Formal Aspects of Workflow Management
Part 2: Distributed Scheduling[*][†]

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

singh@ncsu.edu

June 18, 1997

Abstract

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

We address the problem of scheduling workflows from declarative specifications given in terms of intertask dependencies and event attributes. Our approach 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. It has been implemented.

workflows, temporal logic, scheduling.

[*]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.

[†]Some of the preliminary work of this paper was performed in the Carnot Project at MCC. I participated in discussions with Paul Attie, Greg Meredith, and Chris Tomlinson. Many significant revisions and enhancements—including a new formal semantics, all proofs, most definitions, and most theorems—were made at NCSU. I have benefited from discussions with Neerja Bhatt, Anuj Jain, and Chubin Lin, and comments from Krithi Ramamritham.

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

1 Introduction

Part 1 [15] motivated a language for specifying workflows, gave a formal semantics for it, and showed how to reason about schedules through an algebraic operator. Here we carry that approach further to obtain a generic distributed scheduler. Scheduling a workflow means coordinating its constituent tasks by causing, delaying, or preventing "events" based on the events of other tasks, and the state of the environment. Workflow scheduling is challenging because the tasks retain some autonomy in execution and design, and subtle correctness and organizational requirements are involved. Distributed scheduling is desirable, because workflows arise in distributed, heterogeneous, open environments, whereas centralized approaches are not scalable.

We follow the execution model of [15, section 2], which postulates a task agent for each task, each agent representing the events significant for coordination. We now show how that model can be instantiated in a distributed manner, and how the executions of the different agents can be orchestrated based on the stated specifications. This involves compiling the declarative specifications into representations that can be executed eagerly by each event with few unnecessary interactions among events.

Architecture and Operation

The object for an event maintains its current *guard* and manages communications. The guard is a temporal logic expression defining 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. If the guard is true, the event occurs; if false, it is rejected; else, it waits. When an event happens, messages announcing its occurrence are sent to actors of other relevant events. Upon receiving an event announcement, an actor simplifies its guard to incorporate this information. If the receiving actor's guard becomes true or false, then a decision is made on any waiting event.

Additional complexity is introduced in some situations. One, the events may have mutual constraints, which naively may lead to deadlock. Two, the events may have to be proactively triggered, or may be allowed to occur immediately. Three, some events may be instantiated multiple times—all and only the right instances must occur. Our approach has the following main steps:

1. dependencies are asserted, specifying a workflow
2. events are created (as objects)
3. guards are installed (based on the dependencies)
4. guards are preprocessed to detect mutual constraints, and messages are set up to achieve correct behavior
5. events are activated, thus enacting the workflow.

The designer does step 1. The system does steps 2, 3, and 4 at compile-time, and step 5 at run-time.

Contributions

We develop a distributed approach for workflow scheduling. Our approach includes a temporal language, which can express the knowledge necessary to execute events eagerly. This language requires a different formal semantics than previous temporal logics. We show how guards are compiled from workflow specifications. Our formal definition yields technical results about correctness and independence properties that facilitate compilation of the guards and enable concurrent execution. Our approach is formal, and handles ongoing or iterative activities in a simple manner.

Organization

Section 2 introduces our temporal logic, and shows how guards can be compiled. Section 3 shows how workflows can be executed in a distributed manner that respects the "attributes" of the events. Section 4 establishes correctness and justifies efficiency-enhancing transformations. Section 5 discusses some enhancements, including how to handle arbitrary tasks. Section 6 reviews the relevant literature. Appendix A gives the proofs.

2 Guards on Events

A naive implementation of [15] would represent all dependencies in one place, but suffer from the problems of centralization. Distribution requires that information be placed locally on each event. Our, or any, implementation requires (a) determining the conditions, i.e., *guards*, on the events by which decisions can be taken on their occurrence, (b) arranging for the relevant information to flow from one event to another, and (c) providing an algorithm by which the different messages can be assimilated.

2.1 Temporal Logic

Intuitively, the guard of an event is the weakest condition that guarantees correctness if the event occurs. Guards must be temporal expressions 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. Typically, guards are succinct.

$\mathcal{T}$ is the formal language in which the guards are expressed. This language captures the above distinctions. Syntax rules 1, 2, and 3 are exactly as for the dependency language $\mathcal{E}$ [15]. Rule 4 adds the new operators. 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. For simplicity, we assume the following binding precedence (in decreasing order): $\neg$; $\cdot$; $\Box$ and $\Diamond$; $\wedge$; $\vee$.

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

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

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

Syntax 4 $E \in \mathcal{T}$ implies $\Box E$, $\Diamond E$, $\neg E \in \mathcal{T}$

The semantics of $\mathcal{T}$ is given with respect to a trace (as for $\mathcal{E}$) *and* an index into that trace. Whereas the semantics of $\mathcal{E}$ distinguishes good traces from bad traces—precisely what a specifier cares about, the semantics of $\mathcal{T}$ characterizes the progress along a given trace—precisely what is needed to determine the scheduler's action at each event. The semantics of $\mathcal{T}$ has important differences from common linear temporal logics [5]. 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, i.e., intervals, rather than a single index. For $0 \leq i \leq k$, $u \models_{i,k} E$ means that E is satisfied over the subsequence of u between i and k. For $k \geq 0$, $u \models_k E$ means that E is satisfied on u at index k—implicitly, i is set to 0. $\Lambda \triangleq \langle \rangle$ is the empty trace.

Definition 1 A trace, u, is *maximal* iff for each event, either the event or its complement occurs on u.

Definition 2 $\mathbf{U}_{\mathcal{T}} \triangleq$ the set of maximal traces.

We assume $\Sigma \neq \emptyset$; hence, , $\neq \emptyset$. Semantics 1, which involves just one index i, invokes the semantics with the entire trace until i. The second index is interpreted as the present moment. Semantics 3, 4, 6, and 7 are as in traditional formal semantics. Semantics 8 and 9 involve looking into the future. Semantics 2 and 5 capture the dependence of an expression on the immediate past, bounded by the first index of the semantic definition. Semantics 5 introduces a nonzero first index.

