Synthesizing Coordination Requirements for Heterogeneous Autonomous Agents

MUNINDAR P. SINGH singh@ncsu.edu
Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, USA

Abstract. As agents move into ever more important applications, there is a natural growth in interest in techniques for synthesizing multiagent systems. We describe an approach for engineering the coordination requirements of a multiagent system based on an analysis of conversation instances extracted from usage scenarios. This approach exploits the notion of Dooley graphs that were recently introduced to the multiagent systems community from the linguistics and discourse analysis literature. We show how, with a few key modifications, Dooley graphs can be used to generate coordination requirements and constraints on the behavior models of the agents participating in a multiagent system.

Our present approach is embodied in the context of our recent work on a distributed coordination service for heterogeneous, autonomous agents. This approach takes as input (a) agent *skeletons*, giving compact descriptions of the given agents in terms of their events that are significant for coordination, as well as (b) *relationships* among the events occurring in these skeletons. A natural question is how may the skeletons and relationships be produced in the first place. It turns out that a methodology that begins with Dooley graphs can readily yield the skeletons and relationships needed to achieve the desired coordination.

Consequently, our approach combines the benefits of an intuitive methodology with a formal and distributed framework for developing multiagent systems from autonomous agents.

Keywords: Coordination; development and engineering methodologies.

1. Introduction

For the past several years, agents have been steadily moving into more and more significant applications [4]. Among the benefits of using agents is that they more naturally support the development of software systems whose components are heterogeneous (built in different ways), and autonomous (representing different interests). These properties make agents ideally suited to applications in electronic commerce, virtual enterprises, and other open settings [14]. In these applications, agents must work in cooperation with traditional systems. Because of the importance of these applications and the risks of developing invalid systems, techniques for building agents must compare well with the techniques for building traditional software systems. There is thus a major need for industrial-strength approaches for engineering agent-based systems. Indeed, what distinguishes engineering from science is the use of rigorous tools and sound methodologies for the construction of solutions. Many current tools take coordination requirements as input. We develop an approach through which such requirements may be synthesized.

Of all the kinds of agent-based systems, multiagent systems are the most interesting for two reasons. One, the interaction that is inherent in multiagent systems distinguishes them from other kinds of software systems, even single-agent systems. For instance, the dozen traditional software architectural styles cataloged by Shaw & Garlan [27] do not include anything that is at the level of abstraction of communication protocols among agents. In this paper, we treat agents as persistent computational objects that can perceive, reason, act

in their environment and interact with other agents [17]. The chief mode of interaction that we consider is communication. Multiagent systems thus consist of several communicating agents; we do not consider single-agent systems here.

Two, the challenges posed by their construction are qualitatively different, and we believe harder, than the challenges posed by the construction of traditional distributed systems. This is largely because of the above properties of multiagent systems, specifically because of two reasons. Multiagent systems involve interaction in the face of heterogeneity and autonomy. Further, conventional ways of software engineering even lack the abstractions necessary to understand and model multiagent systems. As a result, tools and methodologies for traditional software systems, while sometimes applied to multiagent system design, leave room for several enhancements.

1.1. Traditional Software Engineering

The problem of building complex systems has of course been intensively studied in the software engineering community. Most pertinent for our present discussion is the previous work on methodologies in object-oriented analysis and design. This is so for two main reasons. One, the object-oriented approaches are the most recent and sophisticated of the software engineering approaches. Object-oriented modeling languages too have evolved recently, culminating in the unified modeling language (UML) [12]. Two, objects are closely related to agents; in fact, agents are frequently realized as objects with some additional properties.

Excellent summaries of the software engineering approaches are available in textbooks by Texel & Williams [37] and Pressman [25, chap. 20]. We follow Pressman more closely in the following discussion. Approaches such as class diagrams that are geared for some aspects of design and programming are not directly relevant here; we emphasize the behavioral and interactive aspects of agent designs. Class diagrams consider design at the micro level, whereas the interest in multiagent systems is at the macro level. Four main kinds of software engineering approaches, designed for the behavioral and interactive aspects of objects, are relevant here.

- Use cases involve the capture of how a system will be used and the different user roles that will invoke its functionality. Use cases are a good first step toward designing a system, but they are usually too sparse to help in a detailed analysis. For detailed analysis, the system builder must construct a representation such as an activity diagram, which can describe the computations as they affect the macro functionalities of the system.

- The class responsibility collaboration (CRC) model indicates for each class the methods for which it is responsible and the other classes on which it might invoke methods. In this way, the "collaboration" is really only a client-server relationship; it is not sufficient to capture true peer-to-peer collaboration as studied in the multiagent systems community.

- The behavior of an object may be described using a representation such as statecharts [16, 15]. A statechart shows the states of an object along with transitions among them. The transitions take place when an event occurs and a specified condition is true; in

AAMAS, , 3–25 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

that case, an additional action may also be performed. Interestingly, the usual way to apply statecharts for behavior modeling assumes that the transitions are driven only from external events. This is because, unlike agents, objects are not expected to behave proactively.

- Event traces show how the events on different objects in a system are ordered; the events correspond to messages among the classes. These are related to the lowest level descriptions of conversations, which we will take as input. Event flow diagrams show the messages that different classes may exchange; these are a somewhat more static representation than event traces. Sometimes, the messages to which a class responds are called "protocols," but these lack the structure that multiagent (or even network) protocols carry.

Pressman describes the common approach of all software methodologies. The existing methodologies all involve (a) capturing requirements through use cases, (b) developing class diagrams, and (c) determining object behavior. By contrast, our approach involves (a) beginning with use cases of conversations, (b) determining the agent skeletons and relationships, (c) fleshing out the agent designs to meet the skeletons, which might involve the development of class diagrams. Going top-down by beginning with interactions, as in our approach, emphasizes the modularity that is one of the attractions of agent technology.

1.2. Interaction-Oriented Programming

Our main motivations therefore are to study the macro aspects of multiagent systems construction and especially to do so in the context of characterizing interactions among autonomous and heterogeneous agents. With these motivations, we have been pursuing a program of research on *interaction-oriented programming (IOP)*. IOP seeks to develop techniques and tools for the construction of multiagent systems by specifying the interactions among (usually) autonomous agents. IOP has three main components or layers: coordination, commitments, and collaboration. Coordination deals with how the agents operate in a shared environment. Coordination enables the construction of multiagent systems whose constituent agents are properly orchestrated [32]. Commitments capture the agents' obligations to one another, so as to realize the organizational structure of a multiagent system [30, 19, 38]. Collaboration deals with how the agents carry out higher-order activities, such as teamwork and negotiation [33]. We lack the space to review all of IOP here, but additional details may be found in the above references.

Here we concentrate on the coordination layer. Specifically, we consider the problem of creating specifications for agent behavior and interaction to achieve the necessary coordination to support various kinds of communicative or "conversational" interactions. Since communication is inherently reliant on the agent's commitments, our approach naturally leads us to a discussion of commitments as well.

