Coordinating Heterogeneous Autonomous Agents

Munindar P. Singh*
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534, USA

singh@ncsu.edu

Abstract

We address the problem of constructing multiagent systems by coordinating heterogeneous, autonomous agents, whose internal designs may not be fully known. We develop a customizable coordination service that (a) takes declarative specifications of the desired interactions, and (b) automatically enacts them. Our approach is based on temporal logic, and has a rigorous semantics and a naturally distributed implementation. We show how this approach provides a general unifying framework in which to formalize the coordination components of some recent approaches to multiagent system construction.

*Munindar Singh is supported by the NCSU College of Engineering, the National Science Foundation under grants IRI-9529179 and IRI-9624425, and IBM corporation.

1 Introduction

Modern applications of computing arise most frequently in environments that are open, heterogeneous, distributed, dynamic, large, and with autonomous components. For these reasons, they require solutions that marry artificial intelligence (AI) and traditional techniques to yield extensibility and flexibility. Agents are an issue of this marriage. Currently, many agent approaches are centralized in a single agent. However, centralization has obvious shortcomings in accommodating the above properties of open environments. Consequently, there has been increasing interest in multiagent systems, which can yield the benefits of intelligent agency while preserving openness and scalability.

What sets multiagent systems apart from single agents is that they require the agents to behave in a coordinated manner—agents must follow some protocol even to compete effectively. Therefore, the designer of a multiagent system must handle not only the application-specific aspects of the various agents, but also their interactions with one another. Current approaches to constructing multiagent systems offer no special coordination support to the designer, who must manually ensure that the (potentially autonomous) agents interact appropriately. This can lead to unnecessarily rigid or suboptimal designs, wasted development effort, and sometimes to the autonomy of the agents being violated. These limitations, in effect, subvert many of the features that make multiagent systems attractive in the first place.

Technical Motivation We believe that it is the difficulty of constructing effective coordination that has led many researchers and practitioners to the centralized approaches. An effective approach to coordination must support some important features to be of practical value in the design of multiagent systems.

We postulate that in many interesting applications of multiagent systems, the individual agents will be preexisting or contributed by different vendors. Thus, a suitable approach to coordination ought neither unnecessarily restrict the architecture or design of the individual agents, nor demand knowledge of its details. Similarly, in many important applications, the agents will represent different interests. Thus, a suitable coordination approach ought not to compromise the agents' autonomy. Further, each approach to coordination has some metamodel associated with it that defines what can be coordinated in what circumstances. To assist in the development of systems, this metamodel must be highlighted and related to a methodology for building agents and multiagent systems.

We view coordination as a substrate of functionality upon which additional functionalities, for example, for social commitments and collaboration, are built. We propose that coordination be separated into a distinct service. The service would be responsible for delivering the desired coordination. This presupposes that the service takes declarative specifications of the desired interactions, and includes the functionality to enact them. This service should be customizable, because each application has its own requirements for coordination. It should be minimally intrusive so as to preserve the autonomy of the participating agents. Such a service can help improve

- designer productivity, by providing higher-level abstractions
- system efficiency by optimizing the desired coordination for the specific environment in which it is applied.

Simplicity and rigor are both crucial: a service should be easy to use and highly reliable. Intuitively, this service is analogous to truth maintenance systems (TMSs), which are immensely successful because of their simplicity, and enable design of complex systems.

Approach The above is the kind of coordination service we have developed. Our service mediates between the infrastructure and the application-specific components. It includes functionality to specify the desired coordination, translate them into low-level *events*, and schedule them through passing appropriate messages among agents. The low-level events correspond to the agents' significant (external) transitions. Capturing the specifications of the coordination explicitly enables us to flexibly execute them, thereby maintaining the key properties of interactions across different situations. Thus a programmer can create a multiagent system by defining (or reusing) agents, and setting them up to interact as desired. Managing the coordination requires knowledge only of the agents' external events that feature in the interactions, not of the details of their behavior. Our service is rigorous, being based on temporal logic—it enhances techniques from workflow and relaxed transaction scheduling in databases. It includes abstractions for (a) a semantics of events in a multiagent system and (b) message passing to implement control and data flow, for example, as described by Hewitt [27]. Our approach naturally leads to a distributed implementation.

Relevant Approaches Because we separate out the components of commitment and collaboration, our view of coordination is narrower than of some others in the distributed AI literature. Our service can effectively support the higher-level requirements of some of the traditional approaches, for example, those of Decker & Lesser [13] and Sycara & Zeng [54]. In this manner, it would not replace their insights, but assist in their realization. Section 15 revisits this point.

Several approaches deal with coordination in multiagent systems. It is easier to relate our approach to these previous approaches after we have developed some of our terminology and concepts. Accordingly, we consider dMARS [33] and STEAM [55] in some detail in Section 4.2 to show how we can model their coordination components here. We also consider executable temporal as epitomized by Concurrent MetateM [17, 60] in Section 15.

Organization Section 2 motivates and presents our conceptual approach. Section 3 describes our specification language. Section 4 uses it to specify some important relationships, and to model some approaches in the literature. Section 5 shows how the service operates. Section 15 reviews the pertinent literature.

2 Coordination Service

Although our approach is generic, we consider information search applications for concreteness. In such applications, agents cooperate to perform combinations of tasks such as resource discovery, querying heterogeneous databases, and information retrieval, filtering, and fusion. Our running example follows.

Example 1 Consider a ship on the high seas. Suppose an engine spare-part, a valve, runs low in the ship's inventory. This can lead the maintenance engineer to a search for information: Are such valves available at the next sea-port to be visited? Intuitively, she must access the bridge to find the next sea-port, query a directory of suppliers, and call up the suppliers at the next sea-port. ▮

Consider a multiagent approach that uses information agents for each resource—a common information architecture, for example, see Huhns *et al.* [29]:

Example 2 The search of Example 1 involves querying the bridge agent for the next port, querying a directory agent to find suppliers in the next port, and mapping over the list of suppliers to ask each of their agents about the desired valve. One positive response is enough, but additional responses improve reliability and help optimize other criteria, for example, the price. ▮

Clearly, since the directory and suppliers are autonomous, so must their agents be. Further, these agents may have differing designs, which may not be revealed by their implementers. However, the agents must be coordinated to carry out the search.

2.1 Coordination Model

There are two aspects of the autonomy of agents that concern us. One, the agents are designed autonomously, and their internal details may be unavailable. Two, the agents act autonomously, and may unilaterally perform certain actions within their purview. We assume that, in order to be able to coordinate the at all, the designer of the multiagent system has some limited knowledge of the designs of the individual agents. This knowledge is in terms of their externally visible actions, which are potentially significant for coordination. We call these the significant *events* of the agent. In other words, the only events we speak of are those publicly known—the rest are of no concern to the service. These events are organized into *skeletons* that characterize the coordination behavior of the agents. The idea of using events and skeletons is well-known from logics of programs, especially since Emerson & Clarke [16], and has also been used in the database transaction modeling community [10].

2.1.1 Event Classes

Our metamodel considers four classes of events, which have different properties with respect to coordination. Events may be

- *flexible*, which the agent is willing to delay or omit
- *inevitable*, which the agent is willing only to delay
- *immediate*, which the agent performs unilaterally, that is, is willing neither to delay nor to omit
- *triggerable*, which the agent is willing to perform based on external request.

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

2.1.2 Agent Skeletons

It is useful to view the events as organized into a *skeleton* to provide a simple model of an agent for coordination purposes. This model is typically a finite state automaton. Although the automaton is not used explicitly by the coordination service during execution, it can be used to validate specified coordination requirements. The set of events, their properties, and the skeleton of an agent depends on the agent, and is application-specific. The coordination service is independent of the exact skeletons or events used in a multiagent system. Traditional database approaches, for example, Chrysanthis & Ramamritham [10], are limited to loop-free skeletons, which correspond to single-shot queries or transactions. However, we place no such restrictions here. Examples 3 and 4 discuss two common skeletons in information search. A skeleton for dMARS plans is introduced in Section 4.2.2.

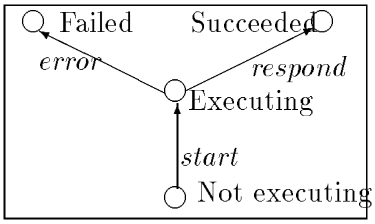


Figure 1: Skeleton for a Simple Querying Agent

Example 3 Figure 1 shows a skeleton that is suited for agents who perform one-shot queries. Its significant events are *start* (accept an input and begin), *error*, and *respond* (produce an answer and terminate). The application-specific computation takes place in the node labeled "Executing." We must also specify the classes of the different events. For instance, we may state that *error* and *respond* are immediate, and *start* is flexible and triggerable. ▮

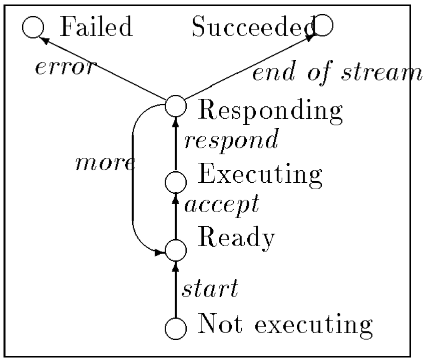


Figure 2: Skeleton for an Information Filtering Agent

Example 4 Figure 2 shows a skeleton that is suited for agents who filter a stream, monitor a database, or perform any activity iteratively. Its significant events are *start* (accept an input, if necessary, and begin), *error*, *end of stream*, *accept* (accept an input, if necessary), *respond* (produce an answer), *more* (loop back to expecting more input). Here too, the application-specific computation takes place in the node labeled "Executing." The events *error*, *end of stream*, and *respond* are immediate, and all other events are flexible, and *start* is in addition triggerable. ▮

2.2 Architecture of the Service

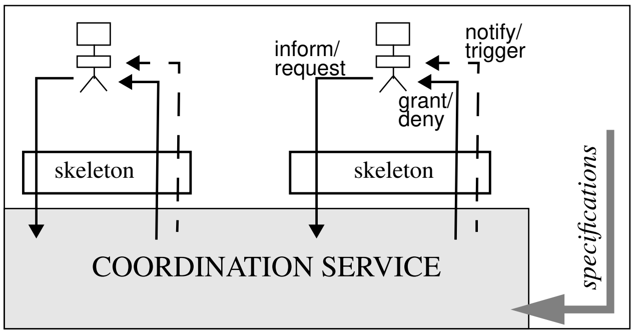


Figure 3: The Coordination Service, Logically

Figure 3 shows how the service interacts with agents. The service is customized through the specifications. The agents *inform* it of the immediate events that have happened unilaterally and *request* permission for inevitable and flexible events, which it may control; The service *grants* or *denies* permissions, *notifies* the agents, or *triggers* more events. It can delay inevitable events; delay or deny flexible events; and trigger triggerable events.

Operation Although we logically view the service as lying beneath each agent, it is *not* a separate entity in the implementation! It is distributed across the significant events of each agent. The following sections show how events exchange messages whose content and direction are automatically compiled. The interested reader may peek ahead at Figure 5 to see the effect.

The events are implemented as objects with threads that communicate asynchronously with other events, and respond to `attempt` requests from their associated skeletons. The skeletons can often be implemented procedurally. The skeleton communicates with its associated event objects. It can request permission to proceed with flexible and inevitable events, making its agent wait until a response is received from the event object. In the case of flexible events, it must respect the response `deny`; for inevitable events, this response cannot occur. For immediate events, the skeleton must process signals from its agent and notify the associated event object, which is responsible for notifying the other pertinent events.

Steps Toward a Methodology Although the development of a formal methodology based on our coordination service is still pending, we have gleaned some useful ideas. For freshly constructed agents, the skeletons can be embodied in the agents. However, in general, that is not the case. Since our service is purely qualitative, any necessary reasoning on intermediate results for decision-making is carried out through application-specific subtasks, for which skeletons can also be provided.

In imperative programming languages, we must have separate procedures to serve as calls from the different triggerable events. The flexible and inevitable events can be defined in terms of callbacks, but we find a thread-based style of programming more natural for the asynchronous, distributed environments of interest. In languages that support procedural reflection, we can use a context return mechanism to process continuations, thereby achieving the effect of the suspension and resumption of an agent more naturally.

3 Coordination Specification Language

We formalize interactions in an event-based linear temporal logic. $\mathcal{I}$, our specification language, is propositional logic augmented with the *before* $(\cdot)$ temporal operator. The literals denote event types, and can have parameters. A literal with all constant parameters denotes an event instance. *Before* is formally a dual of the more conventional "until" operator. Crucially, $\mathcal{I}$ can express a remarkable variety of interactions, yet be compiled and executed in a distributed manner.

Our key motivation in keeping the specification language separate from the internal language is that specifications should describe entire computations, without regard to the details scheduling. But the execution mechanism should be able to allow or trigger events on the basis of whatever information is available at the given stage of the computation.

3.1 Syntax

The syntax of $\mathcal{I}$ follows. Ξ includes all event literals (with constant or variable parameters); $\Gamma \subseteq \Xi$ contains only constant literals. A *dependency* is an expression in $\mathcal{I}$. A *workflow*, $\mathcal{W}$, is a set of dependencies.

Syntax 1 $\Xi \subseteq \mathcal{I}$

Syntax 2 $I_1, I_2 \in \mathcal{I} \Rightarrow I_1 \vee I_2, I_1 \wedge I_2, I_1 \cdot I_2 \in \mathcal{I}$

To simplify the notation, we assume that $\cdot$ (before) has precedence over $\vee$ and $\wedge$, and $\wedge$ has precedence over $\vee$.

3.2 Semantics

The semantics of $\mathcal{I}$ can be elegantly given in an algebraic style as below. However, Appendix **??** shows that we can easily give an equivalent, and more conventional, semantics based on a first-order metalanguage.

Like for many process algebras, our formal semantics is based on traces, that is, sequences of events. Our universe is $\mathbf{U}_{\mathcal{I}}$, which contains all consistent traces involving event instances from Γ. Consistent traces are those in which an event instance and its complement do not occur, and in which event instances are not repeated. $[\![\,]\!] : \mathcal{I} \mapsto \wp(\mathbf{U}_{\mathcal{I}})$ gives the denotation of each member of $\mathcal{I}$. The specifications in $\mathcal{I}$ select the acceptable traces—specifying I means that the service may accept any trace in $[\![I]\!]$.

Let constant parameters be written as c_i etc.; variables as v_i etc.; and either variety as p_i etc. $e[c_1 \ldots c_m]$ means that e occurs appropriately instantiated.

Semantics 1 $[\![e[c_1 \ldots c_m]]\!] = \{\tau \in \mathbf{U}_{\mathcal{I}} : e[c_1 \ldots c_m] \text{ occurs on } \tau\}$

$I(v)$ refers to an expression free in variable v. $I(v ::= c)$ refers to the expression obtained from $I(v)$ by substituting every occurrence of v by c. Variable parameters are effectively universally quantified by:

Semantics 2 $[\![I(v)]\!] = \bigcap_{c \in \mathcal{C}} [\![I(v ::= c)]\!]$

$I_1 \vee I_2$ means that either I_1 or I_2 is satisfied. $I_1 \wedge I_2$ means that both I_1 and I_2 are satisfied (in any interleaving). $I_1 \cdot I_2$ means that I_1 is satisfied before I_2 (thus both are satisfied).

Semantics 3 $[\![I_1 \vee I_2]\!] = [\![I_1]\!] \cup [\![I_2]\!]$

Semantics 4 $[\![I_1 \wedge I_2]\!] = [\![I_1]\!] \cap [\![I_2]\!]$

Semantics 5 $[\![I_1 \cdot I_2]\!] = \{\tau_1\tau_2 \in \mathbf{U}_{\mathcal{I}} : \tau_1 \in [\![I_1]\!] \text{ and } \tau_2 \in [\![I_2]\!]\}$

Section 5.2 presents a set of equations, which enable symbolic reasoning on $\mathcal{I}$ to determine when a certain event may be permitted, prevented, or triggered.

3.3 Event Complementation

The treatment of complementary events in our approach is novel in some respects and merits additional explanation.

$\overline{e}$ refers to the complement of e. Complemented literals are included in Ξ and, therefore, need no separate syntax or semantics rule. Since $[\![\,]\!]$ yields sets of traces, complementation is stronger than negation in other temporal logics. Intuitively, $\overline{e}[c_1 \ldots c_m]$ is established only when it is definite that $e[c_1 \ldots c_m]$ will never occur. This is crucial in the correct functioning of our service. The weaker, more common, notion of negation is of course also required for scheduling. However, we do not include it in the specification language, only in the internal temporal language for taking scheduling decisions, which is introduced in Section 5.

4 Specifying Coordination

4.1 Coordination Relationships

Our language allows a variety of relationships to be captured. We now consider some common examples. The events all carry parameter tuples, but we don't show them below to reduce clutter. Some of these relationships are then applied on our running example. (Notice that some of the relationships involve coordinating multiple events.)