Semantics 1 $u \models_i E$ iff $u \models_{0,i} E$

Semantics 2 $u \models_{i,k} f$ iff $(\exists j : i \leq j \leq k$ and $u_j = f)$, where $f \in$,

Semantics 3 $u \models_{i,k} E_1 \vee E_2$ iff $u \models_{i,k} E_1$ or $u \models_{i,k} E_2$

Semantics 4 $u \models_{i,k} E_1 \wedge E_2$ iff $u \models_{i,k} E_1$ and $u \models_{i,k} E_2$

Semantics 5 $u \models_{i,k} E_1 \cdot E_2$ iff $(\exists j : i \leq j \leq k$ and $u \models_{i,j} E_1$ and $u \models_{j+1,k} E_2)$

Semantics 6 $u \models_{i,k} \top$

Semantics 7 $u \models_{i,k} \neg E$ iff $u \not\models_{i,k} E$

Semantics 8 $u \models_{i,k} \Box E$ iff $(\forall j : k \leq j \Rightarrow u \models_{i,j} E)$

Semantics 9 $u \models_{i,k} \Diamond E$ iff $(\exists j : k \leq j$ and $u \models_{i,j} E)$

Example 1 Let $u = \langle efg \ldots \rangle$ be a trace in $\mathbf{U}_{\mathcal{T}}$. One can verify that (a) $u \models_0 \Diamond g$; (b) $u \not\models_0 \Diamond(g \cdot f)$; (c) $u \models_1 \Box e \wedge \neg f \wedge \neg g$; (d) $u \not\models_1 (e \cdot g)$; and (e) $u \models_3 (e \cdot g)$ (i.e., $(e \cdot g)$ 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 $\Diamond$. ▌

2.2 Important Properties

We describe some results, which are used in compiling and processing guards.

Definition 3 $E \cong F \triangleq (\forall i, k : u \models_{i,k} E \text{ iff } u \models_{i,k} F)$, where $E, F \in \mathcal{T}$.

Definition 4 [15] $E \equiv F$ iff $[\![E]\!] = [\![F]\!]$, where $E, F \in \mathcal{E}$.

Thus, $\cong$ means equivalence for $\mathcal{T}$ and $\equiv$ means equivalence for $\mathcal{E}$. The relations are not in $\mathcal{E}$ or $\mathcal{T}$. Observations 1 and 2 indicate a fundamental difference between $\mathcal{E}$ and $\mathcal{T}$.

Observation 1 $(\forall e : \top \not\cong e \vee \overline{e})$ and $(\forall e : 0 \cong e \wedge \overline{e})$. Also, $(\forall e : \top \cong \Diamond e \vee \Diamond \overline{e})$. ∎

Observation 2 $(\forall e : \top \not\equiv e \vee \overline{e})$ and $(\forall e : 0 \not\equiv e \wedge \overline{e})$. ∎

Definition 5 An event e is *stable* iff if e is satisfied at a given index, then it is satisfied at all future indices.

Observation 3 means that events are *stable*. However, Observation 4 shows that the temporal operators cannot always be eliminated. Intuitively, $\neg e$ means "not yet e." Observation 5 states that if the last event of a sequence has occurred, then the entire sequence has occurred. Similarly, Observation 6 states that if the first event of a sequence has not occurred, then the next event has not occurred either.

Observation 3 $\Box e \cong e$. ∎

Observation 4 $\Box \neg e \not\cong \neg e$. ∎

Observation 5 $\Diamond(e_1 \cdot e_2) \wedge \Box e_2 \cong \Box(e_1 \cdot e_2)$. ∎

Observation 6 $\Diamond(e_1 \cdot e_2) \wedge \neg e_1 \cong \Diamond(e_1 \cdot e_2) \wedge \neg e_1 \wedge \neg e_2$. ∎

Example 2 The possible maximal traces for $, = \{e, \overline{e}\}$ are $\{\langle e \rangle, \langle \overline{e} \rangle\}$. On different 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. Hence

- $\Box e \vee \Box \overline{e} \not\cong \top$: 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
- $\Diamond e \vee \Box \overline{e} \not\cong \top$: 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 $\neg e$. ∎

For larger alphabets, the set of traces is larger, but the above results hold. These and allied results were our main motivation in designing the formal semantics of $\mathcal{T}$.

2.3 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 3.2 discusses additional features. Section 4.2 exploits the special properties of our formal approach to give a series of formal results that improve elegance and efficiency.

The guards must permit precisely the traces that satisfy the given dependencies. We associate a set of *paths* with each dependency D. A path ρ is a sequence of event symbols that residuate D to $\top$—the dependency is satisfied if the events in the path occur in that order. We require that $,_\rho \supseteq ,_D$, i.e., all events in D (or their complements) feature in ρ. Each path is effectively a correct execution for its dependency. A path may have more events than those explicitly mentioned in a dependency. This is not a problem: section 4.2 develops an equivalent approach that only looks at the dependency itself, not the paths.

Definition 6 $D/\rho \triangleq ((D/e_1)/\ldots)/e_n$.

Definition 7 $\rho = \langle e_1 \ldots e_n \rangle$ is a path iff the events e_i are distinct, and not complements of each other.

Definition 8 $\Pi(D)$ is the set of paths satisfying D. $\Pi(D) \triangleq \{\rho : \rho \text{ is a path and } ,_\rho \supseteq ,_D \text{ and } D/\rho = \top\}$.

Lemma 7 means that the paths satisfy the given dependency. Lemma 8 establishes that there is a unique dependency corresponding to any set of paths.

Lemma 7 $\rho \in \Pi(D)$ iff ρ is a path and $,_\rho \supseteq ,_D$ and $\rho \models D$. ■

Lemma 8 $D \equiv \bigvee_{\rho \in \Pi(D)} \rho$. ■

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

Definition 9 $pre(\rho, e) \triangleq$ if $e = e_i$, then $\Box(e_1 \cdot \ldots \cdot e_{i-1})$, else 0.

Definition 10 $post(\rho, e) \triangleq$ 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.

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

Definition 11 $\mathsf{G}_b(\rho, e) \triangleq pre(\rho, e) \wedge post(\rho, e)$.
$\mathsf{G}_b(D, e) \triangleq \bigvee_{\rho \in \Pi(D)} \mathsf{G}_b(\rho, e)$.
$\mathsf{G}_b(\mathcal{W}, e) \triangleq \bigwedge_{D \in \mathcal{W}} \mathsf{G}_b(D, e)$.

Observation 9 $u \models_k \mathsf{G}_b(D, e) \Rightarrow (\exists \rho \in \Pi(D) : u \models_k \mathsf{G}_b(\rho, e))$. ∎

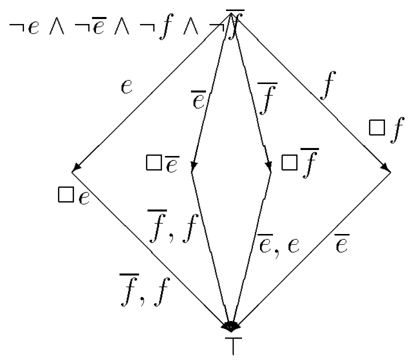

Figure 1: Guards with respect to $D_< = \overline{e} \vee \overline{f} \vee e \cdot f$

Figure 1 illustrates our procedure for the dependency of [15, Example 2]. The figure implicitly encodes all paths in $\Pi(D_<)$. By combining the paths into a graph, we reflect the "state of the scheduler" intuition of [15, section 3.2], and relate better to the results of section 4. Each path contributes the conjunction of *pre* and *post* for that event and path. The initial node is labeled $\neg e \wedge \neg \overline{e} \wedge \neg f \wedge \neg \overline{f}$ to indicate that no event has occurred yet. The nodes in the middle layer are labeled $\Box e$, etc., to indicate that the corresponding event has occurred. To avoid clutter, labels like $\Diamond e$ and $\neg e$ are not shown after the initial state.

