Debugging Agent Interactions: a Case Study

David Flater
National Institute of Standards and Technology
100 Bureau Drive, Stop 8260
Gaithersburg, MD 20899-8260
U.S.A.
dflater@nist.gov

ABSTRACT
The Contract Net protocol is a general-purpose protocol for distributed problem solving. Many modern agent infrastructures facilitate the generation of agents supporting Contract Net. We used one such infrastructure to simulate a Contract Net-based approach to job scheduling and found that some jobs failed to get scheduled even though the resources were available. This paper describes two phases of the subsequent debugging effort. The first phase was enhancing the visualization of the agent community to reveal the causes of failed negotiations. The second phase was formalizing the problem using a Temporal Calculus of Communicating Systems (TCCS) and attempting to find a solution. After exploring a number of solutions that would not generalize, we found that switching from one-stage to two-stage commitment sufficed. For coordination problems in general, our case study demonstrates the applicability of rigorous methods and the importance of providing run-time visibility into agents' logic.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed debugging*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms
Experimentation, Verification.

Keywords
Agents, coordination, negotiation, scheduling.

1. INTRODUCTION
For some time, the manufacturing sector has maintained interest in agent-based approaches to supply chain management, planning, scheduling, and control. The difficulty of coordinating the flow of information through the many domains of responsibility within or among manufacturing enterprises often makes autonomous agents seem an attractive way to simplify the problem. However, systems composed of interacting agents are notoriously difficult to test and debug. It is hard enough to achieve sufficient visibility into agents' interactions to be able to determine whether individual agents are behaving as specified. It is harder yet to know where to begin when the collective behavior of a group of apparently sane agents was not as expected.

Many modern agent infrastructures facilitate the generation of agents supporting the general-purpose Contract Net protocol [1] for distributed problem solving. We used one such package, Zeus[*] [7] version 1.02, in a case study of debugging agent interactions. In the following sections, we describe our test scenario, the testability enhancements that helped us understand its behavior, and the analysis that revealed the true depth of the problem we faced. We conclude with a summary of those observations of relevance to agent standards and coordination in general.

2. TEST SCENARIO
We came upon our test scenario honestly when a learning exercise to build an agent-based simulation of job scheduling went awry. The scenario contained a merged user interface and supervisory agent called the Guardian. To achieve the user's goals, the Guardian had to arrange for jobs to be done by a pool of three workcells, aptly named Good, Cheap and Fast (see Figure 1). By refactoring the standard shop floor scheduling problem as procurement of labor as a commodity instead of centralized choreography of completely subordinated workcells, we opened the door to a style of virtual manufacturing wherein the means of production would be rented as needed by transient manufacturing enterprises. But it is not really necessary to motivate the particular scenario, since the encountered problem and the techniques used to analyze it are scenario-independent.

The Contract Net negotiation in our scenario involved only five different message types: Call For Proposals (CFP), Refuse, Propose, Accept-Proposal, and Reject-Proposal. The Guardian sent a CFP message to each workcell to initiate negotiations. Each workcell responded with either a Propose message if it was willing and able to do the work at some price, or a Refuse message if it was unwilling or unable to do the work. The

[*] Commercial equipment and materials are identified in order to describe certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

Guardian collected proposals and then chose one that was cheapest. The "winner" received an Accept-Proposal message; any other proposers received Reject-Proposal messages. If no proposals were received, the work did not get done.

In Zeus, after identifying the Guardian as a Contract Net manager and the workcells as Contract Net contractors, it was only necessary to set the parameters of the workcells' tasks to be able to generate the entire simulation. We used the following parameters in our experiment: Workcell Good took two time slices and charged 1000 units of currency to supply the labor for one job; Cheap took two time slices and charged 500; Fast took one time slice and charged 1000.

Having configured three workcells all capable of performing the same jobs, we innocently requested the Guardian to accomplish three jobs at the same time. For no particular reason, we expected one job to go to each workcell. It would have been equivalent for two jobs to go to Fast and one to go to Cheap. But, to our shock and horror, the simulation routinely gave one job to Fast, one job to Cheap, and one job to no one at all.