	Name	**Description**	**Formal notation**
R1	e is required by f	If f occurs, e must occur before or after f	$e \vee \overline{f}$
R2	e disables f	If e occurs, then f must occur before e	$\overline{e} \vee \overline{f} \vee f \cdot e$
R3	e feeds or enables f	f requires e to occur before	$e \cdot f \vee \overline{f}$
R4	e conditionally feeds f	If e occurs, it feeds f	$\overline{e} \vee e \cdot f \vee \overline{f}$
R5	Guaranteeing e enables f	f can occur only if e has occurred or will occur	$e \wedge f \vee \overline{e} \wedge \overline{f}$
R6	e initiates f	f occurs iff e precedes it	$\overline{e} \wedge \overline{f} \vee e \cdot f$
R7	e and f jointly require g	If e and f occur in any order, then g must also occur (in any order)	$\overline{e} \vee \overline{f} \vee g$
R8	g compensates for e failing f	if e happens and f doesn't, then perform g	$(\overline{e} \vee f \vee g) \wedge (\overline{g} \vee e) \wedge (\overline{g} \vee \overline{f})$
R9	$e_1, \ldots, e_n$ disjunctively yield f	f occurs iff any of the e_i do	$(\overline{f} \vee e_1 \vee \ldots \vee e_n) \wedge (f \vee \overline{e_1} \wedge \ldots \wedge \overline{e_n})$
R10	$e_1, \ldots, e_n$ conjunctively yield f	f occurs iff all of the e_i do	$(\overline{f} \vee e_1 \wedge \ldots \wedge e_n) \wedge (f \vee \overline{e_1} \vee \ldots \vee \overline{e_n})$

Table 1: Example Relationships

The above (and similar) relationships can capture different coordination requirements. For example, R3 suggests an enabling condition or a data flow from e to f. R8 captures requirements such as that if an agent does something (e), but another agent does not match it with something else (f), then a third agent can perform g. This is typical pattern of coordination in information applications with data updates, where g corresponds to an action to restore the consistency of the information (potentially) violated by the success of e and the failure of f. Hence the name *compensation*.

R9 and R10 are related to Goldman's *generation* relation among actions [22]. In these, the event f arises as a consequence of the disjunction or conjunction of the other events. Thus, f and $\overline{f}$ should be modeled as triggerable events, so the coordination service can decide which to execute. Other variants, which involve sequences or other combinations of the e_i, can also be formalized.

Example 5 formalizes Example 2. Here, x denotes the unique id of the information search through which the various instantiations of the relevant computations in the agents are tied together. *tup* is a variable bound to a tuple. *sup* is a variable bound to a supplier. v indicates the availability of the desired valve. Subscripts s, r, and a respectively denote *start*, *respond*, and *accept* events.

Example 5 Assume five types of agents corresponding to different functions: B to the bridge, D a directory lookup, Q the main queries (there is one agent for each supplier), M to map over the responses of D, and F to fuse the results. M takes a single input at start. Thus, B, D, and the Qs are information agents, and M and F are task agents [54].

Assume all agents except M have skeletons as in Figure 1 with D returning a tuple response containing a list of suppliers and Q being invoked on each of its members. M has a skeleton as in Figure 2. M is started with tuple *tup* of suppliers, and initiates a query to each supplier agent. This yields:

D1. $B_r[x\ port]$ feeds $D_s[x\ port]$

D2. $D_r[x\ tup]$ feeds $M_s[x\ tup]$

D3. $M_r[x\ sup]$ initiates $Q_s^{sup}[x]$

D4. $M_{eof}[x]$ initiates $F_s[x]$

D5. $Q_r^{sup}[x\ v]$ conditionally feeds $F_s[x]$.

4.2 Modeling Other Approaches

Coordination has drawn much attention in the research community. We consider some important related approaches next.

4.2.1 Contract Net Protocol

Our approach applies well to higher-level coordination protocols, such as the contract net protocol (CNP) of Davis & Smith [11]. Briefly, the CNP begins when the manager sends out a request for proposals (RFP); some potential contractors respond with bids; the manager accepts one of the bids and awards the task. Much of the required reasoning is application-specific and performed autonomously be the agents, for example, who to send the RFP to, whether to bid, and how to evaluate bids.

Example 6 Since all agents can play the role of manager or contractor, we assume that all have the same significant events. Any agent because of internal reasons can perform the $A_{rfp}[m\ t\ c]$ event. Here m is the manager id, t is the task id, and c is a potential contractor—there will be a separate event for each c. (Multicasts can also be captured.) This involves the following dependencies:

D6. $A_{rfp}[m\ t\ c\ \text{info}]$ initiates $A_{think}[c\ t\ m\ \text{info}]$

D7. $A_{bid}[c\ t\ m\ \text{bid}]$ conditionally feeds $A_{eval}[m\ t\ c\ \text{bid}]$

D8. $A_{award}[m\ t\ c\ \text{task}]$ initiates $A_{work}[c\ t\ m\ \text{task}]$.

Dependency D6 means that the receiving agents reason about the RFP and autonomously decide whether to bid. If not, they exit the protocol. If they continue, dependency D7 kicks in. The manager now autonomously evaluates bids, leading to an award on one of them, which triggers the work, because of dependency D8. ▮

4.2.2 dMARS

Kinny & Georgeff propose a methodology for building multiagent systems using the dMARS tools [33]. They separate external and internal viewpoints. The former includes (a) a description of the agent classes, (b) their taxonomic structure and instances, (c) their interactions and control relationships, and (d) their communications. The latter includes a description of the beliefs, goals, and plans of each agent class. This approach has a number of good ideas that we shall not discuss here. However, a limitation is that the external and internal models are not closely related, and it is not clarified how the interactions and control relationships might be expressed. We show how our service complements this approach in the above respects.

The dMARS plan model includes a plan graph, which is a state transition diagram. A plan is *activate*d when its activation event occurs and its activation condition holds. A plan terminates with a *pass*, *fail*, or *abort* event. *Pass* and *fail* occur when the plan goes to completion; *abort* occurs when some exception is raised during execution. The internal details of plans are not of interest here, but their overall structure can be captured by the skeleton of Figure 4.

The lower part of Figure 4, up to the states "Aborted," "Failed," and "Passed" is extracted from Kinny & Georgeff's plan graph [33, Figure 3]. The other states and events are

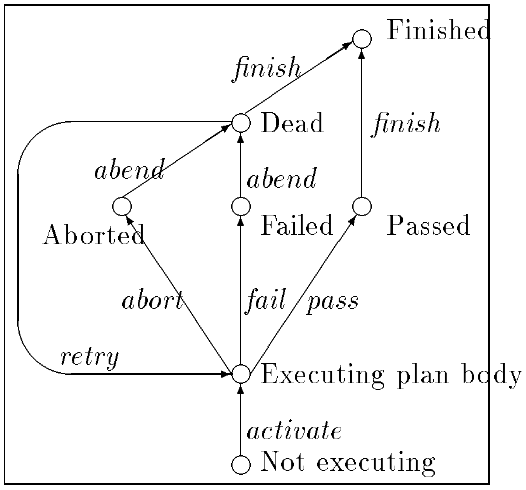


Figure 4: Skeleton for dMARS Plans

included based on their textual description of plans and how they may be coordinated. The state "Dead" is included as a convenience, since "Aborted" and "Failed" are often treated alike. *Abend* stands for "abnormal end." *Retry* is included, because retrying is implicit in the dMARS execution model, and can be disabled only through an explicit assertion of a *plan property*. The state "Finished" represents final termination of the plan after any number of iterations, and without any implication regarding success or failure. We assume that *activate* is triggerable, *pass*, *abend*, *finish*, and *fail* are inevitable, *retry* is flexible, and *abort* is immediate.

Name	**dMARS description**	**Our terminology**
Priority	P_1 is executed before P_2	P_1.*finish* enables P_2.*activate*
Precedence	P_2 is attempted only if P_1 was attempted and didn't pass	P_1.*abend* enables P_2.*activate*
No retry	Should not be retried on failure	P.*fail* disables P.*retry*

Table 2: dMARS Plan Properties

Kinny & Georgeff define three *plan properties*, which determine how plans are executed, especially in concurrence with other plans. P_1 has *priority* over P_2 if both execute, but P_1 goes before P_2. P_1 has *precedence* over P_2 if P_2 executes only if P_1 doesn't succeed. P is not retried if it ends upon failure. Table 2 shows how the dMARS properties can be captured in our approach. This implicitly justifies the appropriateness of the skeleton of Figure 4.

dMARS plans are internal to an agent, and are not related to the external view. However, by associating the above skeleton with them, we show how they may be related to the external view, and coordinated with the plans of other agents, just as easily as the other plans of the same agent. Indeed, this accords naturally with the the *aggregation* of agents, which is supported by dMARS, It can be viewed as a declarative means to capture the *interaction model*, which is included in the dMARS methodology, but appears primarily to be procedurally specified.

4.2.3 STEAM

STEAM is an architecture for teamwork by agents [55]. STEAM offers abstractions for joint intentions and teams, which are beyond our present scope. However, STEAM uses some coordination abstractions, which are relevant here. One of STEAM's features is the specification of team plan operators in terms of *role operators*—that is, plan operators for member agents. Three *role-monitoring constraints* are defined, through which STEAM can infer the potential achievability of a team operator. If a team operator becomes unachievable because of a role-monitoring failure, it can be repaired by examining the roles that caused the failure.

We show how STEAM's constraints can be expressed and reasoned with in our notation. We simplify the presentation for expository ease. In the following, O refers to a team operator, and o_i to role operators. In our formulation, we introduce events for O and the o_i as well as their complements. The events O and $\overline{O}$ are triggerable, and capture the derived behavior of the team, which is effectively treated as an agent in its own right. Table 3 shows a translation from STEAM's notation to ours. Example 7 shows how the example of reasoning given by him can be realized in our approach.

Name	STEAM notation	Our terminology
AND-combination	$O \Longleftrightarrow \bigwedge_{i=1}^{n} o_i$	$o_1, \ldots, o_n$ conjunctively yield O
OR-combination	$O \Longleftrightarrow \bigvee_{i=1}^{n} o_i$	$o_1, \ldots, o_n$ disjunctively yield O
Role-dependency	$o_i \Longrightarrow o_j$	o_j is required by o_i

Table 3: STEAM Constraints

Example 7 Given the constraint $T_1 \triangleq (O \Longrightarrow (o_i \vee o_j) \wedge (o_i \Rightarrow o_j))$, STEAM can infer that the nonperformance of o_j makes O unachievable, whereas the nonperformance of o_i does not. We can capture this reasoning through residuation. In our notation, T_1 is expressed as $T_2 \triangleq ((\overline{O} \vee o_i \vee o_j) \wedge (O \vee (\overline{o_i} \wedge \overline{o_j})) \wedge (\overline{o_i} \vee o_j))$. We can readily check that

- $T_2/\overline{o_j} = (\overline{O} \wedge \overline{o_i})$, which means that O can no longer occur.
- $T_2/o_j = O$, which means that O occurs.
- $T_2/o_i = (O \wedge o_j)$, which means that O must occur.
- $T_2/\overline{o_i} = ((\overline{O} \vee o_j) \wedge (O \vee \overline{o_j}))$. Consequently, (a) $(T_2/\overline{o_i})/o_j = O$, which means that O must occur after $\overline{o_i}$ and o_j, and (b) $(T_2/\overline{o_i})/\overline{o_j} = \overline{O}$, which means that O must not occur after $\overline{o_i}$ and $\overline{o_j}$.

Thus satisfying T_2 is incompatible with performing $\overline{o_j}$. Performing o_i or $\overline{o_i}$ has no effect—in either case, o_j must eventually be performed if O is to be realized. ∎

We can represent not only the above constraints, but also more complex ones involving temporal ordering relationships, which currently STEAM does not handle. Further, we can reason with such constraints in the same manner as above. Having a rigorous semantics helps in clarifying certain concepts. For example, the nonoccurrence of o_2 would cause a failure of the role-dependency $o_1 \Rightarrow o_2$ only if o_1 occurs, which means that either o_1 has already occurred or is inevitable or immediate. These attributes do not arise in the original STEAM analysis.

5 Coordination Scheduling

One of our requirements is that the coordination service be as distributed as possible, which presupposes that the events take decisions based on local information. Our approach requires (a) determining the conditions, that is, *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.

5.1 Temporal Logic for Internal Reasoning

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 taken 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. The guards are compiled from the stated dependencies; in practice, they are quite succinct.

$\mathcal{T}$, the language in which the guards are expressed, 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. $E \cdot F$ means that F has occurred preceded by E. For simplicity, we assume the following binding precedence (in decreasing order): $\neg$; $\cdot$; $\Box$ and $\Diamond$; $\wedge$; $\vee$.

Syntax 3 $\Gamma \subseteq \mathcal{T}$

Syntax 4 $E, F \in \mathcal{T} \Rightarrow E \vee F,\ E \wedge F,\ E \cdot F,\ \Box E,\ \Diamond E,\ \neg E \in \mathcal{T}$

The semantics of $\mathcal{T}$ is given with respect to a trace (as for $\mathcal{I}$) *and* an index into that trace (unlike for $\mathcal{I}$). This semantics characterizes progress along a given computation to determine the decision on each event. It has important differences from traditional linear temporal logics [15]. 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, that is, intervals, rather than a single index. For $0 \leq i \leq k$, $u \models_{i,k} E$ means that E is satisfied over the subsequence of u

between i and k. For $k \geq 0$, $u \models_k E$ means that E is satisfied on u at index k—implicitly, i is set to 0.

A trace, u, is *maximal* iff for each event, either the event or its complement occurs on u. $\mathbf{U}_{\mathcal{T}} \triangleq$ the set of maximal traces. We assume $\Xi \neq \emptyset$; hence, $\Gamma \neq \emptyset$. Semantics 18, 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 22 introduces a nonzero first index. Semantics 19 and 22 capture the dependence of an expression on the immediate past, bounded by the first index of the semantic definition. Semantics 20, 21, 23, 24, 25, and 26 are as in traditional semantics. Semantics 25 and 26 involve looking into the future. Semantics 19 implicitly treats events as being in the scope of a past-time operator. Consequently, Semantics 24 interprets $\neg$ as *not yet*.

Semantics 6 $u \models_i E$ iff $u \models_{0,i} E$

Semantics 7 $u \models_{i,k} f$ iff $(\exists j : i \leq j \leq k$ and $u_j = f)$, where $f \in \Gamma$

Semantics 8 $u \models_{i,k} E \vee F$ iff $u \models_{i,k} E$ or $u \models_{i,k} F$

Semantics 9 $u \models_{i,k} E \wedge F$ iff $u \models_{i,k} E$ and $u \models_{i,k} F$

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

Semantics 11 $u \models_{i,k} \top$

Semantics 12 $u \models_{i,k} \neg E$ iff $u \not\models_{i,k} E$

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

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

5.2 Deriving Guards from Specifications

Since the guards must yield precisely the computations that are allowed by the given dependencies, a natural intuition is that the guard of an event covers each computation in the denotation of the specified dependency. For each computation, the guard captures how far that computation ought to have progressed when the guarded event occurs, and what obligations would remain to realize that computation. We term this reasoning *residuation* and notate it by an operator $/ : \mathcal{I} \times \Xi \mapsto \mathcal{I}$, which is not in $\mathcal{I}$ or $\mathcal{T}$. Roughly, given a dependency D and event e, D/e gives the residual or "remnant" of D after e occurs. Although $/$ does not occur in our formal language, we extend the notation to define its denotation as follows.

Definition 1 $\nu \in [\![D/e]\!]$ iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_{\mathcal{I}} \Rightarrow \upsilon\nu \in [\![D]\!]))$

Interestingly, $/$ can be computed symbolically. We propose a set of equations using which the "residual" of any dependency with respect to an event can be computed. These equations require that the expressions be in a form such that there is no $\wedge$ or $\vee$ in the scope of the $\cdot$ (CNF is one such form). Such a representation exists, because of the distribution laws validated by the semantics of $\mathcal{I}$. Because of this restriction, in the equations below, D is a sequence expression, and E is a sequence expression or $\top$ (the latter allows us to treat an atom as a sequence, using $f \equiv f \cdot \top$). $\Gamma_E \triangleq \{e : e$ or $\overline{e}$ occurs in $E\}$. (We define $\overline{\overline{e}}$ as e.) We set the denotation of any sequence $e_1 \cdot \ldots \cdot e_n$ in which (for $i \neq j$) $e_i = e_j$ or $e_i = \overline{e_j}$ to the empty set; we assume such sequences are reduced to 0.