Example 3 Using Figure 1, 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 $\Diamond \overline{e} \vee \Box e$.

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

3 Scheduling with Guards

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, e is rejected; else e is made to wait. Whenever e 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. Example 4 illustrates this. More complex cases follow.

Example 4 Using the guards from Example 3, 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. ▮

3.1 Mutual Constraints Among Events

The execution mechanism should avoid potential race conditions and deadlocks. It should also ensure that the necessary information flows to an event 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 4.

Prohibitory Relationships

During guard evaluation for an event e, a subexpression of the form $\neg f$ may need to be treated carefully. We must allow for situations where the message announcing f occurrence could be in transit when $\neg f$ is evaluated, leading to an inconsistent evaluation. A message exchange with f's actor is essential to ensure that f has not happened and is not happening.

Example 5 Following Example 3, e should not occur unless we can be sure that f has not occurred. ▮

This is a *prohibitory* relationship between events, since f's occurrence can possibly disable e (depending on the rest of the guard of e). Theorem 26 shows how prohibitory messages can sometimes be avoided.

Promissory Relationships

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 6 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 6 do not reflect an inconsistency, since f is allowed to occur after e. 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.

3.2 Incorporating Event Attributes

Any acceptable schedule must respect the intrinsic properties or *attributes* of the events that occur in it. The following attributes were introduced in [2]: (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.

To facilitate reasoning, it is useful to introduce the attributes *immediate* and *inevitable* as combinations of the above. We believe that *triggerable* is a more appropriate name for forcible events, because of its actual effect during execution. Thus our attributes are as follows (see [15, Figure 1] for the events mentioned):

- *Normal:* (Σ_n) delayable and rejectable, e.g., *commit*
- *Immediate:* (Σ_m) nondelayable and nonrejectable, e.g., *abort*
- *Inevitable:* (Σ_v) delayable and nonrejectable, e.g., *forget*—*forget* (not shown) corresponds to a task clearing its bookkeeping data and releasing locks
- *Triggerable:* (Σ_t) forcible, e.g., *start*.

For triggerable events, the dependencies are not modified, although the execution mechanism must be proactive. For other attributes, the dependencies must be modified—to eliminate traces that violate some event attribute— although the execution mechanism remains unchanged. The approach of section 2.3 applies to normal events; we consider inevitable and immediate events below.

3.2.1 Inevitable Events

An inevitable event must remain permissible on every trace until it or its complement (if attempted) has happened. Thus we eliminate paths on which there is a risk of violating the inevitability of the given event.

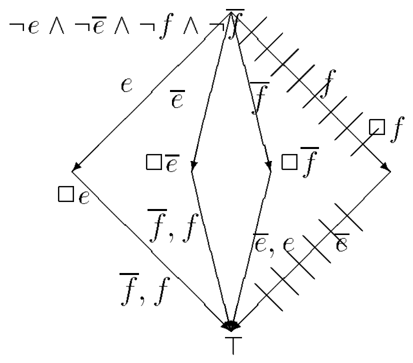

Figure 2: Guards from $D_<$ assuming e is inevitable

Figure 2 shows the dependency of Figure 1. The path $\langle f\overline{e}\rangle$ is deleted because if f occurs first, e must not occur. We can verify that $\mathsf{G}_b(D_<, e)$ is unchanged but $\mathsf{G}_b(D_<, f)$ is stronger: since e cannot be rejected, we cannot let f happen unless e or $\overline{e}$ has already happened.

3.2.2 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 unless the event or its complement has happened.

Example 7 Figure 2 still holds when e is immediate. Thus the same guards are obtained as before. ▮

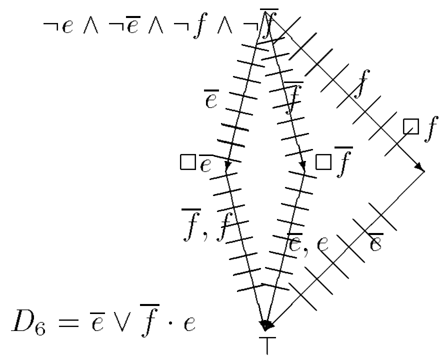

Figure 3: Extreme example of an immediate event (e)

Example 8 Referring to Figure 3, 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 4 below). ▮

3.2.3 Enforceability of Dependencies

A dependency is enforceable if, given the event attributes, its denotation is nonempty. Enforceability must be checked when dependencies are asserted. Individual dependencies may become unenforceable if the asserted attributes are overconstraining. Examples 9 and 10 show how the enforceability of dependencies can vary based on the event attributes that are asserted. Example 11 shows dependencies that are individually, but not jointly, enforceable.

Example 9 $D_<$ (in Figure 2) 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. ▮

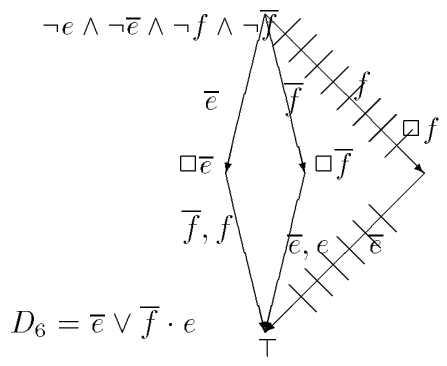

Figure 4: Unenforceable when e and f are inevitable

Example 10 $D_6 = \overline{e} \vee \overline{f} \cdot e$ (diagramed in Figure 4) is unenforceable when e and f are both inevitable, because there is no path on which they both occur. ▮

Example 11 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 e is inevitable, then D_6 and D_7 are jointly unenforceable. ▮

3.2.4 Triggerable Events

Triggerable events do not affect the guards, only their processing. A naive approach will cause e to wait forever if f is triggerable. A solution 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 12 Consider $D_{\rightarrow} = \overline{e} \vee f$ when f is triggerable. Now 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. ▮

A more complex case arises when the guards are as in Example 6, but the later event is to be triggered.

Example 13 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. ▮

4 Formalization

A careful formalization can help in proving the correctness of an approach and in justifying improvements in efficiency. Correctness is a concern when (a) guards are compiled, (b) guards are preprocessed, and (c) events are executed and guards updated.