As part of IOP, we developed an approach for coordinating heterogeneous, autonomous agents [32]. Our approach specifies individual agents in terms of *skeletons*, which give coarse descriptions of the agents' behavior. Being coarse, these descriptions succinctly capture the essential aspects of the agent's behavior that are significant to their potential

coordination with other agents. This lack of detail is essential to achieving the heterogeneity of agents—we don't know and don't care how an agent may be implemented as long as it satisfies the rather minimal interface specified by a suitable skeleton.

Desired coordinations are specified by stating *relationships* among the events of different agents. The events are mostly actions but in some cases could also be observations made by the agents. These relationships are expressed in a formal language based on temporal logic, and can be automatically processed to yield distributed means of coordinating the events of different agents. The key motivation for these relationships is that they specify the constraints on an agent's interactions; beyond these, the agent is fully autonomous. Applying the coordination layer of IOP requires a means to create specifications of the desired coordination intuitively and correctly.

We consider Dooley graphs from discourse analysis [7], which were recently introduced to the multiagent community by Parunak [24]. Dooley graphs offer a vivid representation of conversation instances. Interestingly, given a conversation, its graph can be used to generate the skeletons and relationships that are required by our approach. To do so, a Dooley graph is processed to highlight the causal relationships among actions, and the structural properties of the interactions of agents. As a result, as much information as can be gleaned from a conversation is extracted from it.

Although we develop this approach in the context of our approach to coordination, we believe its ideas will apply wherever the coordination of heterogeneous and autonomous agents must be specified. In fact, a number of current approaches to protocols or conversation policies are based on a few basic kinds of finite representations of the agents' behaviors [20, 6]. These approaches assume that the protocol specification is given; by contrast, our approach seeks to develop such specifications. In this way, our approach is complementary to most existing approaches and can in principle be combined with any of them to support their usage.

There has been a large amount of good work on some related areas, especially agent communication languages and protocols. We shall not discuss that lively topic in any detail here—we discuss some of the challenges in [31]. For simplicity and ease of presentation, we follow Parunak's classification of speech acts and his set of "relationships" among utterances. However, we believe that our approach can be applied in other settings as well, provided they can identify the different "characters" played by an agent in a conversation. (The quoted terms are explained below.)

The rest of this paper is organized as follows. Section 2 describes the concepts of our coordination approach. Section 3 presents a brief exposition of Dooley graphs. Section 4 shows how we carry out the synthesis by working out an example from [24] to convert a Dooley graph into a set of agent skeletons. Section 5 discusses two important enhancements to the approach. Section 6 discusses the relevant literature.

2. Coordination

For concreteness, we develop our approach in the context of our previous formal work on coordination among agents. We now summarize the key concepts of our coordination approach. Additional details are available in [32]. Our coordination approach deals with the problem of creating and realizing formal specifications of the coordination that is desired

in a multiagent system. This aspect of coordination is of necessity at a lower level than the various advanced heuristic approaches for coordination, for example, as reviewed by Durfee [9].

2.1. Coordination Metamodel

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. In order to be able to coordinate the agents, the designer of the multiagent system must have some knowledge of the designs of the individual agents. Ideally, to maximize the agents' heterogeneity, this knowledge should be as limited as possible. However, the designer will need to know the agents' externally visible events, which are potentially significant for coordination. These events include the agents' actions as well as their significant observations. In other words, the only events we speak of are those that are publicly known—the rest are of no concern to the coordination service.

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 if requested.

The first three classes are mutually exclusive; each can be conjoined with triggerability. The category where an agent will entertain omitting but not delaying an event is empty, because unless the agent performs the event unilaterally, there must be some delay in receiving a response from the service. In the present version, we assume that the event classes are set at design time. However, in principle, the classes of events could change during execution, which can make sense if the coordination specifications are also modified. There is a simple ordering among the first three classes: from the perspective of the coordination service, immediate events are the most restrictive and flexible events are the least restrictive. Thus, if an event may sometimes be executed as an immediate event, it must be modeled as an immediate event at design time.

Based on the events, our metamodel involves constructing behavioral models for each of the agents. The events of an agent are organized into a *skeleton* to provide a simple model of the agent for coordination purposes. Skeletons are well-known from logics of program, especially since Emerson & Clarke [10]. The skeletons are typically finite state automata. That is, in the diagrams below, the nodes correspond to abstract states of the agent and the transitions to actions by the agent. The states in the skeleton are abstract in that each may correspond to a large number of computational states in the agent that are considered alike for the purposes of coordination. Each skeleton corresponds to one or more threads in the agent—one thread per connected component.

Although we consider finite state skeletons, they are not restricted by our formal system and implementation. Neither the formal system nor the implementation looks at the structure of the skeletons. In particular, the skeletons may be *sets* of finite state automata, which can be used to model the different threads of a multithreaded agent. The set of events, their properties, and the skeletons of the agents are usually realized by an agent and, if so, in an application-specific manner. These can be viewed as requirements that are set by the protocol in which the designer wishes the agents to participate. The next example discusses two common skeletons.

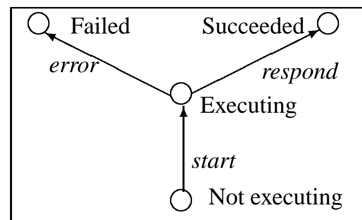

Figure 1. Example skeleton for querying

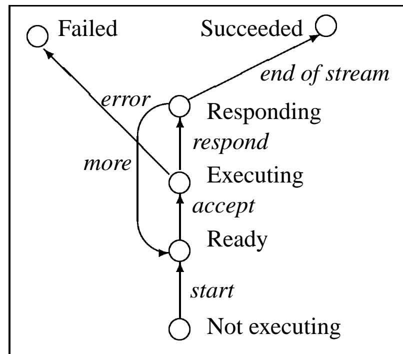

Figure 2. Example skeleton for information filtering

EXAMPLE: Figures 1 and 2 show two skeletons that arise in information search. The skeleton of Figure 1 suits agents who respond to one-shot queries. Its significant events are *start* (accept an input and begin), *error*, and *respond* (produce an answer and terminate). The skeleton of Figure 2 suits 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), and *more* (loop back to expecting more input). In both skeletons, the application-specific computation 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. □

Although the skeleton is not used explicitly by the coordination service during execution, it can be used to validate specified coordination requirements. More importantly, the skeleton is essential for understanding the public behavior of an agent, and for giving intuitive meaning to its actions.

2.2. *Coordination Relationships*

Ultimately, to create a multiagent system, we must not only specify the skeletons for different agents, but also show their actions are coordinated with each other. Coordinations are specified by expressing appropriate temporal relationships among the events of different agents. For example, the events may have to be ordered in a certain way or the occurrence of an event in one agent may necessitate or preclude an event in another agent.

Table 1. Example coordination relationships

	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	$(\overline{e} \vee f \vee g) \wedge (\overline{g} \vee e) \wedge (\overline{g} \vee \overline{f})$

Table 1 presents some common examples of coordination relationships from [32]. Some of the relationships involve coordinating multiple events. For example, 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 a typical pattern in 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*.