Equation 1 $0/e = 0$
Equation 2 $\top/e = \top$
Equation 3 $(E_1 \wedge E_2)/e = ((E_1/e) \wedge (E_2/e))$
Equation 4 $(E_1 \vee E_2)/e = (E_1/e \vee E_2/e)$
Equation 5 $(e \cdot E)/e = E$, if $e \notin \Gamma_E$
Equation 6 $D/e = D$, if $e \notin \Gamma_D$
Equation 7 $(e' \cdot E)/e = 0$, if $e \in \Gamma_E$
Equation 8 $(\overline{e} \cdot E)/e = 0$

The above equations yield important independence and modularity properties, and closed-form answers for symbolic reasoning. For example, / distributes over $\vee$ and $\wedge$, and expressions that do not mention a given event have no effect on it.

Interestingly, these equations are not sound in our formal model (or any simple variation thereof, for example, those considered in [43])! This is because the models include legal as well as illegal traces. When the illegal traces are eliminated, our equations are indeed sound. Eliminating the illegal traces takes the form of a quotient construction, which identifies expressions that are equivalent with respect to the desired coordination of the agents. The equations embody a *combination* of a notion of change—of the system evolving because of events—and a notion of the system's knowledge—its state for scheduling decisions.

Theorem 1 Equations 9–16 are sound. ∎

We define guards as below. These cases cover all the syntactic possibilities of $\mathcal{I}$. Importantly, our definition distributes over $\wedge$ and $\vee$: using our normalization requirement, each sequence subexpression can be treated separately. Thus the guards are quite succinct for the common cases, such as the relationships of Section 4.

Definition 2 The guards are given by the operator $\mathsf{G} : \mathcal{I} \times \Xi \mapsto \mathcal{T}$:
(a) $\mathsf{G}(D_1 \vee D_2, e) \triangleq \mathsf{G}(D_1, e) \vee \mathsf{G}(D_2, e)$
(b) $\mathsf{G}(D_1 \wedge D_2, e) \triangleq \mathsf{G}(D_1, e) \wedge \mathsf{G}(D_2, e)$
(c) $\mathsf{G}(e_1 \cdot \ldots \cdot e_i \cdot \ldots \cdot e_n, e_i) \triangleq \Box e_1 \wedge \ldots \wedge \Box e_{i-1} \wedge \Diamond(e_{i+1} \cdot \ldots \cdot e_n)$
(d) $\mathsf{G}(e_1 \cdot \ldots \cdot e_n, e) \triangleq \Diamond(e_1 \cdot \ldots \cdot e_n)$, if $\{e, \overline{e}\} \not\subseteq \{e_1, \overline{e_1}, \ldots, e_n, \overline{e_n}\}$
(e) $\mathsf{G}(e_1 \cdot \ldots \cdot e_i \cdot \ldots \cdot e_n, \overline{e_i}) \triangleq 0$
(f) $\mathsf{G}(0, e) \triangleq 0$
(g) $\mathsf{G}(\top, e) \triangleq \top$.

Example 8 We compute the guards for the events in R2 as follows:

- $\mathsf{G}(\mathrm{R2}, e) = \Diamond\overline{f} \vee \Box f$
- $\mathsf{G}(\mathrm{R2}, \overline{e}) = \top$
- $\mathsf{G}(\mathrm{R2}, \overline{f}) = \top$
- $\mathsf{G}(\mathrm{R2}, f) = (\Diamond\overline{e} \vee (\neg e \wedge \Diamond e))$, which equals $\neg e$ under the semantics of $\mathcal{T}$.

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

5.3 Scheduling with Guards

Execution with guards is straightforward. It involves the following main steps:

1. 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
   - $\top$: e is executed
   - 0: e is rejected
   - else: e is made to wait
2. Whenever an event e occurs, a notification $\Box e$ is sent to each pertinent event f, whose guards are updated accordingly. If f was waiting, it is handled according to the value of its new guard as described in step 1.

We show below how this procedure must be refined when events have mutual waits or race conditions. Example 9 illustrates this procedure. It assumes that both events are *flexible*.

Example 9 Using the guards from Example 8, if f is attempted and e has not already happened, f's guard evaluates to $\top$. Consequently, f is allowed and a notification $\Box f$ is sent to e (and $\overline{e}$). Upon receipt of this notification, e's guard is simplified from $\Diamond\overline{f} \vee \Box f$ to $\top$. Now if e is attempted, it can happen immediately.

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

Example 10 (Figure 5) Following Example 5, when the search is initiated, B is invoked. It returns with a port name, which is used in D, which could already have been started, but would be waiting. D produces a tuple of suppliers. For each of these a separate query is initiated, which goes to the given supplier's agent. These queries execute concurrently. However, there is only one instance (per search) of the fuser, which waits until all the queries terminate. ▮

5.4 Correctness of Guard Processing

Abstractly, given a workflow $\mathcal{W}$, our evaluation technique *generates* traces as follows. We sloppily write generation as $\mathcal{W} \rightsquigarrow u$ and define it as $(\forall i : i \leq |u| \Rightarrow \mathcal{W} \rightsquigarrow_i u)$, where $\mathcal{W} \rightsquigarrow u_i \triangleq (\forall j : 1 \leq j \leq i \Rightarrow u \models_{j-1} \mathsf{G}(\mathcal{W}, u_j))$. From this we obtain the following correctness result, which states that precisely those traces are generated that are in the denotation of the stated dependencies. A rigorous formalization is available in [53].

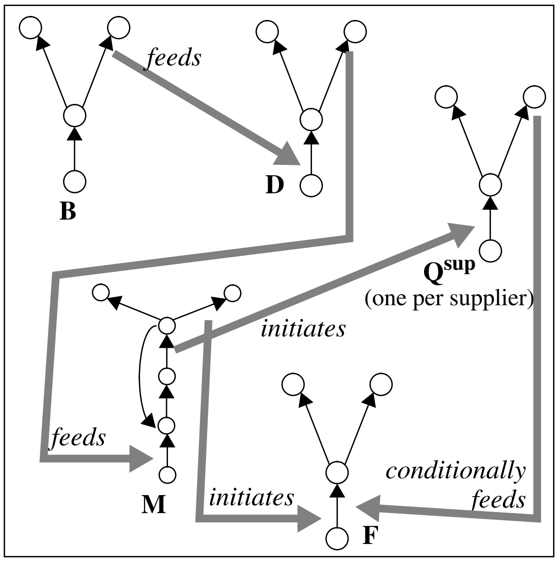


Figure 5: Search of Example 5

Theorem 2 $\mathcal{W} \rightsquigarrow u$ iff $(\forall D \in \mathcal{W} : u \models D)$. ∎

The above result establishes correctness abstractly without regard to how it is determined whether $u \models_{j-1} \mathsf{G}(\mathcal{W}, u_j)$ for a trace u and an index j. Our approach computes $\models_{j-1}$ incrementally as much as possible. Events produce notifications, which are incrementally assimilated by the recipients, leading to simplification of their guards. The operator $\div$ captures the assimilation process. This operator embodies a set of "proof rules" to reduce guards when notifications are received.

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 involve $\neg$ expressions, which are not ordinarily sent as messages. These are discussed as prohibitory relationships below. Notifications of the form $\Diamond e$ arise in promissory relationships, as discussed below.

Theorem 3 means that the operator $\div$ preserves the truth of the original guards. The receipt of a message, no matter how delayed, cannot cause any violation. In other words, no spurious traces are generated by our assimilation process. We can also show that all of the original traces are still generated.

Theorem 3 $(\exists k \leq j : u \models_k M$ and $u \models_j G \div M) \Rightarrow u \models_j G$. ∎

5.5 Mutual Constraints Among Events

By the above, if the events send the appropriate notifications, we can compute the semantics of $\mathcal{T}$ incrementally. But in some situations potential race conditions and deadlocks can

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
$\Box(e_1 \cdot e_2)$	$\Box e_1 \wedge \neg e_2$	$\Box e_2$
$\Box(e_1 \cdot e_2)$	$\Box e_2 \wedge \neg e_1$	0
$\Box(e_1 \cdot e_2)$	$\Box \overline{e_i}$ or $\Diamond \overline{e_i}$, $i \in \{1, 2\}$	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
$\Diamond(e_1 \cdot e_2)$	$\Box \overline{e_i}$, $i \in \{1, 2\}$	0
$\neg e$	$\Box e$	0
$\neg \overline{e}$	$\Box e$ or $\Diamond e$	$\top$
G	M	G, otherwise

Table 4: Assimilating Messages

arise. To ensure that the necessary information flows to an event when needed, the execution mechanism should be more astute in terms of recognizing and resolving mutual constraints among events. This reasoning is essentially encoded in terms of heuristic graph-based reasoning. Although we believe we can handle the interesting cases, pathological cases can arise that cannot be easily handled without assistance from a human designer.

5.5.1 Prohibitory Relationships

During guard evaluation for an event e, subexpressions 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—essentially to serialize the execution where necessary. Figure 6 diagrams the message flow in this case.

Example 11 Following Example 8, f should not occur unless we can be sure that e has not occurred. ▮

This is a *prohibitory* relationship between events, since e's occurrence can possibly disable f (depending on the rest of the guard of f). Prohibitory messages can be avoided if the disabler is made responsible for preserving the correct order of execution—in our approach this can always be done, except when the disabler is an immediate event.

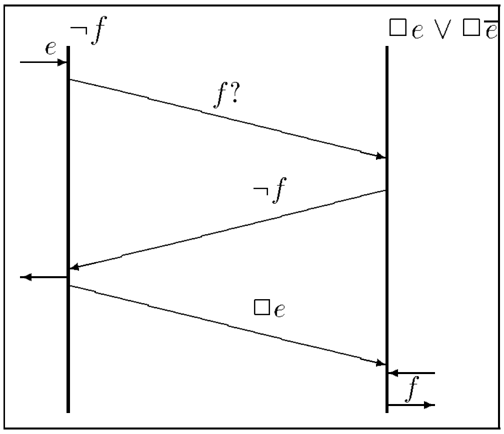


Figure 6: Prohibitory Message Flows

5.5.2 Promissory Relationships

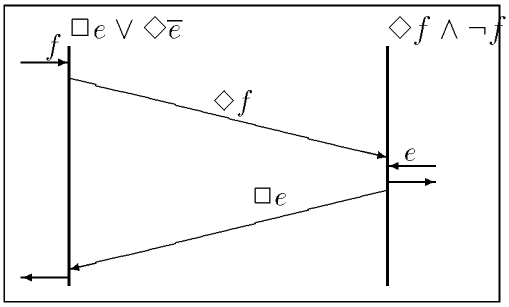


Figure 7: Promissory Message Flows

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 12 Consider $\mathcal{W} = \{\text{R1}, \text{R2}\}$. $\mathsf{G}(\mathcal{W}, e) = \Diamond f \wedge \neg f$ and $\mathsf{G}(\mathcal{W}, f) = \Box e \vee \Diamond \overline{e}$. Roughly, this means that e waits for $\Diamond f$, while f waits for $\Box e$. ▮

The guards given in Example 12 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. We emphasize that the term *promising* is used in a narrow technical sense here; the events have no notion of social commitments! Figure 7 diagrams the message flow in this case.

The correctness of these and other strategies for resolving mutual constraints can be established by recourse to the formal semantics of $\mathcal{I}$ and $\mathcal{T}$, and an associated formalization of the execution process.

5.6 The Effect of Event Classes

Our definitions apply directly to flexible events. For inevitable and immediate events, the dependencies must be strengthened to ensure that an inevitable event is never denied and an immediate event is neither denied nor delayed. The strengthened dependencies ought to be used in the reasoning about coordination. Table 5 shows revised representations for some of the relationships in Table 1.

	Name	**Description**	**Formal notation**
R11	e is required by f	If f occurs, e must occur before or after f	$e \vee \overline{f}$
R12	e disables f	If e occurs, then f must occur before e	$\overline{e} \vee \overline{f} \vee f \cdot e$
R13	e feeds or enables f	f requires e to occur before	$e \cdot f \vee \overline{f}$
R14	e conditionally feeds f	If e occurs, it feeds f	$\overline{e} \vee e \cdot f \vee \overline{f}$
R15	Guaranteeing e enables f	f can occur only if e has occurred or will occur	$e \wedge f \vee \overline{e} \wedge \overline{f}$
R16	e initiates f	f occurs iff e precedes it	$\overline{e} \wedge \overline{f} \vee e \cdot f$
R17	e and f jointly require g	If e and f occur in any order, then g must also occur (in any order)	$\overline{e} \vee \overline{f} \vee g$
R18	g compensates for e failing f	if e happens and f doesn't, then perform g	$(\overline{e} \vee f \vee g) \wedge (\overline{g} \vee e) \wedge (\overline{g} \vee \overline{f})$
R19	$e_1, \ldots, e_n$ disjunctively yield f	f occurs iff any of the e_i do	$(\overline{f} \vee e_1 \vee \ldots \vee e_n) \wedge (f \vee \overline{e_1} \wedge \ldots \wedge \overline{e_n})$
R20	$e_1, \ldots, e_n$ conjunctively yield f	f occurs iff all of the e_i do	$(\overline{f} \vee e_1 \wedge \ldots \wedge e_n) \wedge (f \vee \overline{e_1} \vee \ldots \vee \overline{e_n})$

Table 5: Example Relationships Revised

The guard on an immediate event is set to $\top$ by fiat, so that such an event may be allowed immediately, that is, whenever the agent chooses to perform it. 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.

Triggerable events do not affect the guards, only their processing. A naive approach will cause e to wait forever if f is triggerable. However, we can proactively execute a triggerable event whenever its guard is $\top$. This typically needs to be checked at startup and whenever a message is received. It is the specifier's obligation to state sufficient dependencies so that a triggerable event is no erroneously executed.

6 Discussion

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

Presently, the specifications are given when the multiagent system is constructed. If the specifications do not conflict with the autonomy of the agents, then they can be executed in a distributed manner. Determining the coordination requirements on the fly would be an important extension, and would be necessary, for example, when the coordination requirements emerge from the agents' social commitments [51].

We implemented our approach in an actor programming language. We are now reimplementing it with enhancements in Java. One of the enhancements is being able to switch between TCP/IP and CORBA for the underlying functionality. CORBA is important, because it is becoming a de facto standard for lower-level functionality in distributed systems. It provides an event service with notifications and triggers [47], but not a coordination service of the sort we described.

Literature The relevant previous tools for developing multiagent systems are either not formal, are centralized, or violate the autonomy of agents. AgenTalk [37] gives a programming environment, but no formal semantics. Kabanza adapts a traditional temporal logic for synchronizing agent plans; his approach has a centralized scheduler and violates autonomy by requiring full knowledge of, and modifying, the agents' plans [31]. Traditional temporal logic approaches preclude encapsulation of the component computations as agents; they do not accommodate the notion of admissibility, which captures the knowledge of the scheduler; they (in the case of databases) are limited to single-shot transactions and not applicable to arbitrary, nonterminating, complex computations that characterize agents.

Huhns et al. [29] and Sycara & Zeng [54] articulate many of our conceptual intuitions, including the ultimate necessity of multiagent, versus single-agent, approaches. They show how agents need to be coordinated to collectively search or manage information in open environments. Oates *et al.* propose an approach for planning searches [42]. However, their approach does not have an explicit representation of the structure of the search as here, and does not apply generically. Their search techniques are captured as sets of coordination relationships in our approach. von Martial [57] motivates relationships among plans. For example, the relationship *facilitates* can be captured in our framework by postulating events, such that the success of one plan enables an event in the other plan. In this manner, von

Martial's approach is higher level than ours. Decker & Lesser [13] present coordination algorithms in the generalized partial global planning framework. This work is both more and less ambitious than our work. It includes heuristics to reason about deadlines and coordination problems in various situations, but it does not provide a formal semantics.

Our approach complements the above, because they develop representations that are close to applications, whereas our approach focuses on the activity management infrastructure itself. However, our approach can help rigorously capture the intuitions of the above approaches.

High-level abstractions for agents have been intensively studied, for example, by Rao & Georgeff [44] and Singh [48]. Formal research on interactions among agents includes [25]. These approaches develop formal semantics (like the present approach) but do not give as precise an operational characterization (unlike the present approach). There has been much work on social abstractions for agents, for example, by Castelfranchi [8] and Gasser [19]. We believe that the present infrastructure will facilitate the development of a computational treatment of the social constructs by succinctly capturing the mechanics of possible interactions. Including mental and social abstractions into a generic executable system is an important open problem.

Executable Temporal Logics We consider Concurrent MetateM as a representative executable temporal logic [39], because it has a long intellectual history in temporal logics, and has been used to specify and build agents and multiagent systems [17, 60]. Concurrent MetateM enables the specification of the behavior of the various agents, somewhat like reactive systems in traditional logics of programs. This is a major difference from our approach, because we only formalize the coordination requirements in our logic, and leave the internal details to the implementors.

However, Concurrent MetateM bears a number of similarities with our approach. Both approaches use linear models with a bounded past and infinite future. Our $\cdot$ (before) operator is related to the *until* and *since* operators; however, Concurrent MetateM has a greater range of operators.

