A Customizable Coordination Service for 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 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.

1 Introduction

Open information environments are heterogeneous, distributed, dynamic, large, and frequently comprise autonomous components. For these reasons, they require solutions that marry artificial intelligence (AI) and traditional techniques to yield extensibility and flexibility. Agents are a result 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 [11, 19], 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. We believe it is the difficulty of constructing effective coordination that has led many researchers and practitioners to the centralized approaches.

To alleviate this problem, 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,

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

and include the functionality to enact them. This service should be customizable, because each application has its own requirements for coordination. This service should be minimally intrusive so as to preserve the autonomy of the participating agents. Such a service would help improve designer productivity. It would also help improve system efficiency by optimizing and enacting the desired coordination in a changing environment, potentially generating a different execution each time.

This is the kind of service we have developed. It 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.

Our service enhances techniques from workflow and relaxed transaction scheduling in databases. It is rigorous, being based on temporal logic. It includes abstractions for (a) a semantics of events in a multiagent system and (b) message passing to implement control and data flow [9]. Our approach is distributed and requires only limited knowledge of the agents' behavior.

As we show in section 3, our formal language is quite simple. 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. Similarly, we do not expect our service to replace more sophisticated approaches [6, 19], but to facilitate their robust implementation.

Section 2 motivates and presents our conceptual approach. Section 3 describes our algebra for specifying interactions and uses it to formalize an example from section 2, as well as the contract net protocol (CNP). Section 4 shows how the service operates. Section 5 reviews the pertinent literature.

2 A 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 [11]:

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, e.g., the price. ∎

Clearly, since the directory and suppliers are autonomous, so must their agents be. However, the agents must be coordinated to carry out the search.

2.1 Agent Events and Skeletons

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 the designer has some limited knowledge of the agents' designs. 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. We consider four kinds 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 is neither willing to delay nor omit
- *triggerable*, which the agent is willing to perform based on external request.

The first three categories are mutually exclusive; each can be conjoined with triggerability. Intuitively, immediate events are those that the agent performs unilaterally. 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.

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. Although traditional database approaches, e.g., [4], are limited to loop-free skeletons, which correspond to single-shot queries or transactions, we place no such restrictions here. Example 3 discusses two common skeletons in information search.

Example 3. Figure 1 shows two skeletons that arise in information search. The left skeleton 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 right skeleton is suited for agents who filter a stream or monitor a database. 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). In both skeletons, the application-specific computation

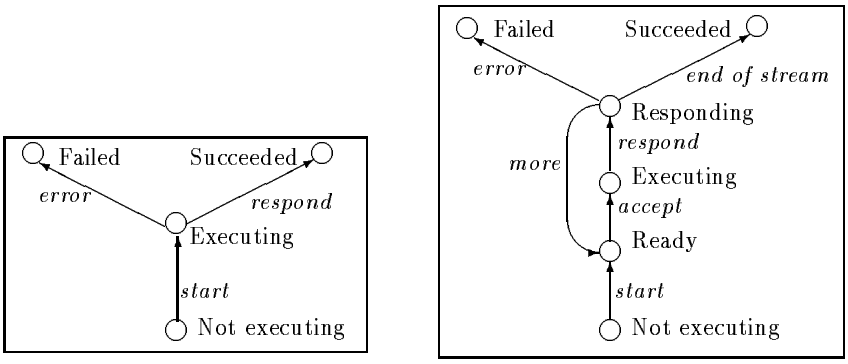

Fig. 1. Example Skeletons: (l) Simple querying agent; (r) Information filtering agent

takes place in the node labeled "Executing." We must also specify the categories of the different events. For instance, we may state that *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

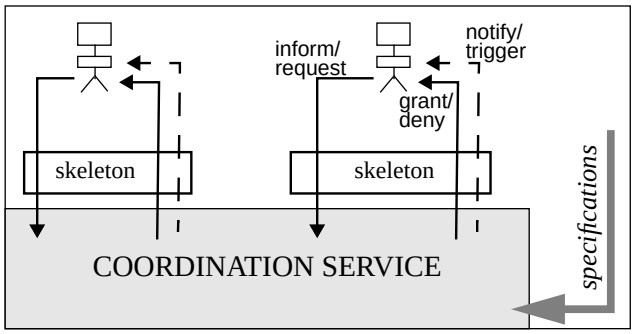

Fig. 2. The coordination service, logically

Figure 2 shows how the service interacts with agents. 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. Any necessary reasoning on intermediate results for decision-making is carried out through application-specific subtasks.