Our formal language allows an even richer variety of coordination relationships to be captured [32]. We include the formal syntax and semantics of this language in Appendix A. However, for the purposes of designing multiagent systems, we will generally restrict ourselves to some carefully chosen set of coordination relationships or patterns. The set of Table 1 is particularly effective in most typical cases that we have encountered.

3. Dooley Graphs

Our presentation of Dooley graphs is based on the exposition in [24] with some enhancements. The key idea that interests us here is that Dooley graphs provide a natural way to present a specific instance of a conversation. By concentrating on specific conversations, Dooley graphs can separate the different *characters* played by a single agent. The characters can correspond to different components in an agent—roughly, this is what interests Parunak. However, we are also interested in the structure imposed on an agent's skeleton by the characters it plays. Most importantly, the interactions among the characters lead to coordination relationships among the skeletons.

Agents act, both communicatively (that is, using speech acts [1]) and physically. We are interested in the agents' interactions with one another. Typically, the agents' interactions do not arise in isolation, but as parts of extended communicative activities. These activities can be thought of as protocols, dialogues, arguments, or negotiations among agents. A *conversation* is a specific instance of these composite activities.

Conversations naturally include not only speech acts, but also some physical actions by means of which the agents deliver on their promises. Parunak allows the speech acts of Solicit (Request or Question) or Assert (Inform, Commit, and Refuse). He allows two physical acts: Ship and Pay.

In addition to the acts in a conversation, there is also knowledge of certain relationships among the speech acts. These relationships are restricted to be one of the following. Here, u_i and u_j refers to different utterances in a conversation. S_i refers to the sender of u_i.

- *Respond*. u_i responds to u_j iff (a) S_i previously received u_j, (b) u_j's impact on S_i caused S_i to send u_i, and (c) u_i is the first utterance of S_i to satisfy (a) and (b).
- *Reply*. u_i replies to u_j iff (a) S_i previously received u_j, (b) u_j's impact on S_i caused S_i to send u_i, and (c) u_i is the first utterance of S_i directed to S_j that satisfies (a) and (b).
- *Resolve*. u_i resolves u_j iff u_i replies to u_j and u_i follows the "rules of engagement" defined in u_j.
- *Complete*. u_i completes u_j iff u_j is a Commit and u_i either satisfies or cancels the associated commitment.

Respond, Reply, and Resolve are progressively more restrictive. Complete is mutually exclusive with Resolve—an act cannot both complete an utterance and resolve an utterance (not even a different one).

EXAMPLE: Consider a request for proposals (RFP) from A to B, C, and D. The first act that any of them does that was caused by the RFP is a Response to it. If it is a message back to A, then it is also a Reply. If the Reply is a Commit or a Refuse, then it is also a Resolve. □

EXAMPLE: Table 2 shows an example conversation from [24]. The #s partially order the utterances from early to late. In this conversation, A announces an RFP for 50 widgets to

Table 2. Example conversation

#	S	R	Utterance	Respond to	Reply to	Resolve	Complete
1	A	B,C,D	Request (RFP for 50)				
2	B	C	Question: bidding?	1			
3	C	B	Inform: yes	2	2	2	
4	B	A	Refuse	3	1	1	
5	C	A	Propose (take 40)	1	1		
6	A	C	Request (send 40)	5	5	5	
7	C	A	Commit (deliver 40)	6	6	6	
8	D	A	Commit (deliver 50)	1	1	1	
9	A	C	Assert (decline)	7, 8	7		
10	C	A	Refuse	9	9		7
11	D	A	Ship (deliver 45)	1	1		8
12	A	D	Assert (short) + Request	11	11		
13	D	A	Ship remainder, i.e., 5	12	12	12	
14	A	D	Pay	13	13	13	

B, C, and D. B checks with C is C is bidding. C says it is. B then refuses A. C, however, makes a counter offer of 40 widgets. A accepts and C commits. In the meanwhile D offers to accept the initial RFP, which is more preferable to A. A then declines C, who cancels its commitment. D delivers, but the order is short (45 only). A informs D of the shortfall. After D sends the remainder of the order, A pays D. Table 2 also shows the discourse relations among the utterances.

This example is oversimplified in that D commits to supplying the widgets without bothering to check if A actually accepted its bid. However, this and other simplifications won't affect the main thrust of our paper. □

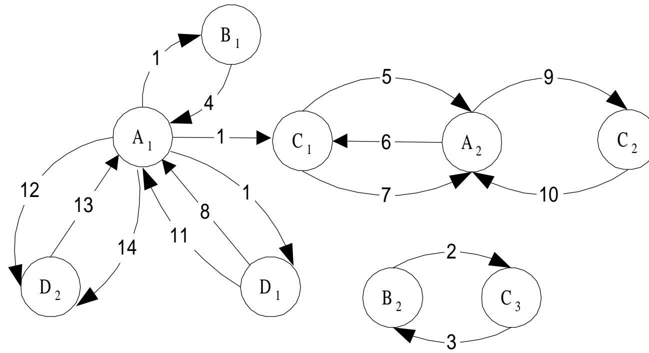

Figure 3. Example conversation as a Dooley graph

A Dooley graph is generated by analyzing a conversation in such a manner that the sets of utterances that are closely related to one another are brought closer. The conceptual idea behind the construction of Dooley graphs can be phrased as follows. Although the following construction based on graph theory produces the same results as Parunak's construction, we believe it is conceptually more perspicuous.

First, construct an auxiliary bipartite graph each of whose two partitions of vertices equals a copy of the set of utterances in the given conversation. Each vertex in the left partition corresponds to the sender of the given utterance and each vertex in the right partition corresponds to the receiver of the given utterance. Second, relate an agent showing up as a sender on the left with the same agent showing up as a receiver on the right, provided some additional property (discussed below) is met. By our construction, this binary relation will relate the same agent engaging as sender or receiver in two different utterances. The exact basis for this relation depends on how we model the relevance of utterances to each other (see below). Third, identify the connected components of the bipartite graph. Each connected component will involve one or more occurrences of the same agent playing the role of sender or receiver in a set of utterances that are somehow related. Thus, each connected component corresponds to a *character*. Fourth, assemble the characters into a new graph—the Dooley graph—by making the characters into distinct vertices and making the utterances into edges between the characters that sent or received them.

The above approach leaves open the specification of the relevance relationship among the vertices in the bipartite graph. The simplest idea (following Dooley and Parunak) is to relate a sender-side vertex s with a receiver-side vertex r if and only if the two vertices involve the same participant and one of the following four conditions hold:

- s replies to r
- r resolves s
- r is the last utterance in the conversation and r replies to s
- r completes t where t resolves s.

EXAMPLE: Figure 4 shows the bipartite graph derived from the example conversation under the above notion of relevance. □

The characters reflect the rhetorical structure of the conversation. In a highly coherent conversation, each participant will yield exactly one character; in an entirely disjointed conversation, there could be as many characters as there are utterances. Most practical situations would lie somewhere in between.