Correctness depends on the *evaluation strategy*, which determines how events are scheduled. We formalize strategies by stating what initial values of guards they use, and how they update them. We begin with a strategy that is simple but correct and produce a series of more sophisticated, but semantically equivalent, strategies.

Given workflow $\mathcal{W}$, S yields a function $\mathsf{S}(\mathcal{W})$, which captures the evolution of guards and execution of events. Given an event e, a trace v, and index j in v, $\mathsf{S}(\mathcal{W}, e, v, j)$ equals the current guard of e at index j of v. Here v corresponds to the trace being "generated" and j indicates how far the computation has progressed.

Definition 12 An evaluation strategy is a function $\mathsf{S} : \wp(\mathcal{E}) \mapsto (, \times \mathbf{U}_{\mathcal{T}} \times \mathbf{N} \mapsto \mathcal{T})$.

Formally, 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 $\mathsf{S}(\mathcal{W}) \rightsquigarrow u$.

Definition 13 $\mathsf{S}(\mathcal{W}) \rightsquigarrow_i u$ iff $(\forall j : 1 \leq j \leq i \Rightarrow u \models_{j-1} \mathsf{S}(\mathcal{W}, u_j))$.

Definition 14 $\mathsf{S}(\mathcal{W}) \rightsquigarrow u \triangleq (\forall i : i \leq |u| \Rightarrow \mathsf{S}(\mathcal{W}) \rightsquigarrow_i u)$.

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 more abstract than execution.

In order to establish model-theoretic correctness of the initial compilation procedure given by Definition 11, we begin with a trivial strategy, S_b. S_b sets the guards using G_b and *never* modifies them. Theorem 11 establishes the soundness and completeness of guard compilation.

Definition 15 $(\forall v, j : \mathsf{S}_b(\mathcal{W}, e, v, j) = \mathsf{G}_b(\mathcal{W}, e))$.

Observation 10 $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$ iff $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(\mathcal{W}, u_j))$. ■

Theorem 11 $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$ iff $(\forall D \in \mathcal{W} : u \models D)$. ■

4.1 Evaluation

The operator $\div$ captures the processing required to assimilate different messages into a guard. This operator embodies a set of "proof rules" to reduce guards when events occur or are promised. Table 1 defines these rules. Because of two-sequence form (TSF) [15], we need not consider longer sequences.

When the dependencies involve sequence expressions, the guards can end up with sequence expressions, which indicate ordering of the relevant events. In such cases, the information that is assimilated into a guard must be consistent with that order. For this reason, the updates in those cases are more complex. Lemma 12 means that the operator $\div$ preserves the truth of the original guards.

Lemma 12 $(\exists k \leq j : u \models_k M$ and $u \models_j G \div M) \Rightarrow u \models_j G$. ■

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

Table 1: Assimilating Messages

The repeated application of $\div$ to update the guards 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 16 $\mathsf{S}_{\div}(\mathcal{W}, e, u, 0) \triangleq \mathsf{G}_b(\mathcal{W}, e)$.
$\mathsf{S}_{\div}(\mathcal{W}, e, u, i{+}1) \neq \mathsf{S}_{\div}(\mathcal{W}, e, u, i) \Rightarrow (\exists k \leq i : u \models_k M$ and $\mathsf{S}_{\div}(\mathcal{W}, e, u, i{+}1) = (\mathsf{S}_{\div}(\mathcal{W}, e, u, i) \div M))$.

$\mathsf{S}_{\div}$ does not require that every M that is true be used in reducing the guard. Lemma 12 enables us to accommodate message delay, because notifications need not be incorporated immediately. This is because when $\Box e$ and $\Diamond e$ hold at an index, they hold on all future indices.

Theorem 13 establishes that the evaluation of the guards according to $\div$ 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).

Theorem 13 Replacing S_b by $\mathsf{S}_{\div}$ preserves correctness, i.e., $\mathsf{S}_{\div}(\mathcal{W}) \leadsto u$ iff $\mathsf{S}_b(\mathcal{W}) \leadsto u$. ∎

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 13 establishes that any such modified execution is correct. However, executability requires in addition that we can take decisions without having to look into the future. The above theorem does not establish that the guards for each event will be reduced to $\Box$ and $\neg$ expressions (which require no information about the future). That depends on how effectively the guards are processed.

4.2 Simplification

We now show how guards as given in Definition 11 can be computed more efficiently. Theorems 14 and 15 show that the computations during guard compilation can be distributed over the conjuncts and disjuncts of a dependency. Since our dependencies are in TSF, this means the constituent sequence terms can be processed independently of each other. Theorem 15 is also important for another reason, namely, because it essentially equates a workflow with the conjunction of the dependencies in it.

Theorem 14 $\mathsf{G}_b(D \vee E, e) = \mathsf{G}_b(D, e) \vee \mathsf{G}_b(E, e)$. ■
Theorem 15 $\mathsf{G}_b(D \wedge E, e) = \mathsf{G}_b(D, e) \wedge \mathsf{G}_b(E, e)$. ■

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 17 means that the guard compilation essentially uses $\Psi(D)$—the other paths in $\Pi(D)$ can be safely ignored. Theorem 18 shows that the guard on an event e due to a dependency D is simply $\Diamond D$. This means that, for most dependencies and events, guards can be quickly compiled into a succinct expression.

Definition 17 $\Psi(D) \triangleq \{w : w \in \Pi(D), ,_w = ,_D\}$.
Lemma 16 $D \equiv \bigvee_{\rho \in \Psi(D)} \rho$. ■

Lemma 17 If $e \in ,_D$, then $\mathsf{G}_b(D, e) = \bigvee_{w \in \Psi(D)} \mathsf{G}_b(w, e)$. ■

Theorem 18 $\mathsf{G}_b(D, e) = \Diamond D$, if $e \notin ,_D$. ■

4.2.1 Relaxing the Past

Definition 18 allows us to replace certain sequences in the past component of Definition 11 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; it is omitted when it equals $\emptyset$).

Definition 18 $\mathsf{G}_p^\Delta(\mathcal{W}, e) \triangleq$ in $\mathsf{G}_b(\mathcal{W}, e)$ substitute $\Box(e_1 \cdot e_2)$ by $\Box e_1 \wedge \Box e_2$, if $\{e_1, e_2\} \cap \Delta \neq \emptyset$

Terms such as $\Box(e_1 \cdot e_2)$ are produced in the guards by an application of Observation 5. S_p^Δ is the evaluation strategy corresponding to G_p^Δ, in which updates happen as before. Theorem 19 establishes the correctness of S_p^Δ.