Rules in Concurrent MetateM are conditional formulae, whose antecedent refers only to the past, and whose consequent refers only to the present and future. Hence, the motto *declarative past and imperative future*. The guard execution mechanism of our approach is essentially a way to derive the rules. We can achieve this because we limit our formal specification language sufficiently to enable that derivation. Further, through the guards, we are able to execute our system in a distributed fashion. A further conceptual similarity is in the use of knowledge to give a semantics of Concurrent MetateM systems. Although our approach does not mention knowledge explicitly, the proof of soundness of our equations involves a quotient construction, which encodes the knowledge required to make correct scheduling decisions.

Wooldridge assumes that the agents execute in lock-step synchronization, and that they always choose the "right" path, which would lead to the stated rules being satisfied (assuming the rules are consistent) [60]. These are strong assumptions, which we believe are relaxed in our approach. Our agents can execute asynchronously, and must be serialized only when called for by the stated specifications. Further, mechanisms such as promising enable avoiding potential deadlocks, using not only th past but also the future in a careful manner. However, we make no claims of completeness of the promising mechanism.

Future Work We hope to explore a number of directions in future work. Within the coordination service itself, there is need for a richer metamodel or ontology of tasks, events, and their attributes. One of the challenges is to have a formal means for constructing complex events. Another is the development of a formal reasoning tool that would assist in creating and debugging coordination specifications. Lastly, there is pressing need for more formal software engineering methodologies for dealing with coordination than has yet been developed.

A major push will be in moving up the interaction-oriented programming (IOP) hierarchy, which involves the commitments and collaboration layers. We plan to build services for those functionalities integrated with the present coordination service.

We treat e and $\overline{e}$ symmetrically as events. We thus define a formal complement for each event, including those like *start*, whose non-occurrence constitutes their complement. This is crucial for eager scheduling—section 9 gives additional rationale. Traces that satisfy assumptions 1 and 2 are termed *legal*:

Assumption 1 Event instances and their complements are mutually exclusive.

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

Our universe set (and Example **??**) includes illegal traces, but these are eliminated from our formal model in section 7.1. The reader should assume legality everywhere.

As running examples, we use two dependencies due to [34], and related to the primitives in [2, 10, 24]. $e < f$ means that if both events e and f happen, then e precedes f; $e \rightarrow f$ means that if e occurs then f also occurs (before or after e). Again, these behave as propositional logic with a temporal flavor:

Example 13 Let $D_< \triangleq \overline{e} \vee \overline{f} \vee e \cdot f$. Let $\tau \in \mathbf{U}_\mathcal{E}$ satisfy $D_<$. If τ satisfies both e and f, then e and f occur on τ. Thus, neither $\overline{e}$ nor $\overline{f}$ can occur on τ. Hence, τ must satisfy $e \cdot f$, which requires that an initial part of τ satisfy e and the remainder satisfy f. That is, e must precede f on τ. ∎

Example 14 Let $D_\rightarrow \triangleq \overline{e} \vee f$. Let $\tau \in \mathbf{U}_\mathcal{E}$ satisfy $D_\rightarrow$. If τ satisfies e, then e occurs on τ. Thus, $\overline{e}$ cannot occur on τ. Hence, f must occur somewhere on τ. ∎

Example 15 Consider a workflow which attempts to *buy* an airline ticket and *book* a hotel for a traveler, such that the ticket is bought if and only if the hotel is booked. Mutual commit protocols cannot be executed, since the airline and hotel are different enterprises and their databases may not have a visible precommit state.

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

Note that D_2 explicitly orders c_{book} before c_{buy}, but D_1, D_3, and D_4 do not order any events. However, to be triggered, the events should have the attribute *triggerable*—introduced in [53]. The scheduler causes the events to occur when necessary, and may order them before or after other events as it sees fit. ▮

We now consider Leymann's *spheres of joint compensation (SOJCs)*, and show how to represent them [38]. An SOJC is a set of activities that either all commit or all are compensated. The compensation happens in the reverse order of the order in which the activities committed. We consider only the scheduling aspects of SOJCs.

Example 16 Let $\{T_1, \ldots, T_n\}$ along with their compensating activities $\{R_1, \ldots, R_n\}$ form an SOJC. Let each T_i and R_i be a transaction as in Figure ??. This SOJC can be captured by the following dependencies.

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

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

An advantage of a formal notation is that it forces one to decide on the desired meaning, i.e., a specific subset of the dependencies in Example 16.

6.1 Residuation in Event Algebra

Events are scheduled—by permitting or triggering—to satisfy all stated dependencies. A dependency is satisfied when a trace in its denotation is realized. We characterize the state of the scheduler by the traces it can allow. Initially, these are given by the stated dependencies. As events occur, the allowed traces get narrowed down.

Example 17 Consider Example 15. Suppose *buy* starts first. Then D_1 requires that *book* start sometime; the other dependencies have no effect, since they don't mention s_{buy} or $\overline{s_{buy}}$. Now if *buy* were to commit *next*, D_2 would be violated, because it states that *buy* can commit only after *book* has committed. Suppose *book* starts, thereby satisfying the remaining obligation from D_1. After *book* starts, D_2 would still prevent *buy* from committing. Eventually, if *book* commits, *buy* can commit thus completing the workflow (because of D_2 and D_4), or *buy* can abort thus causing *cancel* to be started (because of D_3). ∎

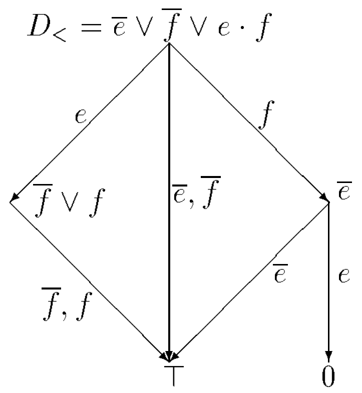


Figure 8: Scheduler states and transitions for $D_<$

Intuitively, two questions must be answered for each event under consideration: (a) can it happen now? and (b) what will remain to be done later? The answers can be determined from the stated dependencies and the history of the system. One can examine the traces allowed by the original dependencies, select those compatible with the actual history, and infer how to proceed. Importantly, our approach achieves this effect symbolically, *without* examining the traces.

Figure 8 shows how the states and transitions of the scheduler may be captured symbolically. The state labels give the corresponding obligations, and the transition labels name the different events. Roughly, an event that would make the scheduler obliged to 0 cannot occur.

Example 18 (Figure 8) If $\overline{e}$ or $\overline{f}$ happens, then $D_<$ is necessarily satisfied. If e happens, then either f or $\overline{f}$ can happen later. But if f happens, then only $\overline{e}$ must happen afterwards (e cannot be permitted any more, since that would mean f precedes e). ∎

The transitions of Figure 8 can be captured through an algebraic operator called *residuation*. This operator ($/ : \mathcal{E} \times \Gamma \mapsto \mathcal{E}$) is not in $\mathcal{E}$, since it is not used in the formulation of dependencies, only in their processing. Dependencies are *residuated* by the events that occur to yield simpler dependencies. The resultant dependencies implicitly contain the necessary history. This proves effective, because the representations typically are small and the processing is simple. We motivate our semantics for the / operator and then present an equational characterization of it.

The intuition embodied in Figure 8 is that to schedule a dependency D, the scheduler first schedules an event e and then schedules the residual dependency D/e. Our formal semantics reflects this intuition—to satisfy D, the scheduler can allow any of the traces in $[\![D]\!]$. Similarly, to schedule e, the scheduler can use any of the traces in $[\![e]\!]$. And, to schedule D/e, the scheduler can use any of the traces in $[\![D/e]\!]$. Thus the following must hold for correctness. (We intersect the set with $\mathbf{U}_{\mathcal{E}}$ to eliminate ill-formed traces such as $\upsilon\nu$, where υ is infinite.)

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

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

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

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

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

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

Semantics 15 $\nu \in [\![D/e]\!]$ iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_{\mathcal{E}} \Rightarrow \upsilon\nu \in [\![D]\!]))$

Theorem 4 Semantics 15 gives the most general solution to the above inequation. ∎

Theorem 4 states that given an occurrence of an event in a scheduler state corresponding to a set of traces, residuation yields the maximal set of traces which could correspond to the resulting state of the scheduler.

6.2 Symbolic Calculation of Residuals

Semantics 15 characterizes the evolution of the state of a scheduler, but offers no suggestions as how to determine the transitions. Fortunately, a set of equations exists using which the residual of any dependency can be computed.

We require that the expressions be in a form such that there is no $\wedge$ or $\vee$ in the scope of the $\cdot$. Such a representation of each expression is possible because the operators $\cdot$, $\vee$, and $\wedge$ distribute over each other. Further, Lemma 20 shows that we can replace all sequences by conjunctions of sequences of length 2, which is extremely convenient for processing, as described in [53]. We assume that all expressions are in this *Two-Sequence Form (TSF)*.

Definition 3 Γ_D gives the alphabet of expression D. $\Gamma_0 \triangleq \Gamma_\top \triangleq \emptyset$. $\Gamma_{E\wedge F} \triangleq \Gamma_{E\vee F} \triangleq \Gamma_{E\cdot F} \triangleq \Gamma_E \cup \Gamma_F$. $\Gamma_e \triangleq \Gamma_{\overline{e}} \triangleq \{e, \overline{e}\}$. Notice that $e \in \Gamma_D$ iff $\overline{e} \in \Gamma_D$.

Because of the restriction to TSF, in the equations below, D is a sequence expression, and E is a sequence expression or $\top$ (the latter case allows us to treat a single atom as a sequence, using $f \equiv f \cdot \top$).

Equation 9 $0/e \doteq 0$

Equation 10 $\top/e \doteq \top$

Equation 11 $(E_1 \wedge E_2)/e \doteq ((E_1/e) \wedge (E_2/e))$

Equation 12 $(E_1 \vee E_2)/e \doteq (E_1/e \vee E_2/e)$

Equation 13 $(e \cdot E)/e \doteq E$, if $e \notin \Gamma_E$

Equation 14 $D/e \doteq D$, if $e \notin \Gamma_D$

Equation 15 $(e' \cdot E)/e \doteq 0$, if $e \in \Gamma_E$

Equation 16 $(\overline{e} \cdot E)/e \doteq 0$

We use the operator $\doteq$ in the equations to highlight that it might not be interpreted simply as $\equiv$. Section 7.1 provides further explanation. The above equations are carefully designed to guide the reasoning of a scheduler when it is considering whether to allow an event e. They have some important properties, including

- dependencies not mentioning an event have no direct effect on it
- the reasoning with respect to different dependencies can be performed modularly
- the history of the scheduler need not be recorded.

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

Example 19 The reader can verify that the above equations yield all the transitions of Figure 8. ▌

Example 20 Consider Example 15 again. The initial state of the scheduler is given by $S_0 = D_1 \wedge D_2 \wedge D_3 \wedge D_4$. The scheduler can allow *buy* to start first, because the resulting state $S_1 = S_0/s_{buy}$ is consistent. This simplifies to $S_1 = s_{book} \wedge D_2 \wedge D_3 \wedge D_4$. The occurrence of c_{buy} next would leave the scheduler in state $s_{book} \wedge 0 \wedge \top \wedge \overline{s_{cancel}}$, which equals 0, i.e., is inconsistent. However, since $S_2 = S_1/s_{book} = \top \wedge D_2 \wedge D_3 \wedge D_4$ is consistent, s_{book} can occur in S_1. In S_2, *book* can commit, resulting in the state $S_3 = \top \wedge (\overline{c_{buy}} \vee c_{buy}) \wedge (c_{buy} \vee s_{cancel}) \wedge (\overline{c_{buy}} \vee \overline{s_{cancel}})$. S_3 allows either of $\overline{c_{buy}}$ and c_{buy} to occur. Since $S_3/c_{buy} = \overline{s_{cancel}}$, the workflow completes if *buy* commits. Since $S_3/\overline{c_{buy}} = s_{cancel}$, *cancel* must be started if *buy* aborts. ▌

An advantage of TSF is that the dependencies are independently stated, and a workflow $\mathcal{W}$ corresponds to one (albeit large) dependency which is the conjunction of dependencies in $\mathcal{W}$. No additional processing is required in putting the dependencies into an acceptable syntactic form for reasoning. Equation 11 enables the different dependencies to be residuated independently. Observation 5 means that events are independent of dependencies that do not mention them.

Observation 5 $E/f \doteq E$, if $f \not\in \Gamma_E$ ∎

We now give some additional results. We define $size(E)$ to be the number of nodes in the parse tree of E. The following results show that the equations yield simpler results than the input expression (Observation 6), are convergent (Lemma 8), and produce TSF expressions, which can be input to other equations (Observation 9).

Observation 6 If $D/e \doteq F$ is an equation, then $size(D) \geq size(F)$, $\Gamma_D \supseteq \Gamma_F$, and $\{e, \overline{e}\} \not\subseteq \Gamma_D$ ∎

Lemma 7 The number of operations to compute D/e is linear in $size(D)$. ∎

Lemma 8 For $D \in \mathcal{E}$ and $e \in \Gamma$, our equations yield exactly one expression in $\mathcal{E}$ for D/e. ∎

Observation 9 If D is in TSF, then D/e is in TSF. ∎

The scheduler can take a decision to accept, reject, or trigger an event only if *no* dependency is violated by that decision. By Observation 5, only dependencies mentioning an event are *directly* relevant in scheduling it. However, we also need to consider events that are caused by events that are caused by the given one, and so on—these might be involved in other dependencies.

There are several ways to apply the algebra. The relationship between the algebra and the scheduling algorithm is similar to that between a logic and proof strategies for it. For scheduling, the system accepts, rejects, or triggers events to determine a trace that satisfies all dependencies.

7 Correctness

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

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

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

The usual meaning of soundness is that whatever is provable is entailed (roughly, everything that is proved is true) and the usual meaning of completeness is that all the entailments are provable (roughly, everything true can be proved). These are captured as definitions 6 and 7. Since soundness can be proved piecemeal for each equation, we define it in terms of individual equations in definition 8. Definition 8 interprets $\doteq$ as $\equiv$, i.e., equivalence or equality of denotations. Lemma 10 follows.

Definition 6 A system of equations is sound iff $D/e \vdash F$ implies that $D/e \models F$

Definition 7 A system of equations is complete iff $D/e \models F$ implies that $D/e \vdash F$

Definition 8 An equation $D/e \doteq F$ is sound iff $D/e \doteq F$ implies that $D/e \models F$

Lemma 10 Equations 9–14 are sound. ∎

But what about Equations 15 and 16? The following lemma appears discouraging at first. However, it can be corrected when our assumptions are incorporated into the model.

Lemma 11 Equations 15 and 16 are *not* sound. ∎

7.1 Admissibility

Equations 15 and 16, along with Equation 14, are especially troublesome to prove sound in typical models. Individually, they can be satisfied in different models, but not together. The present model satisfies Equation 14, but if we had defined the universe set to include only the legal traces, then Equation 14 would not have been satisfied. We define a class of nonstandard, but intuitively natural, models based on what we term *admissibility*.

Recall that assumptions 1 and 2 of section **??** define legal traces. Admissible traces are legal traces that are also maximal, as defined below.

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

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

Admissibility is designed to formalize our special method of applying residuation to scheduling decisions. Consider Equation 15. Assume the scheduler is enforcing a dependency $D = (e' \cdot E)$, where E is a sequence expression mentioning e. Suppose the scheduler is taking a decision on e. We know that e' has not occurred yet, or it would have been residuated out already. Therefore, if we let e happen now, then either we must (a) prevent e', or (b) eventually let e' happen followed by an instance of e or $\overline{e}$. Option (a) clearly violates D, since all traces in $[\![D]\!]$ must mention e' (by Observation 52, see appendix A). Option (b) violates admissibility. Thus, assuming admissibility, we can prove Equation 15 to be sound. We formalize this argument next.

Technically, admissibility captures the context of evaluation, given by the state of the scheduler. Two expressions are interchangeable with respect to a set of admissible traces A if they allow exactly the same subset of A. As a result, two expressions with different denotations may be interchangeable in certain evaluation contexts.

Example 21 In general $e \not\equiv 0$. However, after e or $\overline{e}$ has occurred, another occurrence of e is impossible. Hence, e is effectively equivalent to 0. ∎

Definition 9 A is an admissible set iff $A = \mathbf{A}_\Theta$ for some alphabet Θ

Initially the admissible set is the entire set of admissible traces for Γ. After event e happens, the admissible set must be shrunk to exclude traces mentioning e or $\overline{e}$, because they cannot occur any more. Let A be an admissible set before e occurs. Then, after e, A must be replaced by $A \uparrow e$, where $A \uparrow e$ yields the resulting set of admissible traces. The operator $\uparrow$ thus abstractly characterizes execution.