EXAMPLE: Figure 3 gives the Dooley graph for Table 2. The numbered utterances relate the characters that send and receive them. □

4. Approach

Dooley graphs highlight the rhetorical structure of a conversation, but hide its causal structure. In other words, information about the control flow among the agents is lost where more than one character of an agent is involved. (Parunak's proposed extension to Dooley graphs also does not display the actual causal connections, and we do not consider it in detail here.) By reconstructing the causal structure of the given conversation, we are more naturally able to use Dooley graphs to produce the coordination requirements for the given multiagent system.

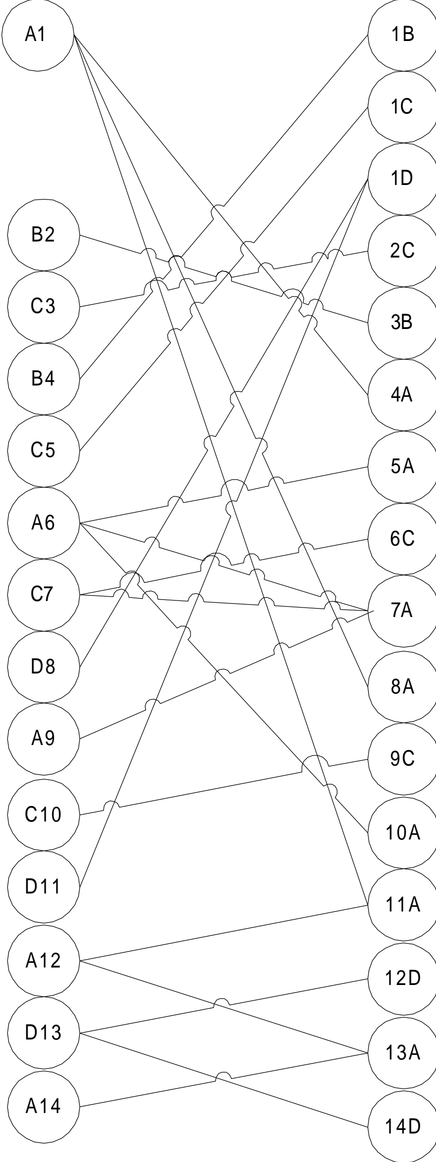

Figure 4. Bipartite graph based on example conversation

Our approach proceeds as follows. We begin with a Dooley graph depicting the conversation being analyzed. We analyze this Dooley graph to explicitly identify the causal relationships among the various utterances. We separate out the histories of the different

participants, but record the contribution of each character. The different characters are highlighted in each history.

Table 3. The histories of agents in a conversation

Role	History
B	$(A_1, 1, B_1)$; $\boxed{(B_2, 2, C_3); (C_3, 3, B_2)}$; $(B_1, 4, A_1)$
C	$(A_1, 1, C_1)$; $\boxed{(B_2, 2, C_3); (C_3, 3, B_2)}$; $(C_1, 5, A_2)$; $(A_2, 6, C_1)$; $(C_1, 7, A_2)$; $\boxed{(A_2, 9, C_2); (C_2, 10, A_2)}$
D	$(A_1, 1, D_1)$; $(D_1, 8, A_1)$; $(D_1, 11, A_1)$; $\boxed{(A_1, 12, D_2); (D_2, 13, A_1); (A_1, 14, D_2)}$
A	$(A_1, 1, B_1)$; $(A_1, 1, C_1)$; $(A_1, 1, D_1)$; $(B_1, 4, A_1)$; $\boxed{(C_1, 5, A_2); (A_2, 6, C_1); (C_1, 7, A_2)}$; $(D_1, 8, A_1)$; $\boxed{(A_2, 9, C_2); (C_2, 10, A_2)}$; $(D_1, 11, A_1)$; $(A_1, 12, D_2)$; $(D_2, 13, A_1)$; $(A_1, 14, D_2)$

EXAMPLE: Table 3 shows the histories derived from the Dooley graph of Figure 3. Notice that except for role D, the histories of the different characters of a role are not contiguous. For instance, role B goes from character B_1 to B_2 and then back to B_1. □

4.1. *Inducing Agent Skeletons*

Now we discuss how to induce agent skeletons from the histories produced in the previous step.

To induce the skeletons, it is helpful to think of the events (utterances sent or received) that are observed by the local models. Figure 5 displays the local histories for each agent with the different characters separated. For instance, agent B is involved in four utterances; each of its characters being engaged in two of them.

We use the following conventions in the following proposed skeletons. An event type named "get X" corresponds to the receipt of an utterance, whereas the event named "X" corresponds to the making of that utterance. We would thus expect to see complementary event types in at least two roles, where one sends and the other receives the given type of utterance. There is no assumption that the two complementary events happen in synchrony, and usually they would not, because most real-life multiagent systems are asynchronous. In the skeletons, we parenthetically show the corresponding utterance number from Table 2. A star (*) marking a transition indicates an action not in the given conversation, but one that is inferred based on the designer's knowledge, and usually motivated on grounds of making the skeleton more flexible or reusable.

Figures 6 and 7 show two possible strawman skeletons for B. The skeleton of Figure 6 requires the agent to consult C before deciding whether to propose. It is as if this skeleton merges the two characters of agent B. This skeleton would be inappropriate in most settings, because it puts strong constraints on B's design. The skeleton of Figure 7 goes even farther and requires B's decision to depend on C. This skeleton is less intuitive, because it

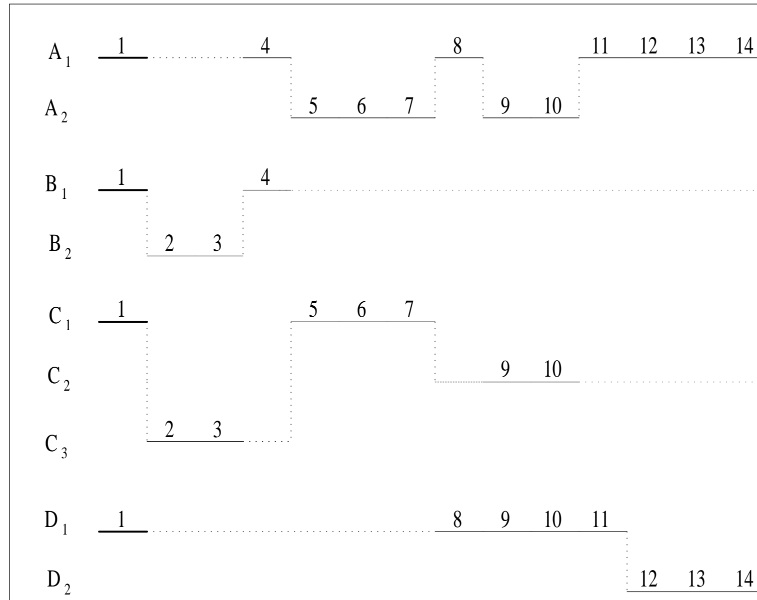

Figure 5. The local observations by each agent, partitioned by character

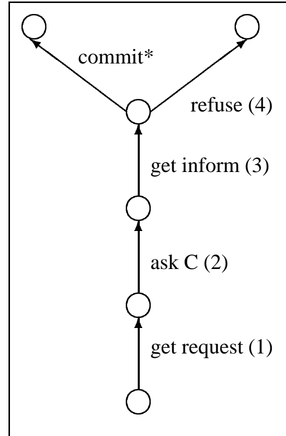

Figure 6. Possible skeleton for agent B: required consultation

essentially requires that utterance 4 be treated as part of the same character as utterance 3, which it is not.