Theorem 19 For all Δ, replacing $\mathsf{S}_\div(\mathcal{W})$ by $\mathsf{S}_p^\Delta(\mathcal{W})$ preserves correctness. ■

4.2.2 Relaxing the Past and the Future

Can we do any better than the above? Yes and No. In some cases, we can replace all sequences in guards with conjunctions The idea is that each event would locally capture what comes before and what comes after. $\mathsf{S}^{\Delta}_{\wedge}$ is the evaluation strategy corresponding to $\mathsf{G}^{\Delta}_{\wedge}$.

Definition 19 $\mathsf{G}^{\Delta}_{\wedge}(\mathcal{W}, e) \triangleq$ in $\mathsf{G}^{\Delta}_{p}(\mathcal{W}, e)$ substitute $\neg e_1 \wedge \Diamond(e_1 \cdot e_2)$ by $\neg e_1 \wedge \Diamond e_1 \wedge \neg e_2 \wedge \Diamond e_2$, if $\{e_1, e_2\} \cap \Delta \neq \emptyset$

But Lemma 20 holds, because Definition 19 allows the traces validly generated from the given dependency to be combined into spurious traces.

Lemma 20 Replacing $\mathsf{S}_{\div}(\mathcal{W})$ by $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W})$ does not preserve correctness. ■

However, we obtain a positive result by restricting the syntax of dependencies. A *3-dependency* is in "3-clausal" form, which means that it has no conjunctions and each sequence expression has no more than 3 events. A *3-workflow* is a *single* 3-dependency, i.e., has been combined into a single 3-dependency to show that it does not "hide" a sequence of more than 3 events. 3-workflows cover many of the cases of interest, including [15, Example 4], and Examples 15 and 16 of section 4.5. Lemma 21 shows the present formulation applies to 3-workflows.

Syntax 5 $0, \top \in \mathcal{E}_3$

Syntax 6 $e_1 \cdot \ldots \cdot e_n \in \mathcal{E}_3$ if $e_i \in$, and $n \leq 3$

Syntax 7 $E_1, E_2 \in \mathcal{E}_3$ implies that $E_1 \vee E_2 \in \mathcal{E}_3$

Lemma 21 If $\mathcal{W}$ is a 3-workflow, then replacing $\mathsf{S}_{\div}(\mathcal{W})$ by $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W})$ preserves correctness. ■

Given a 3-workflow, we can compile the guards modularly from the original dependencies using Definition 19. However, to determine whether a given workflow is a 3-workflow requires combining the dependencies to check that no sequence is longer than 3 events. This computation can, in the worst case, be exponential in the number of dependencies, but is often tractable. It only needs to be performed once at compile-time, and may be avoided if one decides to forego possible simplification.

4.2.3 Eliminating Irrelevant Guards

Theorem 22 shows that the guard on an event e due to a dependency D in which e does not occur can be set to $\top$, provided D is entailed by the given workflow—an easy test of entailment is that D is in the workflow. Thus dependencies in the workflow that don't mention an event can be safely ignored! 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, $\mathsf{G}^{\Delta}_{\top}$ replaces the irrelevant guards for events not in Δ; $\mathsf{S}^{\Delta}_{\top}$ is the corresponding strategy.

Definition 20 $\mathsf{G}^{\triangle}_{\top}(D, e) = \top$, if $e \not\in\ ,_D$ and $D \in \mathcal{W}$, where $e \not\in \Delta$; $\mathsf{G}^{\triangle}_{\top}(D, e) = \mathsf{G}^{\triangle}_{p}(D, e)$ otherwise.

Theorem 22 Replacing $\mathsf{S}^{\triangle}_{p}(\mathcal{W})$ by $\mathsf{S}^{\triangle}_{\top}(\mathcal{W})$ does not violate correctness. ∎

Theorems 15 and 22 establish that the guard on an event e due to a conjunction of dependencies is the conjunction of the guards due to the individual dependencies that mention e. Thus, we can compile the guards modularly and obtain expressions that are more amenable to processing.

4.3 Formalizing Event Attributes

We 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. That is, 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 $|,_D|/2$.

Definition 21 $\sigma^0(D) \triangleq \Psi(D)$.
$\sigma^{i+1}(D) \triangleq \sigma^i(D) \cap \{w : \beta(w, \sigma^i(D))\}$, where $\beta(w, P)$ iff

- if $e \in\ ,_D \cap \Sigma_v$, 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 $e \in\ ,_D \cap \Sigma_m$, 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 22 $\Phi(D) \triangleq \sigma^{\lceil |\Gamma_D|/2 \rceil}(D)$.

$\Phi(D)$ 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 $\alpha(D)$ as the disjunction of the safe paths. Lemmas 23 and 24 show that the formal definition behaves as desired. Lemmas 23 and 24 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 $\alpha(D)$.

Definition 23 $\alpha(D) \triangleq \bigvee_{w \in \Phi(D)} w$.
$\alpha(\mathcal{W}) = \{\alpha(D) : D \in \mathcal{W}\}$.

Lemma 23 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 24 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 $v \sim_l \alpha(D))$. ∎

Example 14 We can verify 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 3, we can check that if $\Sigma_m = \{e\}$, then $\alpha(\overline{e} \vee \overline{f} \cdot e) = 0$. ∎

Scheduling with Event Attributes

If $e \in \Sigma_v$ is attempted, it is made to wait until the opportune moment. When its guard becomes $\top$, it can happen. $\Diamond e$ 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 guard on an immediate event may not be $\top$. Yet, such an event must be allowed immediately. When events must be mutually ordered, the guards redundantly attach ordering information on all events that are ordered. 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 allow each order of e and f.) S_m is the strategy that enables immediate events to happen immediately.

Definition 24 $\mathsf{S}_m(\mathcal{W}, e, u, i) \triangleq$ (if $e \in \Sigma_m$, then $\top$, else $\mathsf{S}_{\neg}^{\Sigma_m}(\mathcal{W}, e, u, i)$).

Theorem 25 Replacing $\mathsf{S}_{\neg}^{\Delta}$ by S_m does not violate correctness. ∎

Avoid Prohibitory Messages

For any two ordered events, the guards of both include information about their ordering. This can be exploited by letting the event which comes first in the given order to occur anyway, and making the event which comes later to be responsible for the ordering. In other words, while evaluating guards, subexpressions of the form $\neg f$ can be treated as $\top$, unless f is immediate. Theorem 26 shows that this simplification does not violate correctness. Thus events can be executed with greater decoupling. Expressions such as $\neg f$ are not equivalent to $\top$, because they can reduce to 0 when a $\Box f$ message arrives. $\mathsf{S}_{\neg}^{\Delta}$ 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 $\mathsf{S}_{\top}^{\Delta}$—the reduction kicks in independently at each moment.

Definition 25 $\mathsf{S}_{\neg}^{\Delta}(\mathcal{W}, e, u, i) =$
$\mathsf{S}_{\top}^{\Delta}(\mathcal{W}, e, u, i)|_{(\forall f \in \Gamma \backslash \Delta : \Box f ::= 0, \neg f ::= \top)}$.