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.

3 Formalizing Coordination

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 token. *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.

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

Like for many process algebras, our formal semantics is based on traces, i.e., sequences of events. Our universe is $\mathbf{U}_{\mathcal{I}}$, which contains all consistent traces involving event tokens from Γ. Consistent traces are those in which an event token and its complement do not occur, and in which event tokens 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. $[\![e[c_1 \ldots c_m]]\!] = \{\tau \in \mathbf{U}_{\mathcal{I}} : e[c_1 \ldots c_m] \text{ occurs on } \tau\}$

$\overline{e}$ refers to the complement of e. 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. Complemented literals are included in Ξ and need no separate syntax or semantics rule.

$I(v)$ 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 4.2 presents a set of equations, which enable symbolic reasoning on $\mathcal{I}$ to determine when a certain event may be permitted, prevented, or triggered.

3.1 Specification

Our language allows a variety of relationships to be captured. We now consider some common examples. These assume that we are given events e, f, and g in different agents. The events all carry parameter tuples, but we don't show them below to reduce clutter. (We assume that $\cdot$ (before) has precedence over $\vee$ and $\wedge$, and $\wedge$ has precedence over $\vee$.) Some of these relationships are then applied on our running example.

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 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 if f doesn't happen: $(\overline{e} \vee f \vee g) \wedge (\overline{g} \vee e) \wedge (\overline{g} \vee \overline{f})$

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—which might restore consistency by undoing e. Notice that R7 and R8 involve events of three agents.

Example 4 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 4. 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 [19].

Assume all agents except M have skeletons as in Figure 1(l) 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 1(r). 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 $F_s[x]$. ∎

3.2 Contract Net Protocol

Our approach applies well to higher-level coordination protocols, such as the CNP [5]. 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, e.g., who to send the RFP to, whether to bid, and how to evaluate bids.

Example 5. 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. $A_{rfp}[m\ t\ c$ info$]$ initiates $A_{think}[c\ t\ m$ info$]$
D7. $A_{bid}[c\ t\ m$ bid$]$ conditionally feeds $A_{eval}[m\ t\ c$ bid$]$
D8. $A_{award}[m\ t\ c$ task$]$ initiates $A_{work}[c\ t\ m$ task$]$.

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

4 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, 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.

4.1 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 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. This semantics characterizes progress along a given computation and uses it to determine the decision on each event. Our semantics has important differences from traditional linear temporal logics [7]. 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.

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 6, 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 8, 9, 11, and 12 are as in traditional formal semantics. Semantics 13 and 14 involve looking into the future. Semantics 7 and 10 capture the dependence of an expression on the immediate past, bounded by the first index of the semantic definition. Semantics 10 introduces a nonzero first index.

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 $u \models_{i,j} E)$

4.2 Calculating Guards

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.

Interestingly, $/$ can be computed symbolically. We propose a set of equations exists 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 $f \equiv f \cdot \top$). $\Gamma_E \triangleq \{e : e \text{ or } \overline{e} \text{ occurs in } E\}$. (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. $(\overline{e} \cdot E)/e = 0$

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 succinct for the common cases, such as the relationships of section 3.1.

Definition 1. 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 6. We compute the guards for the events in R2 as follows:

- $\mathsf{G}(\text{R2}, e) = \Diamond\overline{f} \vee \Box f$.
- $\mathsf{G}(\text{R2}, \overline{e}) = \top$.
- $\mathsf{G}(\text{R2}, \overline{f}) = \top$.
- $\mathsf{G}(\text{R2}, f) = (\Diamond\overline{e} \vee (\neg e \wedge \Diamond e))$, 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). ∎

4.3 Scheduling with Guards

Execution with guards is straightforward. When an event e is attempted, its guard is evaluated. Since guards are updated whenever an event mentioned in them occurs, evaluation usually means checking if the guard evaluates to $\top$. If e's guard is satisfied, e is executed; if it is 0, e is rejected; else e is made to wait. Whenever an event occurs, a notification is sent to each pertinent event f, whose guards are updated accordingly. If f's guard becomes $\top$, f is allowed; if it becomes 0, f is rejected; otherwise, f is made to wait some more. Example 7 illustrates this. The correct disablement interpretation of R2 also requires setting the categories of the events appropriately, which we lack the space to discuss.