The above skeletons place B's decision-making publicly in the protocol, and are clearly unacceptable. B's query to C is publicly in the protocol, but what B does with the answer from C is B's own business. By contrast, the skeleton of Figure 8 leaves it up to B to decide

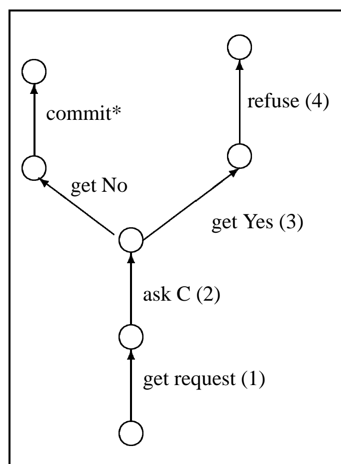

Figure 7. Possible skeleton for agent B: dependence of decision on consultation

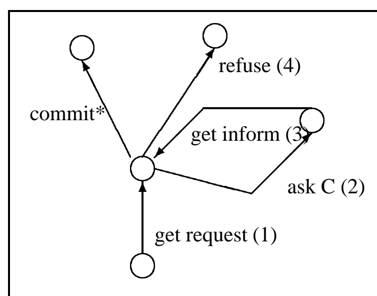

Figure 8. Possible skeleton for agent B: optional consultation

whether to consult C and how to use its response. This skeleton captures the key intuition about the two characters of B, namely, that character B_2 is separate and that it engages in a subdialogue with character C_3. This justifies selecting the skeleton of Figure 8 for B.

Notice that when B asks C, C's response is relevant to B's further actions. However, when B asks C, this query may have no consequence on C's actions (and, in this protocol, it doesn't). Consequently, Figure 9 shows a skeleton for C in which C may get a query from B, but this query is structurally independent of how C handles RFPs. Similarly, the counter-proposal is kept as a separate loop but attached to the main flow. This too is an example where a character is modeled with a separate subskeleton (physically a thread) in the agent's skeleton. (For reasons of brevity, D is discussed when integrated with the other contractors below.)

The decision whether to have a separate thread or a loop in a single thread depends on how we understand the agents to be acting and interacting. Clearly, we must separate what

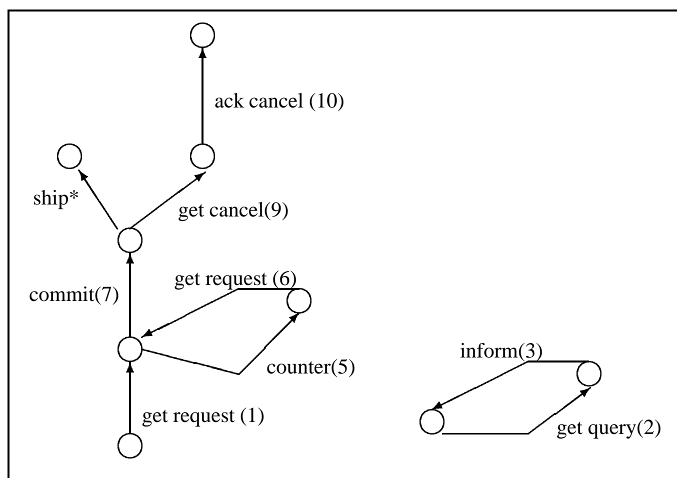

Figure 9. Possible skeleton for agent C

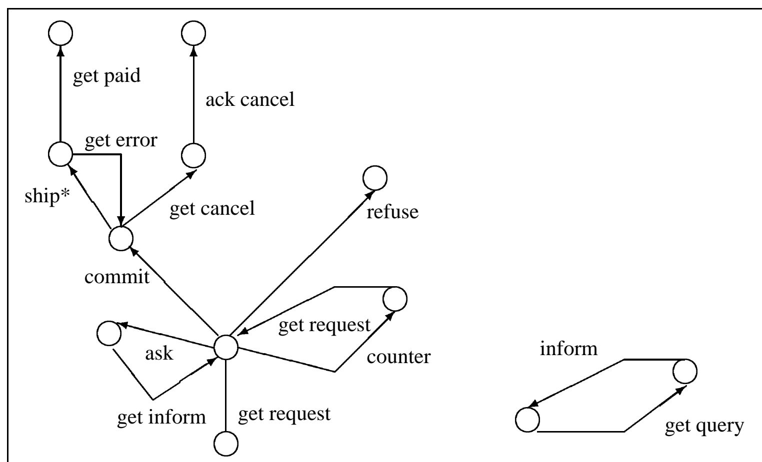

Figure 10. Integrated skeleton for all contractors

the agents happen to do from what is essential for coordination in the given application. Dooley graphs, by focusing on a specific conversation, are in tension with this process. However, in settings such as our present example, we can derive more information from the graph by recognizing that the same role is instantiated by multiple agents. Here, the multicast by A is a clue that B, C, and D are to be treated alike. In such a case, we can achieve the correct solution by integrating the skeletons of B, C, and D. Figure 10 shows a composite skeleton assuming B, C, and D play the role of contractor. By integrating the skeletons, we can construct a single more complete skeleton than any of the agents

in the given conversation indicates. The ship and get-error loop refers to character D_2 of Figure 3. In this case, assigning it a separate loop in the skeleton would have caused the ship action to appear on two different transitions, and would have been less clear.

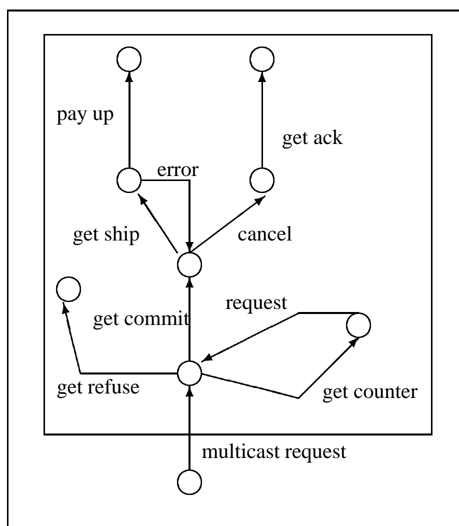

Figure 11. Skeleton for agent A

Figure 11 shows the skeleton for A. The main quirk in this skeleton is that A performs a multicast, and effectively keeps a separate thread to deal with each contractor. Note that it is not clear if only one bid can be accepted, because the bids may each be partial and may need to be combined. If there were a requirement of uniqueness, it would be captured as a disabling relationship (*a la* R2 in Table 1).