Theorem 26 For all Δ, replacing $\mathsf{S}_{\top}^{\Delta}$ by $\mathsf{S}_{\neg}^{\Delta}$ does not violate correctness. ∎

4.4 Preprocessing and Execution

Generation is abstract in that a trace may be generated based on future events. In actual execution, we cannot look into the future. Sections 3.1 and 3.2.4 discussed some approaches through which the desired computations may be realized by sending additional messages.

These approaches set up the exchange of certain messages based on relationships among the events. These heuristics are obviously sound, because they merely call for the exchange of messages that contain true assertions. 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 [Example 15 below] 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 [Example 6] 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 Heuristic 2 does not apply in the case where $n > 2$ events have a cyclic dependency. For such cases, we can use the present heuristic. Let $\{e_0, \ldots, e_{n-1}\}$ be a set of events such that $(\forall i : 0 \leq i < n \Rightarrow \mathsf{G}(e_i) \div \Diamond(e_{i+1 \bmod n}) = \top)$, then set up a promise from e_i to $e_{i+1 \bmod n}$ when e_i is attempted and a promise has not already been received that updates its guard to $\top$. $\bullet$

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

Theorem 27 Heuristics 1, 2, 3, and 4 are sound. ∎

We assume that the underlying message transport layer guarantees message ordering. We also assume that the events are ordered at the appropriate granularity of time.

4.5 Examples

We can compile and preprocess the guards on the different events in the workflow of [15, Example 4] to achieve the desired effect. 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 15 considers the workflow $\{D_1, D_2, D_3\}$; Example 16 adds back D_4. For simplicity, dependencies are given in 3-clausal form rather than TSF.

Example 15 Because of the above attributes, dependency D_3 is 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 $\{D_1, D_2, \alpha(D_3)\}$:

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

We can enact this workflow as follows. Suppose s_{buy} is attempted. It is accepted immediately, because its guard would be satisfied by a triggerable event with a true guard. Thus, s_{book} is triggered. Suppose c_{book} is attempted. It too is accepted because its guard would be satisfied by a triggerable event with a true guard. $\mathsf{G}(\overline{s_{cancel}}) \div \Box c_{book} = \Box c_{buy}$. 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. ∎

Example 16 In Example 15, 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 events c_{book}, $\overline{c_{book}}$, c_{buy}, $\overline{c_{buy}}$, s_{cancel}, and $\overline{s_{cancel}}$.

This proceeds similarly to Example 15. 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. ▌

5 Enhancements

Our approach can accommodate simple real-time dependencies. We model clock values or timer interrupts simply as immediate events. 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.

General real-time scheduling involves reasoning about timer events before they occur, so as to allow enough time for various activities. Such reasoning might be performed while designing effective workflows. This is beyond the scope of this paper.

5.1 Dynamic Modification of Workflows

Modifications can take the form of dependencies being added or removed, or attributes being modified. Checking violation of any dependency or attribute requires maintaining the entire history of the computation, and may not be practical.

When dependencies are removed, their contributions to the guards should be removed to avoid over-constraining the workflow. For this purpose, we can recompile the guards, or maintain the contribution of each dependency to each guard to facilitate updates.

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 promised in the correct order.

Modifying event attributes is unintuitive, since it means a change in the inherent structure of the component tasks of a workflow, and should not occur during 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.

5.2 Parametrization

So far, we considered specific event instances. We consider languages $\mathcal{E}_P$ [15, section 5] and $\mathcal{T}_P$ to interpret events as types, and parametrize them to create unique event instances. $\mathcal{T}_P$

uses the parametric forms introduced in $\mathcal{E}_P$, just as $\mathcal{T}$ used $\mathcal{E}$.

We assume that (a) events from the same task have the same variable parameters, and (b) all references to the same event type involve the same tuple of parameters. These assumptions are reasonable because our focus is on intertask dependencies. 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. Since the event IDs need *not* depend on the structure of the task, the scheduler behaves independently of the structure. 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!

$\mathcal{V}$ 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, , includes all (ground) event literals and Ξ includes all event atoms. Since the universe depends on , , it is automatically redefined to include all traces formed from all possible event instances. We add a semantic rule for $\mathcal{T}_P$:

Semantics 10 $u \models_i E(v)$ iff $(\forall c \in \mathcal{C} : u \models_i E(v ::= c))$

The guard definitions are unchanged. The variables are instantiated to the appropriate constants when an event is attempted or triggered. The updates on guards due to event occurrences of promises are still performed using the $\div$ operator, enhanced to accommodate parameters.

Definition 26 $G(\vec{v}) \div M(\vec{c}) \triangleq G(\vec{v}) \wedge G(\vec{v} ::= \vec{c}) \div M(\vec{c})$.

Parametrized Workflows

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 15.

Example 17 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 15, but with an explicit parameter, which is bound when the event is attempted or a message received from another event. 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

The different events may have unrelated variable parameters. Such cases occur in the specification of concurrency control requirements *across* workflows or transactions. Example 18 shows how mutual exclusion may be captured in our approach.

Example 18 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 occurred, $b_2[y]$ will not occur, or $b_2[y]$ will occur after $e_1[\hat{x}]$, and (b) either $b_2[y]$ has not occurred or $e_2[y]$ has occurred. 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 $\mathsf{G}(b_2[y]) \div \Box b_1[\hat{x}] = \Box e_1[x]$, we obtain that $\mathsf{G}(b_2[y]) = (\Box b_1[x] \vee \Diamond\overline{b_1[x]} \vee \Diamond(e_2[y] \cdot b_1[x])) \wedge (\neg b_1[x] \vee \Box e_1[x]) \wedge \Box e_1[\hat{x}]$.
- 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. ▮

Thus parameters add naturally to our approach and support the usual semantics. Although individual workflows can be parametrized in a straightforward manner, interactions between independent workflows require greater subtlety. 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.

6 Overview of the Literature

Some related literature is discussed in [15]. Several execution environments have been proposed, which support specification and execution of transactions. An actor-based environment is developed in [8]. The DOM project includes a programmable environment (TSME) in which several transaction models can be specified [6]. However, TSME defines correctness criteria based on transaction histories, like in traditional approaches. The task specification languages for *interactions* [12] and METEOR [11] are similar in intent. ASSET is a programming facility for specifying transactions in the Ode environment [3]. This facility borrows intuitions, but not the formalism, from ACTA (discussed below). [9] develops a capability-based framework for activity management. This is a promising approach, which combines ideas from problem-solving agents and activity decomposition. Event-condition action rules are derived from activity graphs. These rules are then executed. The framework is general. Like our approach, it does not look into the details of the tasks. Unlike our approach, however, it is centralized and cannot yet handle concurrency.

