) in $\mathsf{S}^{\Delta}_{\neg}(w, u_{k+1})$. Let $f = u_p$, where $1 \leq p \leq k$. Since, $u \models_{p-1} \mathsf{G}^{\Delta}_{\top}(D, u_p)$, there is a $v \in \Pi(D)$, such that $u \models_{p-1} \mathsf{G}^{\Delta}_{\top}(v, u_p)$. Let $u_p = v_m$. Then, $\{v_1, \ldots, v_{m-1}\} \subseteq \{u_1, \ldots, u_{p-1}\}$ and $\langle u_{p+1}, \ldots \rangle \sqsupseteq \langle v_{m+1}, \ldots \rangle$. Consequently, u_{k+1} occurs on v and $u \models_k \mathsf{G}^{\Delta}_{\top}(v, u_{k+1})$. Thus, $\mathsf{S}^{\Delta}_{\top} \leadsto_{k+1} u$. Therefore, $\mathsf{S}^{\Delta}_{\neg} \leadsto u$ iff $\mathsf{S}^{\Delta}_{\top} \leadsto u$.

Theorem 53

Replacing $\mathsf{S}_{\neg}^{\Delta}$ by S_m does not violate correctness.

Proof.

$\mathsf{S}_{\neg}^{\Delta}(\mathcal{W})$ is correct for all sets of events Δ. Because $\mathsf{S}_m(\mathcal{W})$ only weakens the guards for some events, it does not prevent any traces that would have been generated. Thus it preserves completeness.

Let $\mathsf{S}_m(\mathcal{W}) \rightsquigarrow u$. Let $u_j \in \Sigma_m$. Consider $D \in \mathcal{W}$. If all u_i belong to Σ_m, then by Lemma 50 $u \in [\![D]\!]$. Let $u_k \in \Gamma_D \setminus \Sigma_m$. Clearly, $u \models_{k-1} \mathsf{G}_p^{\Sigma_m}(D, u_k)$. This means there is a $v \in \Pi(D)$, such that $u \models_{k-1} \mathsf{G}(v, u_k)$. By Lemma 27, $u \sqsupseteq v$. Thus, $u \models D$. Hence, $\mathsf{S}_{\neg}^{\Delta}(\mathcal{W}) \rightsquigarrow u$.

Consequently, $\mathsf{S}_m(\mathcal{W}) \rightsquigarrow u$ iff $\mathsf{S}_{\neg}^{\Sigma_m}(\mathcal{W}) \rightsquigarrow u$.