4.2. Inducing Event Classes

With the skeletons in hand, the properties of the events can be readily inferred. The “get” events for requests, queries, or cancelations are all triggerable, because that is how the agent is informed by others and requested to perform various actions. In particular, unexpected events must conceptually be treated as triggerable. In some cases, the agent may be implemented so that triggerability is effected by polling, but that is a low-level detail and is independent of our conceptual understanding.

Many of the agents’ events may be modeled as immediate or at least inevitable, because the agents will perform them if they wish, although they may be willing to wait. For example, when A cancels, it cannot be told that it should not. To cancel is simply its prerogative as an autonomous agent. In some cases, however, when we wish to monitor the agents more closely, we might restrict events such as cancel so they may occur only after a commitment has been created, for example, after a contractor has responded. In such a case, the event may be modeled as flexible.

Notice that changing an event from immediate to inevitable to flexible makes the coordination simpler, but restricts the autonomy of the given agent. This is a trade-off that a designer must resolve.

4.3. Inducing Relationships

The above example does not involve enough variety of relationships to exercise all of our formal language. There are no important ordering constraints among the events of different agents, except when triggering is involved. However, the following rules are easily identified—for convenience we refer below to the definitions given in Table 1.

- There is a Reply for every Solicit (R1). Replies may or may not be required in every protocol, however.
- The Replies must be enabled by the utterances to which they Reply (R3).
- Every Commit is completed (R1).
- Responds is implemented in an application-specific manner. However, input Solicits can enable associated Responds. Sometimes, we may not wish to allow this, for example, so B can ask C anyway, that is, even if there is no incoming Solicit.
- The presence of a non-Reply Respond to an utterance, for example, of utterance 2 from B_2 to C_3, indicates that the Reply is not required right away. The non-Reply Respond itself is performed unilaterally by the agent and must be modeled as immediate.

5. Enhancements

The above is the basic approach for applying Dooley graphs to coordination. However, some important enhancements are possible based on some more general ideas.

5.1. Richer Rhetorical Relationships

For expository ease, we used Parunak's proposed agent communication language in the above. Alternative languages such as KQML [21] or the FIPA language [11] might also have been used. Although more popular, these languages are no more expressive than Parunak's language and there is nothing to be gained by switching to either of them. However, Parunak's language does not cover some other kinds of communications that may be reasoned with in this approach.

For motivation, let's consider the notion of social commitment as we previously formalized [34]. The main idea is that a commitment relates a debtor and a creditor with respect to a proposition. A commitment is modeled as an abstract object and operations are defined to create or manipulate such objects. These are as follows.

O1. *Create* instantiates a commitment. *Create* usually requires a message from the debtor to the creditor.

O2. *Discharge* satisfies the given commitment. It is performed by the debtor concurrently with the actions that lead to the given condition being satisfied, for example, the delivery of promised goods or funds. For simplicity, we treat the *discharge* actions as performed only when the proposition is true. We model the *discharge* as a single message from the debtor to the creditor.

O3. *Cancel* revokes the given commitment. It can be performed by the debtor as a single message.

O4. *Release* essentially eliminates the given commitment. This is distinguished from both *discharge* and *cancel*, because *release* does not mean success or failure, although it lets the debtor off the hook. The *release* action may be performed by the creditor of the given commitment as a single message.

O5. *Delegate* shifts the role of debtor to another agent. It can be performed by the (old) debtor. This assumes a background arrangement, for example, through prior commitments, where the new debtor is somehow expected to adopt the commitment delegated to it by the old debtor. If such an arrangement does not exist, the new debtor has to explicitly acknowledge taking on the commitment.

O6. *Assign* transfers a commitment to another creditor. It can be performed by the present creditor.

Performing these operations can lead to different communications among the agents. Some of the operations can be captured by the previous set of communications. *Create* can be captured by commit, *Discharge* by ship and pay, and *Cancel* and *Release* by refuse. The last is not entirely satisfactory, because *Cancel* and *Release* are conceptually quite different. However, there are no analogs of *Delegate* and *Assign* in Parunak's language. It would not be acceptable to leave them out, because they achieve important operations on commitments, which are clearly practically applicable in domains such as supply chain management that inspired our running example.

However, rather than extend the communication language per se, we propose to extend the allowed rhetorical relationships. First, let's confirm that none of the four relationships (Response, Reply, Resolution, or Completion) proposed by Parunak can cover *Delegate* and *Assign*. Intuitively, these operations are close to being a Completion, except that they do not end the commitment. Therefore, we introduce two new types of relationships called Delegate and Assign.

An utterance i may Delegate or Assign a previous utterance j if i, respectively, *Delegate*s or *Assign*s the commitment created by j. In each case, i Completes j and creates a replacement commitment. In the case of Delegate, we assume for simplicity that if the new debtor refuses to take on the commitment, the refusal is interpreted as if the old debtor had canceled the commitment. A commitment that has been discharged, canceled, released, delegated, or assigned can no longer be subject to a new relationship; however, the commitments created in its stead may be subject to such relationships.

Lastly, we must add an optional clause in the construction of the bipartite graph introduced in Section 3. A sender-side vertex s may be related to a receiver-side vertex r if and only if the two vertices involve the same participant and one of the following five conditions hold:

- s replies to r
- r resolves s
- r is the last utterance in the conversation and r replies to s
- r completes t where t resolves s
- r completes t_0 where t_0 completes $t_1 \ldots t_{n-1}$ completes t_n resolves s.

Notice that the relationships Delegate and Assign do not explicitly arise in the bipartite graph construction, but are necessary to track the commitments, so we know what commitments are outstanding.

5.2. Potential Causality

The notion of *potential causality* was introduced to distributed computing by Lamport [22]. Potential causality is the idea that where there is an information flow across events within an agent or across events in different agents (through message passing), there may be a causal connection. The relevant events are the sending of a message, the receipt of a message, or a local computation.

Briefly, if two events take place at an agent, the first potentially causes the second. The event corresponding to the sending of a message by one agent potentially causes the event corresponding to the receipt of the same message by another agent (provided the message is not lost). Taking the transitive closure of these definitions, we can determine whether any two events in a distributed system are potentially causally related. Potential causes can be inferred just by knowing the message traffic, whereas real causes depend on the details, that is, the semantics, of the local computations. To the extent we can use potential causality instead of true causality, we need reduced input from the designer or analyzer. This would not only simplify their task, but more readily incorporate heterogeneous agents, for example, produced by different vendors, whose internal details are not known (and true causality for which cannot be determined). However, a problem is that there can be far more potential causes than real causes [26]. However, every real cause must be a potential cause. So we can be sure that potential causality will only overestimate the causes and not lose any real cause.