ACTA was the first attempt at formalizing the semantics of extended transaction models [4]. It introduced significant events of database transactions. ACTA provides a history-based formalism for specifying intertask dependencies. It is similar in spirit to [15], 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 the basis for the present paper.

Klein's approach is also event-centric and distributed [10]. However, it is limited to loop-free tasks, and doesn't handle event attributes generically. Günthör's approach is based on temporal logic, but is centralized [7]. These approaches are somewhat *ad hoc* 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 [2]. 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. Recently, a distributed prototype was proposed, but it is not given a formal basis [13].

In our system, events variously wait, send messages to each other, and thereby enable or trigger each other. This appears intuitively similar to Petri nets, which can be applied to workflows [17]. However, our goal 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 messages. 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.

7 Conclusions and Future Work

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. A prototype of our system was implemented in an actor language. It is being reimplemented in Java. [1] report an alternative implementation of our approach in which they restrict the language to capture some commonly occurring workflow patterns. This is a promising approach to optimize important patterns.

Future work includes exploring 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 and improved characterization of the syntactical restrictions with which greater efficiency may be achieved. We are considering an alternative implementational approach in which no preprocessing is performed, but promises are always generated. It remains to be seen if this will prove acceptable.

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 workflow. This is the focus of a follow on research project.

References

[1] M. Altinel, E. Gokkoca, I. Cingil, E. Nesime, P. Koksal, and A. Dogac. Design and Implementation of a Distributed Workflow Enactment Service. In *Proc. Conf. on Intell. & Coop. Info. Syst. (CoopIS)*, 1997.

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

[3] A. Biliris, S. Dar, N. Gehani, H. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. In *Proc. ACM SIGMOD Conf.*, 1994.

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

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

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

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

[8] M. Haghjoo, M. Papazoglou, and H. Schmidt. A semantics-based nested transaction model for intelligent and cooperative information systems. In *Proc. Conf. on Intell. & Coop. Info. Syst. (CoopIS)*, 1993.

[9] K. Karlapalem, H. Yeung, and P. Hung. CapBasED-AMS: A Framework for Capability Based and Event Driven Activity Management System. In *Proc. Conf. on Intell. & Coop. Info. Syst. (CoopIS)*, 1995.

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

[11] N. Krishnakumar and A. Sheth. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Dist. & Par. Databases*, 1994.

[12] M. Nodine, N. Nakos, and S. Zdonik. Specifying flexible tasks in a multidatabase. In *Proc. 2nd Conf. Coop. Info. Syst. (CoopIS)*, 1994.

[13] A. Sheth, K. Kochut, J. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, and I. Shevchenko. Supporting state-wide immunization tracking using multi-paradigm workflow technology. In *Proc. 22nd VLDB Conf*, 1996.

[14] M. Singh. Semantical considerations on workflows: Algebraically specifying and scheduling intertask dependencies. In *Proc. DB Prog. Lang. (DBPL)*, 1995.

[15] M. Singh. Formal Aspects of Workflow Management, Part 1: Semantics. TR-97-04, Computer Science, NCSU, Raleigh, May 1997. *Extends [14].* `http://www.csc.ncsu.edu/ faculty/ mpsingh/ papers/ databases/ wf_semantics.ps`.

[16] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in Carnot. *IEEE Par.& Dist. Tech.*, 16–20, 1993.

[17] W. van der Aalst. Petri-net-based workflow management software. In *Proc. NSF Wkshp. Workflow and Process Automation in Info. Syst.*, 1996.

A Proofs of Important Results

Auxiliary Definitions 1

Definition 27 $(v \sqsupseteq w) \triangleq v$ contains the events of w in the same relative order.

Observation 28 If $v \sqsupseteq w$, then $w \in \Pi(D) \Rightarrow v \in \Pi(D)$. ∎

Observation 29 $\Pi(D \wedge E) = \Pi(D) \cap \Pi(E)$. ∎

Observation 30 $D \equiv E \not\Rightarrow \Pi(D) = \Pi(E)$. ∎

For traces u and v, $u \sim_i v$ means that u agrees with v up to index i. $u \sim_i D$ means that u agrees with dependency D up to index i. Lemma 31 relies on the maximality of u.

Definition 28 $u \sim_i v \triangleq i \leq |u|$ and $i \leq |v|$ and $(\forall j : 1 \leq j \leq i \Rightarrow u_j = v_j)$.

Definition 29 $u \sim_i D \triangleq (\exists v \in [\![D]\!] : u \sim_i v)$.

Lemma 31 $u \sim_{|u|} D \Rightarrow u \in [\![D]\!]$. ∎

Lemma 32 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, $u \sqsupseteq w$. ∎

To simplify the notation, 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 $j = i$.

Proof of Theorem 11

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

Proof. Consider any dependency $E \in \mathcal{W}$. By Observation 10, $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(E, u_j))$.

By Observation 9, $u \models_0 \mathsf{G}_b(E, u_1)$ implies that $(\exists w : w \in \Pi(E)$ and $u \models_0 \mathsf{G}_b(w, u_1))$. By Lemma 32, $u \sqsupseteq w$. By Observation 28, $u \in \Pi(E)$. Therefore, by Lemma 7, $u \models E$. This holds for all dependencies in $\mathcal{W}$.

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, $(\forall j : 1 \leq j \leq |u| \Rightarrow (\mathsf{G}_b(u, u_j) \Rightarrow \mathsf{G}_b(D, u_j)))$. We have $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(D, u_j))$. Hence, by Observation 10, $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$.

Proof of Lemma 12

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

Proof of Theorem 13

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 16, 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 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 12, $(\exists k \leq l : u \models_k M$ 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.

Auxiliary Definitions 2

The next theorems rely on additional auxiliary definitions and results. $I(w, , ')$ gives all the superpaths of w that include all interleavings of w with the events in $, '$. We assume that $(\forall e : e \in , '$ iff $\overline{e} \in , ')$. Lemma 33 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 $, '$ are considered. For each event e in $, '$, e and $\overline{e}$ can occur anywhere relative to w, and thus they essentially factor out. Lemma 34 shows that the guards are well-behaved with respect to denotations.

Definition 30 $I(w, , ') \triangleq \{v : ,_v = ,_w \cup , '$ and $v \sqsupseteq w\}$.