Definition 10 $A \uparrow e \triangleq \{\nu : \langle e \rangle \nu \in A\}$

Observation 12 $A \uparrow e \cap [\![e]\!] = \emptyset$ and $A \uparrow e \cap [\![\overline{e}]\!] = \emptyset$ ∎

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

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

Example 22 Let $A = \mathbf{A}_\Theta$, such that $e \notin \Theta$. Then, $e \approx_A \overline{e}$. Also, $e \approx_A 0$. ∎

Lemma 13 For all admissible sets A, $\approx_A$ is an equivalence relation. ∎

We refer to $\approx_A$ as *adm-equivalence*. We now use it to define a quotient structure on our original models, which preserves all equalities, but only some of the inequalities. The quotient construction would be valid only if we can establish that the behavior of the scheduler is not affected by replacing one adm-equivalent expression for another. This is a crucial requirement upon which our whole technical development hinges. Theorem 14 states that if E and E' are adm-equivalent, then after event f occurs, their respective residuals due to e will also be adm-equivalent.

Theorem 14 Let $E \approx_A E'$. Then, for all $f \in \Gamma$, $E/f \approx_{A\uparrow f} E'/f$. ∎

7.2 Soundness and Completeness

Theorem 14 justifies *adm-soundness*, which (in contrast to Definition 8) uses adm-equivalence instead of equality. This notion also accommodates the change of state implicit in event occurrence.

Definition 12 $D/e \doteq F$ is adm-sound iff for all admissible sets A, $D/e \approx_{A\uparrow e} F$

We have thus established adm-soundness as a reasonable formal notion of correctness for our equations. Since $\approx_A$ is reflexive, we also have the following.

Lemma 15 If $D/e \doteq F$ is sound, then $D/e \doteq F$ is adm-sound. ∎

Lemma 16 Equations 15 and 16 are adm-sound. ∎

Theorem 17 Equations 9–16 are adm-sound. ∎

Theorem 17 thus establishes that all our equations are correct. It takes advantage of our intuitive assumptions of how the scheduler behaves and how events occur. Theorem 17 is important because it enables us to combine the benefits of most general solutions (Theorem 4) with the benefits of efficient computation (Lemma 7).

By Lemma 8, our equations yield just one answer for D/e. Although numerous expressions have the same denotation as $[\![D/e]\!]$, we can show that our equations will find an expression that is equivalent to the desired result with respect to the desired behavior of the scheduler.

Definition 13 A set of equations is *adm-complete* iff $D/e \models F$ implies that $D/e \vdash F'$ and for all admissible sets A, $F \approx_{A\restriction e} F'$

Theorem 18 Equations 9–16 are adm-complete. ∎

8 Arbitrary Tasks

Although we require that event *instances* are not repeated, an event *type* may be instantiated multiple times. Event symbols are interpreted as types; event instances are instantiated from types through parametrization. The parameters in dependencies can be variables, which are implicitly universally quantified. When events are scheduled, all parameters must be constants. The parameters must be chosen so that event instances are unique. Typical parameters include transaction and task IDs, database keys, timestamps, and so on. We can uniquely identify each event instance by combining its task ID with the value of a monotonic counter that records the total number of instances of events of that task. Event IDs are reminiscent of *operation IDs* used to uniquify log operations [23]. It is interesting that this old idea can be adapted to workflows.

We define $\mathcal{E}_P$ as the language generated by (a) substituting '$\mathcal{E}_P$' for '$\mathcal{E}$' in syntax rules **??** and **??**, and (b) adding rules 5, 6, and 7 below. We assume a set $\mathcal{V}$ of variables and a set $\mathcal{C}$ constants to use as parameters. Thus, Γ includes all (ground) event literals and Ξ includes all event atoms. The universe, $\mathbf{U}_{\mathcal{E}}$, depends on Γ, and includes all traces formed from all possible event instances. $\delta(e)$ gives the number of parameters needed to instantiate e.

Syntax 5 $\Xi \subseteq \mathcal{E}_P$

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

Syntax 7 $e \in \Sigma$, $\delta(e) = m$, $p_1, \ldots, p_m \in (\mathcal{V} \cup \mathcal{C})$
implies $e[p_1 \ldots p_m], \overline{e}[p_1 \ldots p_m] \in \Xi$

The semantics of $\mathcal{E}_P$ is given by replacing Semantic rule **??** by 16 and 17 below. Semantics 17 corresponds to universal quantification. $E(v)$ refers to an expression free in variable v (it may also be free in other variables). $E(v ::= c)$ refers to the expression obtained from $E(v)$ by substituting every occurrence of v by constant c.

Semantics 16 $[\![f[p_1 \ldots p_m]]\!] =$
$\{\tau \in \mathbf{U}_{\mathcal{E}} : \tau \text{ mentions } f[p_1 \ldots p_m]\}$, $f[p_1 \ldots p_m] \in \Gamma$

Semantics 17 $[\![E(v)]\!] = \bigcap_{c \in \mathcal{C}} [\![E(v ::= c)]\!]$

We assume that (a) events from the same task have the same variable parameters, and (b) all references to the same event type involve the same tuple of parameters. These assumptions are reasonable because our focus is on intertask dependencies. They enable us to interpret dependencies and schedule events properly.

Parameters can be used *within* a given workflow to relate events in different tasks. Typically, the same variables are used in parameters on events of different tasks. Attempting some key event binds the parameters of all events, thus instantiating the workflow, and scheduling it as before. We redo Example 15 below.

Example 23 Now we use t as the trip or reservation id to parametrize the workflow. The parameter t is bound when the *buy* task is begun. The explanations are as before—now we are explicit that the same customer features throughout the workflow. The desired workflow may be specified by making the trip id explicit in each dependency, e.g., (D'_1) $\overline{s_{buy}[t]} \vee s_{book}[t]$. Let t be bound to a natural number.

The above workflow is satisfied by infinitely many legal traces, e.g., (τ_6) $s_{buy}[65] s_{book}[65] c_{book}[65]$ $c_{buy}[65]$, and (τ_7) $\overline{s_{buy}[34]}\, \overline{s_{book}[34]}\, \overline{s_{cancel}[34]}$. The traces τ_6 and τ_7 are as before but with explicit parameters. Trace τ_8 shows how different instantiations of the workflow may interleave. ∎

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

Example 24 Let b_i denote the event of task T_i entering its critical section and e_i denote the event of T_i exiting its critical section. Then, mutual exclusion between tasks T_1 and T_2 may be formalized as follows by stating that if T_1 enters its critical section before T_2, then T_1 exits its critical section before T_2 enters.

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

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

By Semantics 17, the above dependency is interpreted as $(\forall x, y : D_M(x, y))$. Suppose that $b_1[\hat{x}]$ for a specific and unique $\hat{x}$ occurs. This instantiates and residuates the above expression to $(\overline{b_2[y]} \vee e_1[\hat{x}] \cdot b_2[y])$. Thus the overall dependency becomes $(\forall x, y : x \neq \hat{x} \Rightarrow D_M(x, y)) \wedge (\overline{b_2[y]} \vee e_1[\hat{x}] \cdot b_2[y])$. In other words, $b_2[y]$ is disabled for all y, because residuating the above expression with $b_2[y]$ yields 0. However, residuating the above expression with $e_1[\hat{x}]$

yields $(\forall x, y : x \neq \hat{x} \Rightarrow D_M(x, y)) \wedge \top$. Thus $e_1[\hat{x}]$ can occur. Furthermore, anything allowed by $D_M(x, y)$ (except another occurrence of $b_1[\hat{x}]$ or $e_1[\hat{x}]$) can occur after $e_1[\hat{x}]$.

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

We implicitly used the inference rule $(\forall z : D(z)) \equiv (\forall z : z \neq \hat{z} \Rightarrow D(z)) \wedge D(z ::= \hat{z})$, for any constant $\hat{z}$. By Observation 5, residuating with an event instance $e[\hat{z}]$ returns the first conjunct unchanged, conjoined with the result of residuating $D(z ::= \hat{z})$ by $e[\hat{z}]$. In this manner, dependencies "grow" to accommodate the appropriate instances explicitly. When the given instantiation is satisfied, it is no longer needed. By uniqueness of instances attempted, we can safely forget about past instances. Thus, in quiescense only the original dependency may be stored.

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

Equation 17 $E(\vec{v})/e[\vec{c}] \doteq E(\vec{v}) \wedge (E(\vec{v} ::= \vec{c})/e[\vec{c}])$

Lemma 19 Equation 17 is adm-sound. ∎

Importantly, Example 24 makes no assumptions about the conditions under which the two tasks attempt to enter or exit their critical sections. This turns out to be true in our approach in other cases as well. Importantly, the event IDs need *not* depend on the structure of the associated task, because our scheduler does not need to know the internal structure of a task agent. An agent may have arbitrary loops and branches and may exercise them in any order as required by the underlying task. Hence, we can handle arbitrary tasks correctly!

One might wonder about the value of parametrization to our formal theory. If we cared only about intra-workflow parametrization, parameters could be introduced extralogically, i.e., by modifying the way in which the theory is applied. However, when we care about inter-workflow parametrization, it is important to be able to handle parametrization from within the theory.

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

9 Further Technical Properties

Lemma 20 shows that we need not represent sequences of length greater than 2. This is an easy but important result, because it leads to great simplification in processing, as described in [53].

Lemma 20 $e_1 \cdot \ldots \cdot e_n \equiv e_1 \cdot e_2 \wedge \ldots \wedge e_{n-1} \cdot e_n$ ∎

We show why some alternatives to our approach would not be appropriate. There are subtle relationships between the operators $\wedge$ and $\cdot$, and the constants $\top$ and 1. Sometimes, e.g., [43], $\top$ is the unique maximal element of the algebra and 1 is the unit of the concatenation operator. That is, $[\![\top]\!] = \mathbf{U}$ (as here), whereas $[\![1]\!] = \{\Lambda\}$. However, we have no separate 1, effectively setting $1 \equiv \top$. Intuitively, $[\![E_1]\!] \subseteq [\![E_2]\!]$ means that E_1 is a stronger specification than E_2, because E_1 allows fewer traces. Consequently, any reasonable semantics should validate Lemma 21, which states that if the scheduler satisfies E followed by F, then it satisfies F (and E). Substituting 1 for F yields Observation 22.

Lemma 21 $[\![E \cdot F]\!] \subseteq [\![F]\!]$. ∎

Observation 22 $1 \equiv \top$. ∎

Some approaches require $1 \not\equiv \top$ to *avoid* the results that $(e \vee \overline{e}) \cdot f \equiv f$ and $f \equiv f \cdot (e \vee \overline{e})$. This would cause ordering information to be lost: f may be desired after e or $\overline{e}$, but not before them. But this result arises because those approaches require $e \vee \overline{e} \equiv \top$. Since in our approach, $e \vee \overline{e} \not\equiv \top$, setting $1 \equiv \top$ does not have the counterintuitive ramification that it might elsewhere.

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

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

We assume a formal complement for each significant event. Some events, e.g., *start* and *forget*, are not typically thought of as having complements. A superfluous formal complement causes no harm, because it is never instantiated. When a task agent has a multiway split (instead of two-way between *abort* and *commit*), then the complement of an event is, in effect, the join of all alternative events. Multiway splits are rare in practice.

The admissibility construction enables certain simplifications. Lemma 23 works because the union and intersection of sets of maximal (and legal) traces are also sets of maximal (and legal) traces. Surprisingly, Lemma 24 holds because $\cdot$ inherently assumes nonmaximal traces. Thus the fact that its arguments were equal under maximality carries little significance.

Lemma 23 $E \approx_A E'$ implies that each of the following holds: (a) $E \wedge F \approx_A E' \wedge F$; (b) $F \wedge E \approx_A F \wedge E'$; (c) $E \vee F \approx_A E' \vee F$; and (d) $F \vee E \approx_A F \vee E'$. ∎

Lemma 24 $E \approx_A E'$ does *not* imply that either of the following holds: (a) $E \cdot F \approx_A E' \cdot F$; and (b) $F \cdot E \approx_A F \cdot E'$. ∎

Therefore, we can use Lemma 23 to simplify expressions provided we avoid simplification in the context of a $\cdot$. Fortunately, since by Observation 9, our equations always yield TSF output from TSF input, we never have to worry about an expression in which the $\cdot$ operator is outside of a $\vee$ or $\wedge$. Thus, the simplifications prevented by Lemma 24 would never be needed anyway. So nothing is lost!

10 Conclusions

We developed a model-theoretic semantics for events and dependencies that satisfies both workflow intuitions and formal semantics criteria. This semantics provides a basis for checking the consistency and enforceability of dependencies. It abstractly generates eager schedules from lazy specifications by symbolically computing the preconditions and postconditions of an event. By using admissibility, our definition of residuation is specialized for use in scheduling, and yields strong and succinct answers for various scheduling decisions.

We obtain succinct representations for many interesting dependencies—no worse and often much better than previous approaches. For example, compensation dependencies are given a representation of size 3 here, but of over size 40 in [2]. We have not analyzed the complexity in detail.

More general logic programming techniques for reasoning about integrity constraints and transactions are no doubt important, but the connection has not been explored yet. It appears that we deal with lower-level scheduling issues, whereas the above approaches deal with application-level constraints. We speculate that they could supply the dependencies that are input to our approach. Our focus is on identifying the core scheduling and semantic issues, which will be relevant no matter how the final implementation is achieved.

It has been recently argued that various transaction models can be expressed in existing workflow products, and therefore existing workflow products are sufficient [1]. We reject this position. Compilability into machine code does not make higher-level programming languages irrelevant! Extended transaction and workflow models provide a programming discipline through which computations can be structured. If they could not be translated to lower-level representations, they would not be useful! We agree with [59] that higher-level abstractions are necessary for programming in complex environments.

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

11 Information Control Nets

In order to show how our approach can capture workflows expressed in other approaches, we consider a representative graphical notation called *information control nets (ICN)* [41]. ICN can express various control structures. ICN has a formal graphical syntax, but not a model-theoretic semantics.

Figure 9: ICN: An activity

ICN allows two main kinds of nodes—*activities:* those that are modeled as having a single start-finish execution branch, and *decision nodes:* those that are conditional. Figures 9 and

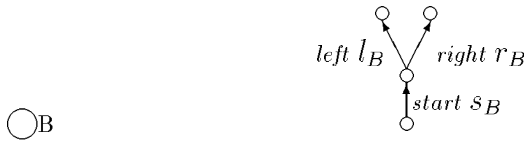
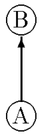
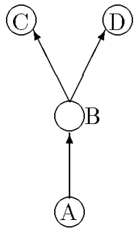


Figure 10: ICN: A decision node



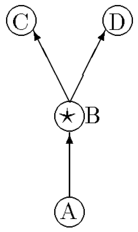
$$\overline{f}_A[x] \vee f_A[x] \cdot s_B[x]$$

Figure 11: ICN: Control flow



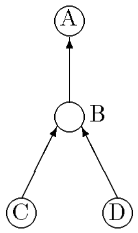
$$\overline{f}_A[x] \vee (f_A[x] \cdot l_B[x] \cdot s_C[x]) \vee$$
$$(f_A[x] \cdot r_B[x] \cdot s_D[x])$$

Figure 12: ICN: Disjunctive (out) branching



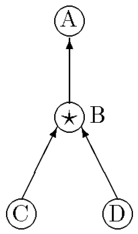
$$(\overline{f}_A[x] \vee f_A[x] \cdot s_C[x]) \wedge (\overline{f}_A[x] \vee f_A[x] \cdot s_D[x])$$

Figure 13: ICN: Conjunctive (out) branching



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

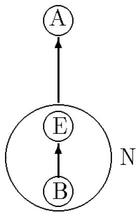
Figure 14: ICN: Disjunctive (in) branching



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

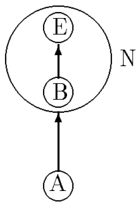
Figure 15: ICN: Conjunctive (in) branching

10 show how we model activities and decision nodes. Using these, we can readily capture the meanings of the constructs for control flow (Figure 11), outbound disjunctive branching (Figure 12), outbound conjunctive branching (Figure 13), inbound disjunctive branching (Figure 14), and inbound conjunctive branching (Figure 15).



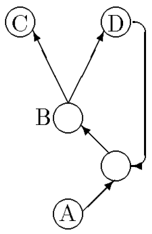
$$\overline{f}_{N.E}[x] \vee f_{N.E}[x] \cdot s_A[x]$$

Figure 16: ICN: Nesting Inbound



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

Figure 17: ICN: Nesting Outbound



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

Figure 18: ICN: Iteration

ICN allows nodes to be nested. Without loss of generality, we restrict ourselves to nested nodes with a single begin subnode and a single end subnode. For a nested node N, these are referred to as $N.B$ and $N.E$, respectively. Figures 17 and 16 show the translation. Lastly, we consider iteration in Figure 18. The flow of control from A to B initializes the iteration. The iteration ends when C is invoked. Each time D is invoked, it passes control back to B. In the formal expressions, x represents the parameters of interest to the application; i represents the loop counter, which helps uniquify events from different iterations.

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.

12 Guards on Events

A naive implementation of [52] 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.