As a result, it is worthwhile to use potential causality unless superior information about true causality can easily be obtained. Simulated conversations and even role playing may not yield all the information about true causality easily.

EXAMPLE: Consider the discussion of agent C in Section 4.1. We took the view that C's future actions do not depend upon B's query. However, under potential causality, we would not know if B's query had an effect on C's decisions. Figure 12 shows a partial order of the messages of Table 2 indicating their potential causation of others. These are written as m_i to emphasize that they are simply low-level messages rather than utterances in this case. Notice that m_5 is captured as potentially being caused by m_3 even though the corresponding utterance was truly caused only by m_1 to which it Responded.

The characters of C would not change (in this example), because the Replies relationships are not affected by using potential Responds instead of true Responds. □

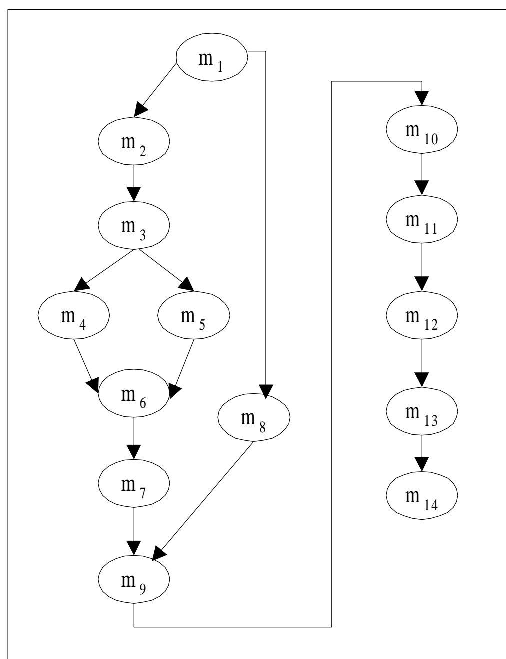

Figure 12. Potential causality illustrated

6. Discussion

The research community has been remarkably creative in discovering and formalizing high-level abstractions for the specification of multiagent systems. Despite this progress, there is a wide gap between theory and practice. All too often, the rich theories that are developed are applied in an entirely ad hoc manner. For multiagent systems to reach their true potential in complex real-life applications, the advances in theories and architectures must be complemented by advances in engineering techniques and methodologies. This is the broad challenge that we addressed in this paper.

Specifically, we showed how we can begin with Dooley graphs and with some heuristics about well-formed skeletons and symmetry across agents in the same role, come up with reasonable skeletons for coordination. We can also infer many of the desired relationships among the skeletons. The process is not automatic, but can help a human designer create a good specification. We believe essentially the same approach can be applied to other coordination approaches. A large-scale evaluation has not yet been performed, however.

We strongly believe that the nascent science of multiagent systems is inherently interdisciplinary. Accordingly, researchers in this area should be continually looking for useful ideas in other fields. We applaud Parunak for his efforts in recruiting ideas from applied linguistics. For our part, we have pursued ideas from logics of program and databases in

developing our coordination service. Here we showed how we can combine separately borrowed ideas to strengthen multiagent approaches still further.

6.1. Literature

The practical aspects of agent development have been drawing increasing attention within the agents community. In a recent paper, Wooldridge & Jennings describe the possible pitfalls of applying agents inappropriately, especially on account of the limited robustness of most existing techniques for developing agent-based applications [40]. We now review some of the most relevant literature on multiagent architectures and methodologies, especially those geared toward coordination.

Koning *et al.* develop an approach based on Petri Nets to model and validate interaction protocols among agents [20]. Like our approach, this approach too captures a behavior model for an agent as a finite state automaton. However, Koning *et al.*'s focus is on using the behavior model, not in trying to create it. In this respect, our approach and theirs are complementary. Barbuceanu and Fox describe a language, loosely based on KQML, for specifying coordination among agents [2]. Their approach involves finite state representations of the entire conversation. While their approach is quite effective in coordinating agents, it leaves open the question of how the given conversation is acquired. Our approach can help in this regard. However, in their case, the desired protocol is kept as a central model that controls the participating agents. By contrast, in our representation, the protocol is distributed across the participating agents.

Decker & Lesser [5] develop coordination algorithms for the generalized partial global planning framework. They study relationships among plans, such as whether a plan supports or interferes with another, and use those as the basis for coordination. They study several heuristics to reason about deadlines and coordination problems in various situations. They seem not to follow any specific software engineering methodology for coming up with the coordination requirements. However, there is a richness in Decker & Lesser's representations that lies beyond the present approach—extensions to accommodate the full power of their approach would be interesting.

Sichman *et al.* consider social dependencies among different agents and use them to guide the agents' interactions with each other [29]. No specific approach for uncovering the dependencies is provided. It appears that the social dependencies are at a higher level than some of the coordination we studied here. Conceivably, we could come with a generic skeleton to capture agents who reason socially and to use it coordinate the agents so that their dependencies can be detected and managed.

The agent-oriented approaches to programming involve the formalization of constructs such as beliefs, intentions, and commitments [28, 35]. Although this class of work has been mostly theoretical, some practical variants have also been developed. Haddadi describes the COSY architecture initiated by Burmeister and Sundermeyer [13, chap. 5]. This architecture involves the specification and execution of various communication protocols, for example, for requests and proposals, through which agents may cooperate. Agentis is an alternative agent-oriented framework for building interactive multiagent systems [6]. Agentis works through a small set of protocols, for example, registration of agents and service requests, that are used as the basis for other interactions. These protocols are rep-

resented as pairs of finite state automata—one for each role in the protocol. d'Inverno *et al.* take these machines as given and prove some useful properties about them. Smith *et al.* apply the theory of joint intentions to analyze conversation policies expressed as finite state diagrams [36]. In this way, the COSY, Agentis, and joint intentions efforts complement the present approach, which seeks to come up with the representations that they assume as given.

Brazier *et al.* apply the DESIRE framework to model multiagent systems [3]. DESIRE is a formal approach related to conventional software engineering that is especially strong in terms of modeling hierarchies of objects and components. The behavior models in DESIRE are similar to statecharts [15]. This approach too is not concerned with constructing behavior models for agents, as we are.

Our emphasis has been on explicitly designed interactions, which are most appropriate when there is a relatively small number of roles in the system, and the roles are expected to be responsible for sizable chunks of activity. Drogoul & Collinot present an alternative approach, called Cassiopeia, in which the agents are endowed with capabilities to detect relationships and to form and dissolve temporary organizations in order to coordinate their actions for a specific task [8]. The designer imparts additional knowledge so the agents can choose the relationships that are the most relevant in the given domain. Agents' behaviors in Cassiopeia correspond to subskeletons in our approach; influences among behaviors correspond to coordination relationships. However, Cassiopeia is a more bottom-up approach and opens the possibility of learning mechanisms for coordination, which also appear to be a promising line of research.