Lemma 33 If $e \in ,_w$, then $\mathsf{G}_b(w, e) = \bigvee_{v \in I(w, \Gamma')} \mathsf{G}_b(v, e)$. ∎

Lemma 34 $D \equiv E \Rightarrow \mathsf{G}_b(D, e) = \mathsf{G}_b(E, e)$. ∎

Proof of Theorem 14

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

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 30 as $I(w, ,_{D \vee E} \setminus ,_w)$, which contains all interleavings of w with events in $,_{D \vee E}$ that aren't in $,_w$. By Observation 28, $I(w) \subseteq \Pi(D)$. Also, $I(w) \subseteq \Pi(D \vee E)$. By Lemma 33, $\mathsf{G}_b(w, e) = \bigvee_{v \in I(w, \Gamma_{D \vee E} \setminus \Gamma_w)} \mathsf{G}_b(v, e)$. Thus the contribution of w to $\mathsf{G}_b(w, e)$ is covered by paths in $\Pi(D \vee E)$.

Proof of Theorem 15

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

Proof. $\mathsf{G}_b(D \wedge E, e) = \bigvee_{w \in \Pi(D \wedge E)} \mathsf{G}_b(w, e)$. By Observation 29, $\Pi(D \wedge E) = \Pi(D) \cap \Pi(E)$. 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 28, $x \in \Pi(D) \cap \Pi(E)$. By Observation 29, $x \in \Pi(D \wedge E)$. 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 35 $\mathsf{G}_b(D, e) = \mathsf{G}_b(D \wedge e, e)$. ∎

Lemma 36 $\mathsf{G}_b(e_1 \cdot e_2, e) = \Diamond(e_1 \cdot e_2)$, if $e, \overline{e} \not\in \{e_1, e_2\}$.

Proof. Let $D^e = (e_1 \cdot \ldots \cdot e_n) \wedge e$. By Lemmas 17 and 35, $\mathsf{G}_b(D, e) = \bigvee_{w \in \Psi(D^e)} \mathsf{G}_b(w, e)$. Let $w = \langle e_1, \ldots, e_k, e, e_{k+1}, \ldots, e_n \rangle \in \Psi(D^e)$. Then
$\mathsf{G}_b(w, e) = \Box(e_1 \cdot \cdots \ldots \cdots e_k) \wedge \neg e_{k+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond(e_{k+1} \cdot \ldots \cdot e_n)$. There is one such w for each position of e, i.e., for $0 \leq k \leq n$. Thus,
$\mathsf{G}_b(D, e) = \bigvee_{0 \leq k \leq n} \Box(e_1 \cdot \cdots \ldots \cdots e_k) \wedge \neg e_{k+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond(e_{k+1} \cdot \ldots \cdot e_n)$. It is easy to verify that for any trace u and index i, $u \models_i \mathsf{G}_b(D, e)$ iff $u \models_i \Diamond D$. ∎

Proof of Theorem 18

$\mathsf{G}_b(D, e) = \Diamond D$, 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 36, $\mathsf{G}_b(D, e) = \Diamond D$. Since $D \in \mathcal{E}$, we can show $\Diamond(D_1 \vee D_2) \cong \Diamond D_1 \vee \Diamond D_2$, and $\Diamond(D_1 \wedge D_2) \cong \Diamond D_1 \wedge \Diamond D_2$.

Definition 31 $\mathsf{S}_p^\Delta(\mathcal{W}, e, u, 0) = \mathsf{G}_p^\Delta(\mathcal{W}, e)$.
$\mathsf{S}_p^\Delta(\mathcal{W}, e, u, i+1) \neq \mathsf{S}_p^\Delta(\mathcal{W}, e, u, i) \Rightarrow$
$e \not\in \Delta$ and $(\exists k \leq i : u \models_k M$ and $\mathsf{S}_p^\Delta(\mathcal{W}, e, u, i+1) = (\mathsf{S}_p^\Delta(\mathcal{W}, e, u, i) \div M))$.

Proof of Theorem 19

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 32, $u \sqsupseteq w$. Thus, $u \models D$. Thus no additional spurious traces are allowed by $\mathsf{S}_p^\Delta(\mathcal{W})$.

Definition 32 $\mathsf{S}_\wedge^\Delta(\mathcal{W}, e, u, 0) = \mathsf{G}_\wedge^\Delta(\mathcal{W}, e)$.
$\mathsf{S}_\wedge^\Delta(\mathcal{W}, e, u, i+1) \neq \mathsf{S}_\wedge^\Delta(\mathcal{W}, e, u, i) \Rightarrow$
$(\exists k \leq i : u \models_k M$ and $\mathsf{S}_\wedge^\Delta(\mathcal{W}, e, u, i+1) = (\mathsf{S}_\wedge^\Delta(\mathcal{W}, e, u, i) \div M))$.

Proof of Lemma 20

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

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

Proof of Lemma 21

If $\mathcal{W}$ is a 3-workflow, then replacing $\mathsf{S}_\div(\mathcal{W})$ by $\mathsf{S}_\wedge^\Delta(\mathcal{W})$ 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 $u \models_{k-1} \mathsf{G}_\wedge^\Delta(w, f)$. 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 $\mathcal{W}$.

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

Proof of Theorem 22

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

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

Consider $D \in \mathcal{W}$ and $e \notin \;,_D$. Let $f \in \;,_D$. By Definition 20 and Lemma 17, we have that $\mathsf{G}_{\top}^{\Delta}(D, f) = \bigvee_{w \in \Psi(D)} \mathsf{G}_p^{\Delta}(w, f)$. 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 $\mathsf{S}_{\top}^{\Delta}(\mathcal{W}) \leadsto u$. If $u \not\models_j \mathsf{S}_p^{\Delta}(D, u_{j+1})$ and $u \models_j \mathsf{S}_{\top}^{\Delta}(D, u_{j+1})$, then $u_{j+1} \notin \;,_D$. Let $B(u) = \{u_i : u \not\models_{i-1} \mathsf{S}_p^{\Delta}(D, u_i)\}$. Let v be such that $u \sqsupseteq v$ and $,_v = \;,_u \setminus B(u)$. Since the guards for events in $,_D$ do not depend on u_{j+1}, we have that $(\forall k, l : 1 \leq k$ and $1 \leq l$ and $u_k = v_l \Rightarrow u \models_{k-1} \mathsf{G}_p^{\Delta}(D, u_k)$ iff $v \models_{l-1} \mathsf{G}_p^{\Delta}(D, v_l))$. Hence, $\mathsf{S}_p^{\Delta}(\mathcal{W}) \leadsto v$. By Theorem 19, $v \models D$. By Observation 28, $u \models D$.

Proof of Theorem 26

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

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

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

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

Proof of Theorem 25

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 $\mathsf{S}_m(\mathcal{W}) \leadsto u$. Let $u_j \in \Sigma_m$. Consider $D \in \mathcal{W}$. If all u_i belong to Σ_m, then by Lemma 23 $u \in [\![D]\!]$. Let $u_k \in \;,_D \setminus \Sigma_m$. 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 32, $u \sqsupseteq v$. Thus, $u \models D$. Hence, $\mathsf{S}_{\neg}^{\Delta}(\mathcal{W}) \leadsto u$.

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