12.1 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 8, 9, and 10 are exactly as for the dependency language $\mathcal{E}$ [52]. Rule 11 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 8 $\Gamma \subseteq \mathcal{T}$

Syntax 9 $0, \top \in \mathcal{T}$

Syntax 10 $E_1, E_2 \in \mathcal{T}$ implies $E_1 \vee E_2$, $E_1 \wedge E_2$, $E_1 \cdot E_2 \in \mathcal{T}$

Syntax 11 $E \in \mathcal{T}$ 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 [15]. 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 14 A trace, u, is *maximal* iff for each event, either the event or its complement occurs on u.

Definition 15 $\mathbf{U}_{\mathcal{T}} \triangleq$ the set of maximal traces.

We assume $\Sigma \neq \emptyset$; hence, $\Gamma \neq \emptyset$. Semantics 18, 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 20, 21, 23, and 24 are as in traditional formal semantics. Semantics 25 and 26 involve looking into the future. Semantics 19 and 22 capture the dependence of an expression on the immediate past, bounded by the first index of the semantic definition. Semantics 22 introduces a nonzero first index.

Semantics 18 $u \models_i E$ iff $u \models_{0,i} E$

Semantics 19 $u \models_{i,k} f$ iff $(\exists j : i \leq j \leq k$ and $u_j = f)$, where $f \in \Gamma$

Semantics 20 $u \models_{i,k} E_1 \vee E_2$ iff $u \models_{i,k} E_1$ or $u \models_{i,k} E_2$

Semantics 21 $u \models_{i,k} E_1 \wedge E_2$ iff $u \models_{i,k} E_1$ and $u \models_{i,k} E_2$

Semantics 22 $u \models_{i,k} E_1 \cdot E_2$ iff $(\exists j : i \leq j \leq k$ and $u \models_{i,j} E_1$ and $u \models_{j+1,k} E_2)$

Semantics 23 $u \models_{i,k} \top$

Semantics 24 $u \models_{i,k} \neg E$ iff $u \not\models_{i,k} E$

Semantics 25 $u \models_{i,k} \Box E$ iff $(\forall j : k \leq j \Rightarrow u \models_{i,j} E)$

Semantics 26 $u \models_{i,k} \Diamond E$ iff $(\exists j : k \leq j$ and $u \models_{i,j} E)$

Example 25 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 $\Diamond$. ▌

12.2 Important Properties

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

Definition 16 $E \cong F \triangleq (\forall i, k : u \models_{i,k} E$ iff $u \models_{i,k} F)$, where $E, F \in \mathcal{T}$.

Definition 17 [52] $E \equiv F$ 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 25 and 26 indicate a fundamental difference between $\mathcal{E}$ and $\mathcal{T}$.

Observation 25 $(\forall e : \top \ncong e \vee \overline{e})$ and $(\forall e : 0 \cong e \wedge \overline{e})$. Also, $(\forall e : \top \cong \Diamond e \vee \Diamond \overline{e})$. ∎

Observation 26 $(\forall e : \top \not\equiv e \vee \overline{e})$ and $(\forall e : 0 \not\equiv e \wedge \overline{e})$. ∎

Definition 18 An event e is *stable* iff if e is satisfied at a given index, then it is satisfied at all future indices.

Observation 27 means that events are *stable*. However, Observation 28 shows that the temporal operators cannot always be eliminated. Intuitively, $\neg e$ means "not yet e." Observation 29 states that if the last event of a sequence has occurred, then the entire sequence has occurred. Similarly, Observation 30 states that if the first event of a sequence has not occurred, then the next event has not occurred either.

Observation 27 $\Box e \cong e$. ∎

Observation 28 $\Box \neg e \ncong \neg e$. ∎

Observation 29 $\Diamond(e_1 \cdot e_2) \wedge \Box e_2 \cong \Box(e_1 \cdot e_2)$. ∎

Observation 30 $\Diamond(e_1 \cdot e_2) \wedge \neg e_1 \cong \Diamond(e_1 \cdot e_2) \wedge \neg e_1 \wedge \neg e_2$. ∎

Example 26 The possible maximal traces for $\Gamma = \{e, \overline{e}\}$ 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

- $\Box e \vee \Box \overline{e} \ncong \top$: 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 $\neg e$. ∎

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 $\mathcal{T}$.

12.3 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 13.2 discusses additional features. Section 14.2 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 $\Gamma_\rho \supseteq \Gamma_D$, i.e., all events in D (or their complements) feature in ρ. Each path is effectively a correct execution for its dependency. A path may have more events than those explicitly mentioned in a dependency. This is not a problem: section 14.2 develops an equivalent approach that only looks at the dependency itself, not the paths.

Definition 19 $D/\rho \triangleq ((D/e_1)/\ldots)/e_n$.

Definition 20 $\rho = \langle e_1 \ldots e_n \rangle$ is a path iff the events e_i are distinct, and not complements of each other.

Definition 21 $\Pi(D)$ is the set of paths satisfying D. $\Pi(D) \triangleq \{\rho : \rho \text{ is a path and } \Gamma_\rho \supseteq \Gamma_D \text{ and } D/\rho = \top\}$.

Lemma 31 means that the paths satisfy the given dependency. Lemma 32 establishes that there is a unique dependency corresponding to any set of paths.

Lemma 31 $\rho \in \Pi(D)$ iff ρ is a path and $\Gamma_\rho \supseteq \Gamma_D$ and $\rho \models D$. ∎

Lemma 32 $D \equiv \bigvee_{\rho \in \Pi(D)} \rho$. ∎

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 22 $pre(\rho, e) \triangleq$ if $e = e_i$, then $\Box(e_1 \cdot \ldots \cdot e_{i-1})$, else 0.

Definition 23 $post(\rho, e) \triangleq$ 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 $\mathsf{G} : \mathcal{E} \times \Gamma \mapsto \mathcal{T}$. $\mathsf{G}_b(\rho, e)$ denotes the guard on e due to path ρ (b stands for *basic*). $\mathsf{G}_b(D, e)$ denotes the guard on e due to dependency D. To compute the guard on an event relative to a dependency D, we sum the contributions of different paths in D. $\mathsf{G}_b(\mathcal{W}, e)$ denotes the guard due to workflow $\mathcal{W}$ and is abbreviated as $\mathsf{G}_b(e)$ when $\mathcal{W}$ is 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 24 $\mathsf{G}_b(\rho, e) \triangleq pre(\rho, e) \wedge post(\rho, e)$.
$\mathsf{G}_b(D, e) \triangleq \bigvee_{\rho \in \Pi(D)} \mathsf{G}_b(\rho, e)$.
$\mathsf{G}_b(\mathcal{W}, e) \triangleq \bigwedge_{D \in \mathcal{W}} \mathsf{G}_b(D, e)$.

Observation 33 $u \models_k \mathsf{G}_b(D, e) \Rightarrow (\exists \rho \in \Pi(D) : u \models_k \mathsf{G}_b(\rho, e))$. ∎

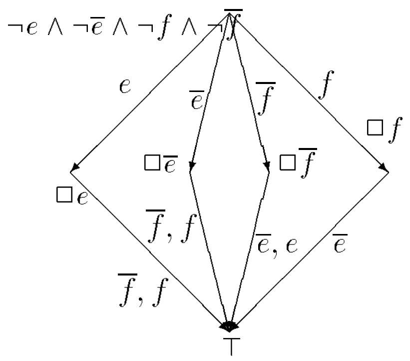


Figure 19: Guards with respect to $D_< = \overline{e} \vee \overline{f} \vee e \cdot f$

Figure 19 illustrates our procedure for the dependency of [52, Example 13]. 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 [52, section 6.1], and relate better to the results of section 14. 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 27 Using Figure 19, 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. ∎

13 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 28 illustrates this. More complex cases follow.

Example 28 Using the guards from Example 27, 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. ▮

13.1 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 14.

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 29 Following Example 27, 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 50 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 30 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 30 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.

13.2 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 [52, Figure **??**] 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 12.3 applies to normal events; we consider inevitable and immediate events below.

13.2.1 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 20 shows the dependency of Figure 19. The path $\langle f\overline{e}\rangle$ is deleted because if f occurs first, e must not occur. We can verify that $\mathsf{G}_b(D_<, e)$ is unchanged but $\mathsf{G}_b(D_<, f)$ is stronger: since e cannot be rejected, we cannot let f happen unless e or $\overline{e}$ has already happened.

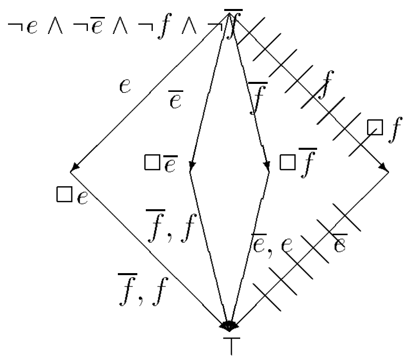


Figure 20: Guards from $D_<$ assuming e is inevitable

13.2.2 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 31 Figure 20 still holds when e is immediate. Thus the same guards are obtained as before. ▮

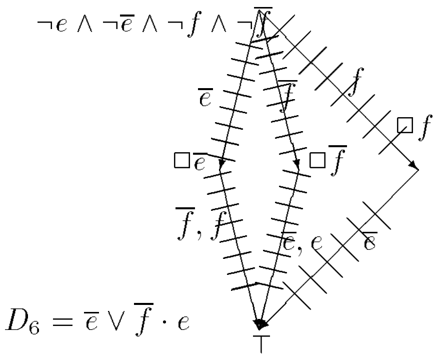


Figure 21: Extreme example of an immediate event (e)

Example 32 Referring to Figure 21, 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 22 below). ▮

13.2.3 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 33 and 34 show how the enforceability of dependencies can vary based on the event attributes that are asserted. Example 35 shows dependencies that are individually, but not jointly, enforceable.

Example 33 $D_<$ (in Figure 20) 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. ▮

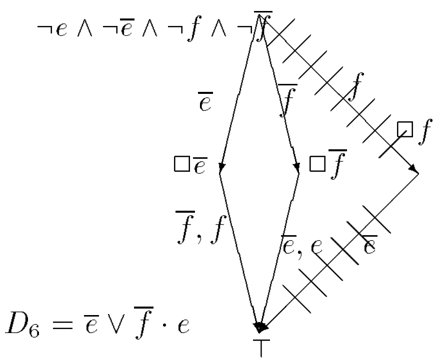


Figure 22: Unenforceable when e and f are inevitable

Example 34 $D_6 = \overline{e} \vee \overline{f} \cdot e$ (diagramed in Figure 22) is unenforceable when e and f are both inevitable, because there is no path on which they both occur. ▮

Example 35 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. ▮

13.2.4 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 36 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 30, but the later event is to be triggered.

Example 37 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. ▮

14 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 25 An evaluation strategy is a function $\mathsf{S} : \wp(\mathcal{E}) \mapsto (\Gamma \times \mathbf{U}_{\mathcal{T}} \times \mathbf{N} \mapsto \mathcal{T})$.

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 $\mathsf{S}(\mathcal{W}) \leadsto u$.

Definition 26 $\mathsf{S}(\mathcal{W}) \leadsto_i u$ iff $(\forall j : 1 \leq j \leq i \Rightarrow u \models_{j-1} \mathsf{S}(\mathcal{W}, u_j))$.

Definition 27 $\mathsf{S}(\mathcal{W}) \leadsto u \triangleq (\forall i : i \leq |u| \Rightarrow \mathsf{S}(\mathcal{W}) \leadsto_i u)$.

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 24, we begin with a trivial strategy, S_b. S_b sets the guards using G_b and *never* modifies them. Theorem 35 establishes the soundness and completeness of guard compilation.

Definition 28 $(\forall v, j : \mathsf{S}_b(\mathcal{W}, e, v, j) = \mathsf{G}_b(\mathcal{W}, e))$.

Observation 34 $\mathsf{S}_b(\mathcal{W}) \leadsto u$ iff $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(\mathcal{W}, u_j))$. ∎

Theorem 35 $\mathsf{S}_b(\mathcal{W}) \leadsto u$ iff $(\forall D \in \mathcal{W} : u \models D)$. ∎

14.1 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 6 defines these rules. Because of two-sequence form (TSF) [52], we need not consider longer sequences.

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 6: Assimilating Messages

When the dependencies involve sequence expressions, the guards can end up with sequence expressions, which indicate ordering of the relevant events. In such cases, the information that is assimilated into a guard must be consistent with that order. For this reason, the updates in those cases are more complex. Lemma 36 means that the operator $\div$ preserves the truth of the original guards.

Lemma 36 $(\exists k \leq j : u \models_k M$ and $u \models_j G \div M) \Rightarrow u \models_j G$. ∎

The repeated application of $\div$ to update the guards 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 29 $\mathsf{S}_\div(\mathcal{W}, e, u, 0) \triangleq \mathsf{G}_b(\mathcal{W}, e)$.
$\mathsf{S}_\div(\mathcal{W}, e, u, i{+}1) \neq \mathsf{S}_\div(\mathcal{W}, e, u, i) \Rightarrow (\exists k \leq i : u \models_k M$ 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 36 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 37 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 37 Replacing S_b by $\mathsf{S}_{\div}$ preserves correctness, i.e., $\mathsf{S}_{\div}(\mathcal{W}) \rightsquigarrow u$ iff $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$. ∎

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 37 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.

14.2 Simplification

We now show how guards as given in Definition 24 can be computed more efficiently. Theorems 38 and 39 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 39 is also important for another reason, namely, because it essentially equates a workflow with the conjunction of the dependencies in it.

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

Theorem 39 $\mathsf{G}_b(D \wedge E, e) = \mathsf{G}_b(D, e) \wedge \mathsf{G}_b(E, e)$. ∎

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 41 means that the guard compilation essentially uses $\Psi(D)$—the other paths in $\Pi(D)$ can be safely ignored. Theorem 42 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 30 $\Psi(D) \triangleq \{w : w \in \Pi(D), \Gamma_w = \Gamma_D\}$.

Lemma 40 $D \equiv \bigvee_{\rho \in \Psi(D)} \rho$. ∎

Lemma 41 If $e \in \Gamma_D$, then $\mathsf{G}_b(D, e) = \bigvee_{w \in \Psi(D)} \mathsf{G}_b(w, e)$. ∎

Theorem 42 $\mathsf{G}_b(D, e) = \Diamond D$, if $e \notin \Gamma_D$. ∎

14.2.1 Relaxing the Past

Definition 31 allows us to replace certain sequences in the past component of Definition 24 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 31 $\mathsf{G}_p^\Delta(\mathcal{W}, e) \triangleq$ 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 29. S_p^Δ is the evaluation strategy corresponding to G_p^Δ, in which updates happen as before. Theorem 43 establishes the correctness of S_p^Δ.

Theorem 43 For all Δ, replacing $\mathsf{S}_\div(\mathcal{W})$ by $\mathsf{S}_p^\Delta(\mathcal{W})$ preserves correctness. ∎

14.2.2 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. $\mathsf{S}_\wedge^\Delta$ is the evaluation strategy corresponding to $\mathsf{G}_\wedge^\Delta$.

Definition 32 $\mathsf{G}_\wedge^\Delta(\mathcal{W}, e) \triangleq$ in $\mathsf{G}_p^\Delta(\mathcal{W}, e)$ 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 44 holds, because Definition 32 allows the traces validly generated from the given dependency to be combined into spurious traces.

Lemma 44 Replacing $\mathsf{S}_\div(\mathcal{W})$ by $\mathsf{S}_\wedge^\Delta(\mathcal{W})$ 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 [52, Example 15], and Examples **??** and **??** of section **??**. Lemma 45 shows the present formulation applies to 3-workflows.

Syntax 12 $0, \top \in \mathcal{E}_3$

Syntax 13 $e_1 \cdot \ldots \cdot e_n \in \mathcal{E}_3$ if $e_i \in \Gamma$ and $n \leq 3$

Syntax 14 $E_1, E_2 \in \mathcal{E}_3$ implies that $E_1 \vee E_2 \in \mathcal{E}_3$

Lemma 45 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 32. 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.

14.2.3 Eliminating Irrelevant Guards

Theorem 46 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 33 $\mathsf{G}^{\Delta}_{\top}(D, e) = \top$, if $e \not\in \Gamma_D$ and $D \in \mathcal{W}$, where $e \not\in \Delta$; $\mathsf{G}^{\Delta}_{\top}(D, e) = \mathsf{G}^{\Delta}_{p}(D, e)$ otherwise.

Theorem 46 Replacing $\mathsf{S}^{\Delta}_{p}(\mathcal{W})$ by $\mathsf{S}^{\Delta}_{\top}(\mathcal{W})$ does not violate correctness. ∎

Theorems 39 and 46 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.

14.3 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 $|\Gamma_D|/2$.