6.2. Future Directions

There are a number of topics for future investigation. One is the consideration of conversations that are effectively nonterminating. Conversations that are specifiable as finite state machines can be accommodated in our approach with some extensions. Repeated interactions among the agents can help identify more of the branches of the possible conversations, but care must be taken so that unnecessary causal connections are not inferred. Another challenge is to use negative examples, that is, graphs that describe failed conversations or conversations that do not meet some desired criteria. These tasks could be facilitated by a tool that incorporates some machine learning over conversations.

The above approach considers specific conversation instances to determine the coordination requirements for a multiagent system. A more ambitious challenge is to induce the coordination requirements for entire classes of interactions, for example, to capture interesting aggregate properties of a system. Specifically, it would be helpful to develop multiagent systems with more flexible control on the extent of cooperation or negotiation that their member agents should support.

Although we considered only the coordination requirements, there is need to support additional types of interaction. Interactions in general can involve more subtle patterns than are captured by our coordination relationships. Within the framework of interaction-oriented programming, we have taken some initial steps towards formalizing these patterns [33, 19, 38]. A major direction will be to develop methodologies corresponding to these

additional abstractions that lead to more flexible and powerful multiagent systems while requiring only a small additional effort by the designer.

References

1. John L. Austin. *How to Do Things with Words*. Clarendon Press, Oxford, 1962.
2. Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multi agent systems. In *Proceedings of the International Conference on Multiagent Systems*, pages 17–24, 1995.
3. Frances M. T. Brazier, Barbara M. Dunin-Kęplicz, Nick Jennings, and Jan Treur. Desire: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6(1):67–94, 1997.
4. Brahim Chaib-draa. Industrial applications of distributed artificial intelligence. In *[18]*, pages 31–35. 1998. (Reprinted from *Communications of the ACM, 1995*).
5. Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. In *[18]*, pages 450–457. 1998. (Reprinted from *Proceedings of the International Conference on Multiagent Systems, 1995*).
6. Mark d'Inverno, David Kinny, and Michael Luck. Interaction protocols in Agentis. In *Proceedings of the 3rd International Conference on Multiagent Systems (ICMAS)*, pages 112–119. IEEE Computer Society Press, July 1998.
7. R. A. Dooley. Appendix B: Repartee as a graph. In *[23]*, pages 348–358. 1976.
8. Alexis Drogoul and Anne Collinot. Applying an agent-oriented methodology to the design of artificial organizations: A case study in robotic soccer. *Autonomous Agents and Multi-Agent Systems*, 1(1):113–129, June 1998.
9. Edmund H. Durfee. Distributed problem solving and planning. In *[39]*, chapter 3, pages 121–164. 1999.
10. E. Allen Emerson and Edmund C. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
11. Foundation for intelligent physical agents (FIPA) specification, 1998. http://www.fipa.org.
12. Martin Fowler. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, 1997.
13. Afsaneh Haddadi. *Communication and Cooperation in Agent Systems: A Pragmatic Theory*. Springer-Verlag, Heidelberg, 1996.
14. Martin Hardwick and Richard Bolton. The industrial virtual enterprise. *Communications of the ACM*, 40(9):59–60, September 1997.
15. David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.
16. David Harel and Anmon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
17. Michael N. Huhns and Munindar P. Singh. Agents and multiagent systems: Themes, approaches, and challenges. In *[18]*, chapter 1, pages 1–23. 1998.
18. Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1998.
19. Anuj K. Jain, Manuel Aparicio IV, and Munindar P. Singh. Agents for process coherence in virtual enterprises. *Communications of the ACM*, 42(3):62–69, March 1999.
20. Jean-Luc Koning, Guillaume Francois, and Yves Demazeau. Formalization and pre-validation for interaction protocols in multiagent systems. In *Proceedings of the European Conference on Artificial Intelligence*, pages 298–302. John Wiley, 1998.
21. Yannis Labrou and Tim Finin. Semantics and conversations for an agent communication language. In *[18]*, pages 235–242. 1998. (Reprinted from *Proceedings of the International Joint Conference on Artificial Intelligence, 1997*).
22. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
23. Robert E. Longacre. *An Anatomy of Speech Notions*. Peter de Ridder, Lisse, Holland, 1976.
24. H. Van Dyke Parunak. Visualizing agent conversations: Using enhanced Dooley graphs for agent design and analysis. In *Proceedings of the 2nd International Conference on Multiagent Systems*, pages 275–282. AAAI Press, 1996.

25. Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, New York, 4th edition, 1997.
26. Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.
27. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Upper Saddle River, NJ, 1996.
28. Yoav Shoham. Agent-oriented programming. In *[18]*, pages 329–349. 1998. (Reprinted from *Artificial Intelligence, 1993*).
29. Jaime Simão Sichman, Rosaria Conte, Yves Demazeau, and Cristiano Castelfranchi. A social reasoning mechanism based on dependence networks. In *[18]*, pages 416–420. 1998. (Reprinted from *Proceedings of the 11th European Conference on Artificial Intelligence, 1994*).
30. 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. Springer-Verlag, May 1997.
31. Munindar P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, December 1998.
32. Munindar P. Singh. A customizable coordination service for autonomous agents. In *Intelligent Agents IV: Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, pages 93–106. Springer-Verlag, 1998.
33. Munindar P. Singh. The intentions of teams: Team structure, endodeixis, and exodeixis. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI)*, pages 303–307. John Wiley, August 1998.
34. Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.
35. Munindar P. Singh, Anand S. Rao, and Michael P. Georgeff. Formal methods in DAI: Logic-based representation and reasoning. In *[39]*, chapter 8, pages 331–376. 1999.
36. Ira A. Smith, Philip R. Cohen, Jeffrey M. Bradshaw, Mark Greaves, and Heather Holmback. Designing conversation policies using joint intention theory. In *Proceedings of the 3rd International Conference on Multiagent Systems (ICMAS)*, pages 269–276. IEEE Computer Society Press, July 1998.
37. Putnam P. Texel and Charles B. Williams. *Use Cases Combined with Booch, OMT, UML: Process and Products.* Prentice-Hall, Upper Saddle River, NJ, 1997.
38. Mahadevan Venkatraman and Munindar P. Singh. Verifying compliance with commitment protocols: Enabling open web-based multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, September 1999.
39. Gerhard Weiß, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999.
40. Michael J. Wooldridge and Nicholas R. Jennings. Software engineering with agents: Pitfalls and pratfalls. *IEEE Internet Computing*, 3(3):20–27, May 1999.

Appendix

Formal Syntax and Semantics

This appendix is only included for completeness, and may safely be skipped on a first reading.

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. 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 $\mathcal{I}$.

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

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

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

Elsewhere [32], we present a set of equations that enable symbolic reasoning on $\mathcal{I}$ to determine when a certain event may be permitted, prevented, or triggered.

Acknowledgments

This work is supported by the NCSU College of Engineering, the National Science Foundation under grants IIS-9529179 and IIS-9624425, and IBM corporation. I am indebted to the anonymous reviewers for their helpful comments.