Example 7. Using the guards from Example 6, 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. Events $\overline{f}$ and $\overline{e}$ have their guards equal to $\top$, so they can happen at any time. ∎

Abstractly, given a workflow, 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 [18].

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

Assimilating Messages 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 an event occurs or is promised.

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.

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,

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 1. Assimilating Messages

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 $u \models_j G \div M) \Rightarrow u \models_j G$. ∎

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 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.

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.

Example 8. Following Example 6, 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.

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 9. Consider $\mathcal{W} = \{\mathrm{R1}, \mathrm{R2}\}$. $\mathsf{G}(\mathcal{W}, e) = \Diamond f \wedge \neg f$ 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 9 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.

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 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. Our approach introduces traditional scheduling ideas into an environment of autonomous agents without requiring unnecessary control over their actions, or detailed knowledge of their designs. In our present approach, 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 when the coordination requirements are based on the agents' social commitments [17].

The relevant previous tools for developing multiagent systems are either not formal, are centralized or violate the autonomy of agents. AgenTalk [13] gives a programming environment, but no formal semantics. Kabanza [12] 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. Traditional temporal logic approaches do not apply here. Such 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.

Sycara & Zeng [19] articulate many of the intuitions that we share, 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* [14] propose an approach for planning searches. However, their approach does not have an explicit representation of search patterns, and does not apply generically. The search techniques are captured as different search patterns in our approach. Decker & Lesser [6] 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. We believe that our approach can help encode their intuitions in a rigorous setting. Our approach complements the above, because they develop semantic representations, whereas our approach focuses on the activity management infrastructure itself.

There is also work on the lower-level aspects of providing robust infrastructures for implementing multiagent systems, e.g., [1]. We believe this work is important, and can in principle be used to support the kind of approach developed here.

High-level abstractions for agents have been intensively studied, e.g., [15]. Formal research on interactions among agents includes [8]. These approaches develop formal semantics, but do not give as precise an operational characterization. The present work has a formal semantics along with an operational interpretation. There has been much work on social abstractions for agents, e.g., [3]. We believe that the present infrastructure will facilitate the development of a computational treatment of the social constructs by capturing the mechanics of possible interactions in a succinct manner. Including mental and social abstractions into a generic executable system is an important open problem.

We prototyped our approach initially in an actor programming language. We are now reimplementing it with enhancements in Java. One of the enhancements being developed 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 [16], but not a coordination service of the sort we described.

References

1. Arvind K. Bansal, Kotagiri Ramamohanarao, and Anand Rao. Distributed storage of replicated beliefs to facilitate recovery of distributed intelligent agents. In this volume.
2. Alan Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Francisco, 1988.
3. Cristiano Castelfranchi. Commitments: From individual intentions to groups and organizations. In *Proceedings of the International Conference on Multiagent Systems*, pages 41–48, 1995.
4. Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.
5. Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983. Reprinted in [2].
6. 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.

7. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. North-Holland, Amsterdam, 1990.
8. Afsaneh Haddadi. Towards a pragmatic theory of interactions. In *Proceedings of the International Conference on Multiagent Systems*, pages 133–139, 1995.
9. Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.
10. Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1997.
11. Michael N. Huhns, Munindar P. Singh, Tomasz Ksiezyk, and Nigel Jacobs. Global information management via local autonomous agents. In *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, August 1994.
12. Froduald Kabanza. Synchronizing multiagent plans using temporal logic specifications. In *Proceedings of the International Conference on Multiagent Systems*, pages 217–224, 1995.
13. Kazuhiro Kuwabara. Meta-level control of coordination protocols. In *Proceedings of the International Conference on Multiagent Systems*, pages 165–172, 1996.
14. Tim Oates, M. V. Nagendra Prasad, and Victor R. Lesser. Cooperative information gathering: A distributed problem solving approach. TR 94-66, University of Massachusetts, Amherst, MA, 1994.
15. 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 [10].
16. Jon Siegel. *CORBA: Fundamentals and Programming*. Object Management Group and Wiley, New York, 1996.
17. 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.
18. Munindar P. Singh. Coordinating heterogeneous autonomous agents. TR 97-07, Department of Computer Science, North Carolina State University, Raleigh, July 1997. Available at www.csc.ncsu.edu/ faculty/ mpsingh/ papers/ mas/ coord-tr.ps.
19. 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.