Definition 34 $\sigma^0(D) \triangleq \Psi(D)$.
$\sigma^{i+1}(D) \triangleq \sigma^i(D) \cap \{w : \beta(w, \sigma^i(D))\}$, where $\beta(w, P)$ iff

- if $e \in \Gamma_D \cap \Sigma_v$, 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 35 $\Phi(D) \triangleq \sigma^{\lceil |\Gamma_D|/2 \rceil}(D)$.

$\Phi(D)$ 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 47 and 48 show that the formal definition behaves as desired. Lemmas 47 and 48 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 36 $\alpha(D) \triangleq \bigvee_{w \in \Phi(D)} w$.
$\alpha(\mathcal{W}) = \{\alpha(D) : D \in \mathcal{W}\}$.

Lemma 47 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 48 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 38 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 21, 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 37 $\mathsf{S}_m(\mathcal{W}, e, u, i) \triangleq$ (if $e \in \Sigma_m$, then $\top$, else $\mathsf{S}_{\neg}^{\Sigma_m}(\mathcal{W}, e, u, i)$).

Theorem 49 Replacing $\mathsf{S}_{\neg}^{\triangle}$ 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 50 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 $f \notin \Delta$. The guards update according to $\mathsf{S}_{\top}^{\Delta}$—the reduction kicks in independently at each moment.

Definition 38 $\mathsf{S}_{\neg}^{\Delta}(\mathcal{W}, e, u, i) =$
$\mathsf{S}_{\top}^{\Delta}(\mathcal{W}, e, u, i)|_{(\forall f \in \Gamma \backslash \Delta : \Box f ::= 0, \neg f ::= \top)}$.

Theorem 50 For all Δ, replacing $\mathsf{S}_{\top}^{\Delta}$ by $\mathsf{S}_{\neg}^{\Delta}$ does not violate correctness. ∎

15 Overview of the Literature

15.1 Models

Database Approaches Several extended transaction models have been proposed [14]. *Sagas* are composed of several subtransactions or *steps* [18], which can be optimistically committed, thereby increasing concurrency but reducing isolation. If a step fails, consistency can be restored by compensating the previously committed steps in the reverse order. The *NT/PV* (nested transactions with predicates and views) model is based on multiple coexisting versions of a database [35]. Transactions are modeled as binary relations on database states, rather than as functions. Specifications for transactions are given as precondition and postcondition pairs. Several transaction models can be expressed in the NT/PV model.

ConTracts group a set of transactions into a multitransaction activity [58]. 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 [12]. Control and data flow may be specified in a script or with event-condition-action rules. Other important models include *flex transactions* [6] and *cooperating activities* [45].

Organizational Approaches There is also increasing interest in the organizational aspects of workflow management. [30] 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. [7] also relate workflows with organizational modeling.

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

15.2 Representing Models

ACTA ACTA provides a formal framework to specify the effects of transactions on other transactions and objects, but does not address scheduling [9]. In ACTA, an execution of a transaction is a partial order—denoting temporal precedence—of the events of that transaction (the object events it invokes, plus its significant events). A history of a concurrent execution of a set of transactions contains all events of each of the transactions, along with a partial order that is consistent with the partial orders for the individual transactions. $e \in H$ means that e occurs in history H. $e \longrightarrow e'$ (in history H) means that e and e' occur in H, and e precedes e'.

ACTA provides a formal syntax, but not a model-theoretic semantics. An important semantic issue from our standpoint is the distinction between event types and instances. ACTA's formal definitions appear to involve event instances, because they expect a partial order of events. However, certain usages are less clear. For example, consider the statement: "(when t_i reads a page x that t_j subsequently writes), if t_j commits before t_i, t_i must reread x after t_j commits." ACTA captures this by the formula: $(read_{t_i}[x] \longrightarrow write_{t_j}[x]) \Rightarrow ((Commit_{t_j} \longrightarrow Commit_{t_i}) \Rightarrow (Commit_{t_j} \longrightarrow read_{t_i}[x]))$ [9, p. 363]. This formula uses $read_{t_i}[x]$ to refer to two different event instances, one before $Commit_{t_j}$, and the other after $Commit_{t_j}$. Thus this formula can hold only vacuously. This is not a major error, but it shows that ACTA is not designed for representing repeating events.

Rule-Driven Transaction Management Klein proposes two primitives for defining dependencies [34]. In Klein's notation, $e \rightarrow f$ means that if e occurs then f also occurs (before or after e). His $e < f$ means that if both events e and f happen, then e precedes f. This work describes a formalism and its intended usage, but gives no formal semantics. The semantics is informally explained using *complete histories*, which are those in which every task has terminated. Further, it is assumed that tasks are expressible as loop-free regular expressions. Thus this approach is not applicable to activities that never terminate, or those that iterate over their significant events.

Temporal Logic Approaches Our previous approach [2] is based on a branching-time temporal logic, CTL (or computation tree logic [15]). This approach formalizes dependencies in CTL and gives a formal semantics. It synthesizes finite state automata for the different dependencies. To schedule events, it searches for an executable, consistent set of paths, one in each of the given automata. This avoids computing product automata, but the individual automata in this approach can be quite large. Further, the CTL representations of the common dependencies are quite intricate. This implementation was centralized. Another temporal logic approach is that of Günthör [24]. Günthör's approach is based on linear temporal logic, and gives a formal semantics. His implementation too is centralized and his approach appears incomplete.

Action Logic This approach is a general-purpose theory of events, but is not designed for specifying and scheduling workflows [43]. Pratt proposes an algebra, and motivates some potentially useful inferences that constrain the possible models for the algebra. Our definitions of complements and admissibility are major extensions beyond [43]. Regular languages, which correspond to linear histories of events, are a class of models. The branching (partially ordered) histories well-known from serializability theory [3] (also formalized in ACTA) can be expressed as sets of linear histories. We find this connection fruitful, although we are not interested in conflicts among operations, or necessarily terminating computations.

Remarks on the Above Approaches The database approaches have useful features, but are either informal and possibly ambiguous, or not accompanied by distributed scheduling algorithms. ACTA and Klein's approaches are noncompositional, since the denotation they give to a formula is not derived from the denotation of its operands. This makes it difficult to reason symbolically. The restriction to loop-free tasks and the lack of an explicit distinction between event types and instances are the limiting properties common to all four approaches. However, these approaches agree on the *stability* of events—an event once occurred is true forever. This is a natural intuition, and one that we preserve for event instances. Pratt's approach is formal and compositional, but lacks a scheduling algorithm—his inferences are too weak to apply in scheduling. It too does not distinguish between event types and instances. Our approach goes beyond ACTA in characterizing the reasoning required for scheduling. Roughly, it is to ACTA what unification is to predicate logic.

Several execution environments have been proposed, which support specification and execution of transactions. An actor-based environment is developed in [26]. The DOM project includes a programmable environment (TSME) in which several transaction models can be specified [20]. However, TSME defines correctness criteria based on transaction histories, like in traditional approaches. The task specification languages for *interactions* [40] and METEOR [36] are similar in intent. ASSET is a programming facility for specifying transactions in the Ode environment [4]. This facility borrows intuitions, but not the formalism, from ACTA (discussed below). [32] 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 [10]. It introduced significant events of database transactions. ACTA provides a history-based formalism for specifying intertask dependencies. It is similar in spirit to [52], 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 [34]. 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 [24]. 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 [46].

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 [56]. 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.

16 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. [21] 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] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Ad-

vanced transaction models in workflow contexts. In *International Conference on Data Engineering*, February 1996.

[2] Paul C. Attie, Munindar P. Singh, Amit P. Sheth, and Marek Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th VLDB Conference*, pages 134–145, August 1993.

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

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

[5] Alan Bond and Les Gasser, editors. *Readings in Artificial Intelligence*. Morgan Kaufmann, San Francisco, 1988.

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

[7] Christoph Bußler and Stefan Jablonski. 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, 1994.

[8] Cristiano Castelfranchi. Commitments: From individual intentions to groups and organizations. In *Proceedings of the International Conference on Multiagent Systems*, pages 41–48, 1995.

[9] Panos Chrysanthis and Krithi Ramamritham. ACTA: The SAGA continues. In *[14]*, chapter 10. 1992.

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

[11] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983. Reprinted in [5].

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

[13] Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the International Conference on Multiagent Systems*, pages 73–80, 1995.

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

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

[16] E. Allen Emerson and Edmund C. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[17] Michael Fisher and Michael Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Intelligent and Cooperative Information Systems*, 6(1):37–65, 1997.

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

[19] Les Gasser. Social conceptions of knowledge and action: DAI foundations and open systems semantics. *Artificial Intelligence*, 47:107–138, 1991.

[20] Dimitrios Georgakopoulos, Mark F. Hornick, and Frank Manola. Customizing transactions models and mechanisms in a programmable environment supporting reliable workflow automation. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):630–649, August 1996.

[21] Esin Gokkoca, Mehmet Altinel, Ibrahim Cingil, E. Nesime Tatbul, Pinar Koksal, and Asuman Dogac. Design and implementation of a distributed workflow enactment service. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, pages 89–98, 1997.

[22] Alvin I. Goldman. *A Theory of Human Action.* Prentice-Hall, Englewood Cliffs, NJ, 1970.

[23] J. N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, September 1981.

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

[25] Afsaneh Haddadi. Towards a pragmatic theory of interactions. In *Proceedings of the International Conference on Multiagent Systems*, pages 133–139, 1995.

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

[27] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[28] Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1997.

[29] Michael N. Huhns, Munindar P. Singh, and Tomasz Ksiezyk. Global information management via local autonomous agents. In *Proceedings of the ICOT International Symposium on Fifth Generation Computer Systems: Workshop on Heterogeneous Cooperative Knowledge Bases*, pages 1–15, 1994. Reprinted in [28].

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

[31] Froduald Kabanza. Synchronizing multiagent plans using temporal logic specifications. In *Proceedings of the International Conference on Multiagent Systems*, pages 217–224, 1995.

[32] Kamalkar Karlapalem, Helen P. Yeung, and Patrick C. K. Hung. CapBasED-AMS: A framework for capability based and event driven activity management system. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, 1995.

[33] David Kinny and Michael P. Georgeff. Modelling and design of multi-agent systems. In *Intelligent Agents III: Agent Theories, Architectures, and Languages*, pages 1–20, 1997.

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

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

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

[37] Kazuhiro Kuwabara. Meta-level control of coordination protocols. In *Proceedings of the International Conference on Multiagent Systems*, pages 165–172, 1996.

[38] Frank Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *The German Database Conference - Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 51–70, 1995.

[39] B. C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, UK, 1986.

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

[41] Gary J. Nutt. Using workflow in contemporary IS applications. TR CU-CS-663-93, University of Colorado, August 1993.

[42] Tim Oates, M. V. Nagendra Prasad, and Victor R. Lesser. Cooperative information gathering: A distributed problem solving approach. *Proceedings of the IEE: Software Engineering*, 144(1), 1997.

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

[44] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, 1991. Reprinted in [28].

[45] Marek Rusinkiewicz, Wolfgang Klas, Thomas Tesch, Jürgen Wäsch, and Peter Muth. Towards a cooperative transaction model — the cooperative activity model. In *Proceedings of the 21st International Conference on Very Large Data Bases*, August 1995.

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

[47] Jon Siegel. *CORBA: Fundamentals and Programming.* Object Management Group and Wiley, New York, 1996.

[48] Munindar P. Singh. *Multiagent Systems: A Theoretical Framework for Intentions, Know-How, and Communications.* Springer-Verlag, Heidelberg, 1994.

[49] Munindar P. Singh. Semantical considerations on workflows: Algebraically specifying and scheduling intertask dependencies. In *Proceedings of the 5th International Workshop on Database Programming Languages (DBPL)*, pages 61–75, September 1995.

[50] Munindar P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 616–623, February 1996.

[51] Munindar P. Singh. Commitments among autonomous agents in information-rich environments. In *Proceedings of the 8th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, pages 141–155, May 1997.

[52] Munindar P. Singh. Formal aspects of workflow management, part 1: Semantics. TR 97-04, Department of Computer Science, North Carolina State University, Raleigh, February 1997. *Extends [49].* Available at www.csc.ncsu.edu/ faculty/ mpsingh/ papers/ databases/ wf_semantics.ps.

[53] Munindar P. Singh. Formal aspects of workflow management, part 2: Distributed scheduling. TR 97-05, Department of Computer Science, North Carolina State University, Raleigh, February 1997. *Extends [50].* Available at www.csc.ncsu.edu/ faculty/ mpsingh/ papers/ databases/ wf_scheduling.ps.

[54] Katia Sycara and Dajun Zeng. Multi-agent integration of information gathering and decision support. In *Proceedings of the European Conference on Artificial Intelligence*, pages 549–553, 1996.

[55] Milind Tambe. Agent architectures for flexible, practical teamwork. In *Proceedings of the National Conference on Artificial Intelligence*, 1997.

[56] W. M. P. van der Aalst. 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*, May 1996. `http:// optimus.cs.uga.edu:5080/ activities/NSF-workflow/ wfm.html`.

[57] Frank von Martial. *Coordinating Plans of Autonomous Agents*. Springer-Verlag, Berlin, 1992.

[58] Helmut Wächter and Andreas Reuter. The ConTract model. In *[14]*, chapter 7, pages 219–263. 1992.

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

[60] Michael Wooldridge. A knowledge-theoretic semantics for concurrent METATEM. In *Intelligent Agents III: Agent Theories, Architectures, and Languages*, pages 357–374, 1997.

A Proofs of Important Results

Auxiliary Definitions and Results

Observation 51 $\tau \in [\![E]\!]$ iff $(\forall \upsilon, \nu : \upsilon\tau\nu \in \mathbf{U}_{\mathcal{E}} \Rightarrow \upsilon\tau\nu \in [\![E]\!])$ ∎

Observation 51 means that if a trace satisfies E, then all larger traces do so too. Conversely, if all traces that include τ satisfy E, then τ satisfies E too (essentially by setting υ and ν to Λ).

Observation 52 If D is a sequence expression and $\tau \in [\![D]\!]$ then $(\forall i : 1 \leq i \leq length(D) \Rightarrow \tau \in [\![e_i]\!])$. ∎

Observation 53 If D is a sequence expression and $\tau \in [\![D]\!]$ then $length(\tau) \geq length(D)$. ∎

Proof of Lemma 8.

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

Proof. Consider any equation that applies on D. If this is equation 12 or equation 11, it results in recursive calls on / but on expressions whose size is strictly smaller than D. If it is any other equation, it produces an answer without making any recursive calls. Hence, the equations terminate. Since exactly one equation applies at each stage, the final answer is unique. Thus our equations are convergent.

Proof of Lemma 10.

Equations 9–14 are sound.

Proof. Equations 9 and 10 follow trivially from Semantics 15. Consider Equation 11. $\nu \in [\![(E_1 \wedge E_2)/e]\!]$ iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_\mathcal{E} \Rightarrow \upsilon\nu \in [\![E_1 \wedge E_2]\!]))$. This holds iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_\mathcal{E} \Rightarrow \upsilon\nu \in [\![E_1]\!] \cap [\![E_2]\!]))$, which is equivalent to $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_\mathcal{E} \Rightarrow \upsilon\nu \in [\![E_1]\!])) \wedge (\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow (\upsilon\nu \in \mathbf{U}_\mathcal{E} \Rightarrow \upsilon\nu \in [\![E_2]\!]))$. But this is equivalent to $\nu \in [\![E_1/e]\!] \cap [\![E_2/e]\!]$.

Equation 12 is the hardest. We can show that $[\![(E_1 \vee E_2)/e]\!] \supseteq [\![E_1/e]\!] \cup [\![E_2/e]\!]$. For the opposite direction, if either E_1 or E_2 is 0 or $\top$, then Equation 12 is trivially satisfied. Let $\nu \in [\![(E_1 \vee E_2)/e]\!]$. Then, since $\langle e \rangle \in [\![e]\!]$, the trace $\langle e \rangle\nu \in [\![E_1 \vee E_2]\!]$. Assume, without loss of generality, that $\langle e \rangle\nu \in [\![E_1]\!]$.

1. Let E_1 be a sequence expression. There are two cases. (a) $E_1 = e \cdot D$. Now, $\langle e \rangle\nu \in [\![e \cdot D]\!]$ implies that (using Observation 51) $\nu \in [\![D]\!]$. Then, by Semantics ??, $(\forall \upsilon \in [\![e]\!] : \upsilon\nu \in [\![e \cdot D]\!])$. Hence, $\nu \in [\![E_1/e]\!]$. (b) $E_1 = f \cdot D$ and $e \neq f$. By Observation 52, $\langle e \rangle\nu \in [\![f \cdot D]\!]$ implies that $\nu \in [\![f \cdot D]\!]$. By Observation 51, $(\forall \upsilon \in [\![e]\!] : \upsilon\nu \in [\![f \cdot D]\!])$. Hence, $\nu \in [\![E_1/e]\!]$.
2. Let E_1 be a disjunction. The proof follows by structural induction.
3. Let E_1 be a conjunction. The proof follows by an application of Equation 11.

Consider Equation 13. Let $\nu \in [\![E]\!]$. Then, by Semantics ??, $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow \upsilon\nu \in [\![e \cdot E]\!])$. Therefore, $\nu \in [\![(e \cdot E)/e]\!]$. Hence, $[\![E]\!] \subseteq [\![(e \cdot E)/e]\!]$. Conversely, let $\nu \in [\![(e \cdot E)/e]\!]$. Then $\langle e \rangle\nu \in [\![e \cdot E]\!]$. By Observation 53, any trace satisfying e must be at least of length 1. Therefore, by Semantics ??, there must a suffix τ of ν that satisfies E. Thus, by Observation 51, $\nu \in [\![E]\!]$. Hence, $[\![(e \cdot E)/e]\!] \subseteq [\![E]\!]$. Thus, $[\![(e \cdot E)/e]\!] = [\![E]\!]$.

Lastly, consider Equation 14. Let $\nu \in [\![D/e]\!]$. Then $\langle e \rangle\nu \in [\![D]\!]$. Since $e, \overline{e} \notin \Gamma_D$, $\nu \in [\![D]\!]$. Thus, $[\![D/e]\!] \subseteq [\![D]\!]$. Let $\nu \in [\![D]\!]$. Then, by Observation 51, $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow \upsilon\nu \in [\![D]\!])$. Thus, $\nu \in [\![D/e]\!]$ or $[\![D/e]\!] \supseteq [\![D]\!]$. Hence, $[\![D/e]\!] = [\![D]\!]$.

Proof of Lemma 11.

Equations 15 and 16 are *not* sound.

Proof. The proof is by simple counterexamples. For Equation 15, let $e' = f$ and $E = e$. We can verify that $\langle fe \rangle \in [\![(f \cdot e)/e]\!]$. Thus $[\![(f \cdot e)/e]\!] \neq \emptyset$. Similarly, for equation 16, let $E = f$. We can verify that $\langle \overline{e} f \rangle \in [\![(\overline{e} \cdot f)/e]\!]$. Thus $[\![(\overline{e} \cdot f)/e]\!] \neq \emptyset$.

Proof of Theorem 14.

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

Proof. Let $\nu \in (A \uparrow f \cap [\![E/f]\!])$. Then, $\langle f \rangle \nu \in A$. Also, $(\forall \upsilon \in [\![f]\!] : \upsilon\nu \in [\![E]\!])$. Therefore, since $\langle f \rangle \in [\![f]\!]$, we have that $\langle f \rangle \nu \in [\![E]\!]$. Thus $\langle f \rangle \nu \in (A \cap [\![E]\!])$. Since $E \approx_A E'$, $\langle f \rangle \nu \in (A \cap [\![E']\!])$. By Observation 51, $\langle f \rangle \nu \in [\![E']\!]$ implies that $(\forall \upsilon \in [\![f]\!] : \upsilon\nu \in [\![E']\!])$. Thus $\nu \in [\![E'/f]\!]$. Since $\nu \in A \uparrow f$, we obtain $\nu \in (A \uparrow f \cap [\![E'/f]\!])$. Consequently, we have established that $(A \uparrow f \cap [\![E/f]\!]) \subseteq (A \uparrow f \cap [\![E'/f]\!])$. By symmetry, $(A \uparrow f \cap [\![E'/f]\!]) \subseteq (A \uparrow f \cap [\![E/f]\!])$. Thus, $(A \uparrow f \cap [\![E/f]\!]) = (A \uparrow f \cap [\![E'/f]\!])$. Or, $E/f \approx_{A\uparrow f} E'/f$.

Proof of Lemma 16.

Equations 15 and 16 are adm-sound.

Proof. Consider Equation 15. By Semantics 15, $\nu \in [\![(e' \cdot E)/e]\!]$ iff $(\forall \upsilon : \upsilon \in [\![e]\!] \Rightarrow \upsilon\nu \in [\![e' \cdot E]\!])$. Since $\langle e \rangle \in [\![e]\!]$, this implies that $\langle e \rangle \nu \in [\![e' \cdot E]\!]$. By Observation 53, any trace that satisfies e' must be at least of length 1. Thus, by Semantics **??**, a suffix τ of ν exists such that $\tau \in [\![E]\!]$. By Observation 51, $\nu \in [\![E]\!]$. Since we convert expressions to TSF, E is a sequence expression. It is given that $e \in \Gamma_E$. Therefore, $\nu \in [\![e]\!]$. Thus by Observation 12, $\nu \notin A \uparrow e$. Consequently, $A \uparrow e \cap [\![(e' \cdot E)/e]\!] = \emptyset$, which equals $A \uparrow e \cap [\![0]\!]$. Hence, Equation 15 is adm-sound and similarly Equation 16.

Proof of Theorem 18.

Equations 9–16 are adm-complete.

Proof. By Lemma 8, D/e always evaluates to a unique expression. Let this be F'. Thus, $D/e \vdash F'$ always holds for some F'. Let $D/e \models F$. By Theorem 17, $F' \approx_{A\uparrow e} F$.

Proof of Lemma 19.

Equation 17 is adm-sound.

Proof. By Semantics 17, the denotation of an expression is the intersection of the denotations of all its possible instantiations. By Observation 5, all instantiations except $\vec{c}$ are independent of $e[\vec{c}]$. By admissibility, $e[\vec{c}]$ or $\overline{e}[\vec{c}]$ cannot occur again.

Proof of Lemma 24.

$E \approx_A E'$ does *not* imply that either of the following holds:
(a) $E \cdot F \approx_A E' \cdot F$
(b) $F \cdot E \approx_A F \cdot E'$

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

A Proofs of Important Results

Auxiliary Definitions 1

Definition 39 $(v \sqsupseteq w) \triangleq v$ contains the events of w in the same relative order.

Observation 54 If $v \sqsupseteq w$, then $w \in \Pi(D) \Rightarrow v \in \Pi(D)$. ∎

Observation 55 $\Pi(D \wedge E) = \Pi(D) \cap \Pi(E)$. ∎

Observation 56 $D \equiv E \not\Rightarrow \Pi(D) = \Pi(E)$. ∎

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 57 relies on the maximality of u.

Definition 40 $u \sim_i v \triangleq i \leq |u|$ and $i \leq |v|$ and $(\forall j : 1 \leq j \leq i \Rightarrow u_j = v_j)$.

Definition 41 $u \sim_i D \triangleq (\exists v \in [\![D]\!] : u \sim_i v)$.

Lemma 57 $u \sim_{|u|} D \Rightarrow u \in [\![D]\!]$. ∎

Lemma 58 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 35

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

Proof. Consider any dependency $E \in \mathcal{W}$. By Observation 34, $(\forall j : 1 \leq j \leq |u| \Rightarrow u \models_{j-1} \mathsf{G}_b(E, u_j))$.

By Observation 33, $u \models_0 \mathsf{G}_b(E, u_1)$ implies that $(\exists w : w \in \Pi(E)$ and $u \models_0 \mathsf{G}_b(w, u_1))$. By Lemma 58, $u \sqsupseteq w$. By Observation 54, $u \in \Pi(E)$. Therefore, by Lemma 31, $u \models E$. 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 34, $\mathsf{S}_b(\mathcal{W}) \rightsquigarrow u$.

Proof of Lemma 36

$(\exists k \le j : u \models_k M \text{ and } u \models_j G \div M) \Rightarrow u \models_j G$.

Proof. The proof is by induction on the structure of expressions. The base cases can be verified by inspection. Let $(\exists k \le j : u \models_k M$ 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 37

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

Proof. From Definition 29, 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 $\mathsf{S}_\div(\mathcal{W}) \leadsto_i u$. Then, $(\forall j : 1 \le j \le i \Rightarrow u \models_{j-1} \mathsf{S}_\div(\mathcal{W}, u_j))$. We prove by induction that $(\forall j : 1 \le j \le i \Rightarrow$
$u \models_{j-1} \mathsf{S}_b(\mathcal{W}, u_j))$. 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, $\mathsf{S}_b(\mathcal{W}) \leadsto_1 u$.

Assume that the inductive hypothesis holds for $1 \le l \le i$, i.e., $(\forall j : 1 \le j \le l \Rightarrow u \models_{j-1} \mathsf{S}_b(\mathcal{W}, u_j))$. We show that $u \models_l \mathsf{S}_b(\mathcal{W}, u_{l+1})$. $\mathsf{S}_\div(\mathcal{W}) \leadsto_{l+1} u$ 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 $\mathsf{S}_b(\mathcal{W}) \leadsto_l u$) holds iff $\mathsf{S}_b(\mathcal{W}) \leadsto_{l+1} u$, as desired.

If $\mathsf{S}_\div(\mathcal{W}, u_{l+1}, u, l{+}1) \ne \mathsf{S}_\div(\mathcal{W}, u_{l+1}, u, l)$, then $(\exists k \le l : u \models_k M$ and $\mathsf{S}_\div(\mathcal{W}, e, u, l{+}1) = (\mathsf{S}_\div(\mathcal{W}, e, u, l) \div M))$. By Lemma 36, $(\exists k \le l : u \models_k M$ 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 $\mathsf{S}_b(\mathcal{W}) \leadsto_{l+1} u$, as desired.

Auxiliary Definitions 2

The next theorems rely on additional auxiliary definitions and results. $I(w, \Gamma')$ gives all the superpaths of w that include all interleavings of w with the events in Γ'. We assume that $(\forall e : e \in \Gamma'$ iff $\overline{e} \in \Gamma')$. Lemma 59 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 60 shows that the guards are well-behaved with respect to denotations.

Definition 42 $I(w, \Gamma') \triangleq \{v : \Gamma_v = \Gamma_w \cup \Gamma' \text{ and } v \sqsupseteq w\}$.

Lemma 59 If $e \in \Gamma_w$, then $\mathsf{G}_b(w, e) = \bigvee_{v \in I(w,\Gamma')} \mathsf{G}_b(v, e)$. ∎

Lemma 60 $D \equiv E \Rightarrow \mathsf{G}_b(D, e) = \mathsf{G}_b(E, e)$. ∎

Proof of Theorem 38

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

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 42 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 54, $I(w) \subseteq \Pi(D)$. Also, $I(w) \subseteq \Pi(D \vee E)$. By Lemma 59, $\mathsf{G}_b(w, e) = \bigvee_{v\in I(w,\Gamma_{D\vee E}\setminus\Gamma_w)} \mathsf{G}_b(v, e)$. Thus the contribution of w to $\mathsf{G}_b(w, e)$ is covered by paths in $\Pi(D \vee E)$.

Proof of Theorem 39

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

Proof. $\mathsf{G}_b(D \wedge E, e) = \bigvee_{w\in\Pi(D\wedge E)} \mathsf{G}_b(w, e)$. By Observation 55, $\Pi(D \wedge E) = \Pi(D) \cap \Pi(E)$. 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 54, $x \in \Pi(D) \cap \Pi(E)$. By Observation 55, $x \in \Pi(D \wedge E)$. 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 61 $\mathsf{G}_b(D, e) = \mathsf{G}_b(D \wedge e, e)$. ∎

Lemma 62 $\mathsf{G}_b(e_1 \cdot e_2, e) = \Diamond(e_1 \cdot e_2)$, if $e, \overline{e} \notin \{e_1, e_2\}$.

Proof. Let $D^e = (e_1 \cdot \ldots \cdot e_n) \wedge e$. By Lemmas 41 and 61, $\mathsf{G}_b(D, e) = \bigvee_{w\in\Psi(D^e)} \mathsf{G}_b(w, e)$. Let $w = \langle e_1, \ldots, e_k, e, e_{k+1}, \ldots, e_n\rangle \in \Psi(D^e)$. Then

$\mathsf{G}_b(w, e) = \Box(e_1 \cdot \cdots \cdot \ldots \cdot \cdots \cdot e_k) \wedge \neg e_{k+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond(e_{k+1} \cdot \ldots \cdot e_n)$. There is one such w for each position of e, i.e., for $0 \leq k \leq n$. Thus,

$\mathsf{G}_b(D, e) = \bigvee_{0\leq k\leq n} \Box(e_1 \cdot \cdots \cdot \ldots \cdot \cdots \cdot e_k) \wedge \neg e_{k+1} \wedge \ldots \wedge \neg e_n \wedge \Diamond(e_{k+1} \cdot \ldots \cdot e_n)$. It is easy to verify that for any trace u and index i, $u \models_i \mathsf{G}_b(D, e)$ iff $u \models_i \Diamond D$. ∎

Proof of Theorem 42

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

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 62, $\mathsf{G}_b(D, e) = \Diamond D$. 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 43 $\mathsf{S}_p^\Delta(\mathcal{W}, e, u, 0) = \mathsf{G}_p^\Delta(\mathcal{W}, e)$.
$\mathsf{S}_p^\Delta(\mathcal{W}, e, u, i+1) \neq \mathsf{S}_p^\Delta(\mathcal{W}, e, u, i) \Rightarrow$
$e \notin \Delta$ and $(\exists k \leq i : u \models_k M$ and $\mathsf{S}_p^\Delta(\mathcal{W}, e, u, i+1) = (\mathsf{S}_p^\Delta(\mathcal{W}, e, u, i) \div M))$.

Proof of Theorem 43

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 $\mathsf{S}^{\Delta}_{p}(\mathcal{W}) \rightsquigarrow u$, 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 58, $u \sqsupseteq w$. Thus, $u \models D$. Thus no additional spurious traces are allowed by $\mathsf{S}^{\Delta}_{p}(\mathcal{W})$.

Definition 44 $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, 0) = \mathsf{G}^{\Delta}_{\wedge}(\mathcal{W}, e)$.
$\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, i+1) \neq \mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, i) \Rightarrow$
$(\exists k \leq i : u \models_k M$ and $\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, i+1) = (\mathsf{S}^{\Delta}_{\wedge}(\mathcal{W}, e, u, i) \div M))$.

Proof of Lemma 44

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 $[\![D]\!]$. $\mathsf{S}_{\div}(D) \not\rightsquigarrow \langle fegh \rangle$.

Proof of Lemma 45

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 $\mathcal{W}$.

Definition 45 $\mathsf{S}^{\Delta}_{\top}(\mathcal{W}, e, u, 0) = \mathsf{G}^{\Delta}_{\top}(\mathcal{W}, e)$.
$\mathsf{S}^{\Delta}_{\top}(\mathcal{W}, e, u, i+1) \neq \mathsf{S}^{\Delta}_{\top}(\mathcal{W}, e, u, i) \Rightarrow$
$(\exists k \leq i : u \models_k M$ and $\mathsf{S}^{\Delta}_{\top}(\mathcal{W}, e, u, i+1) = (\mathsf{S}^{\Delta}_{\top}(\mathcal{W}, e, u, i) \div M))$.

Proof of Theorem 46

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 33 and Lemma 41, we have that $\mathsf{G}^{\Delta}_{\top}(D, f) = \bigvee_{w \in \Psi(D)} \mathsf{G}^{\Delta}_{p}(w, f)$. 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 $\mathsf{S}^{\Delta}_{\top}(\mathcal{W}) \rightsquigarrow u$. 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, $\mathsf{S}^{\Delta}_{p}(\mathcal{W}) \rightsquigarrow v$. By Theorem 43, $v \models D$. By Observation 54, $u \models D$.

Proof of Theorem 50

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 $\mathsf{S}^{\Delta}_{\neg} \rightsquigarrow u \Rightarrow \mathsf{S}^{\Delta}_{\top} \rightsquigarrow u$. We establish this by induction. Clearly, $\mathsf{S}^{\Delta}_{\neg} \rightsquigarrow_1 u \Rightarrow \mathsf{S}^{\Delta}_{\top} \rightsquigarrow_1 u$. Let $D \in \mathcal{W}$. By the inductive hypothesis, assume $\mathsf{S}^{\Delta}_{\neg} \rightsquigarrow_k u$ and $\mathsf{S}^{\Delta}_{\top} \rightsquigarrow_k u$ and $u \not\models_k \mathsf{S}^{\Delta}_{\top}(\mathcal{W}, u_{k+1})$.

Assume $\mathsf{S}^{\Delta}_{\neg} \rightsquigarrow_{k+1} u$. 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 $\mathsf{S}^{\Delta}_{\top} \leadsto u$.

Proof of Theorem 49

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

Proof. $\mathsf{S}^{\Delta}_{\neg}(\mathcal{W})$ 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 47 $u \in [\![D]\!]$. Let $u_k \in \Gamma_D \setminus \Sigma_m$. Clearly, $u \models_{k-1} \mathsf{G}^{\Sigma_m}_{p}(D, u_k)$. This means there is a $v \in \Pi(D)$, such that $u \models_{k-1} \mathsf{G}(v, u_k)$. By Lemma 58, $u \sqsupseteq v$. Thus, $u \models D$. Hence, $\mathsf{S}^{\Delta}_{\neg}(\mathcal{W}) \leadsto u$.

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