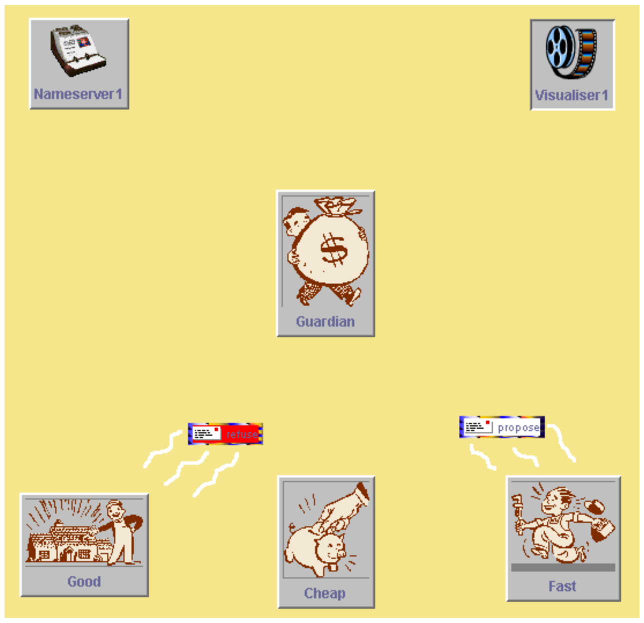
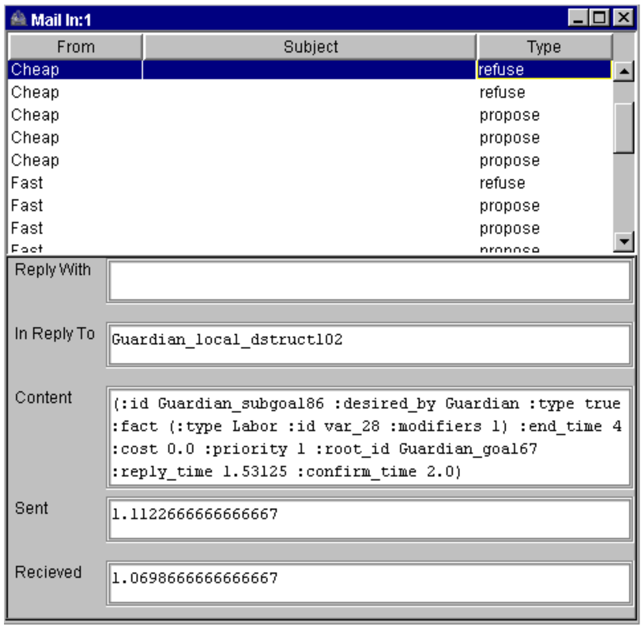


Figure 1. Society view with messages in transit

3. DEBUGGING

Zeus includes a set of tools for monitoring and analyzing agent behavior. At the forefront, the agent society viewer (Figure 1) provides an animated view of the types of messages passing between agents at run-time. We edited the Zeus source code that colorizes messages so that Reject-Proposal and Refuse messages would be easier to see. Confusingly, we saw two rejections (one to Good, one to Fast) and seven refusals (three from Good, two from Cheap, two from Fast). At that point it was not obvious to us why Good rejected all three jobs.

The Zeus agent viewer permits viewing of the incoming and outgoing messages for each agent. We again edited source code to expand the size of the mailbox buffers so that all messages sent during the run could be recalled. Looking at the messages (see Figure 2), we saw that refusals arrived with various attributes, but with no obvious way to determine the reason for the refusal.


Mail In:1

From	Subject	Type
Cheap		refuse
Cheap		refuse
Cheap		propose
Cheap		propose
Cheap		propose
Fast		refuse
Fast		propose
Fast		propose
Fast		propose

Reply With

In Reply To: Guardian_local_dstruct102

Content:
```
(:id Guardian_subgoal86 :desired_by Guardian :type true
:fact (:type Labor :id var_28 :modifiers 1) :end_time 4
:cost 0.0 :priority 1 :root_id Guardian_goal67
:reply_time 1.53125 :confirm_time 2.0)
```

Sent: 1.1122666666666667

Recieved: 1.0698666666666667

Figure 2. Refusal in inbox

We tried different values for the Guardian's budget and the workcells' costs, to no effect. We then tried different confirm and end times for the requested jobs. The confirm time is the deadline for the awarding of the contract, while the end time is the deadline for completion of the contracted work. The resulting behaviors (see Table 1) showed that there were more potential problems, but were not sufficient to diagnose the problem at hand.

Table 1. Effects of changing confirm time and end time

Confirm time	End time	Behavior
Now+2	Now+5	Good idle, Fast one job, Cheap one job. *Rarely*, Good idle, Fast two jobs, Cheap one job.
-1 ("Don't care")	Now+5	Would not run.
Now+5	Now+5	Would not run.
Now+4	Now+5	All workcells idle.
Now+3	Now+5	Good idle, Fast one job, Cheap idle.
Now+3	Now+6	Indistinguishable from +2/+5.
Now+2	Now+6	Good idle, Fast two jobs, Cheap one job.

Despite its regular success, the last variant (+2/+6) was unremarkable because it represented a scenario with substantially less schedule pressure.

We tried running the scenario without Fast. On the first try, Good and Cheap were each awarded one of the jobs; the third job was not feasible. On the second try, Cheap was awarded one job while the other two failed.

We changed the Guardian's negotiation strategy from Default-No-Negotiation to GrowthFunction. This increased the amount of negotiation, but then all three jobs failed. When we made the analogous change to the Workcell strategies, the behavior was as it was before but with more negotiation.

We turned on verbose debugging, but gained no information from it.

We finally examined the source code implementing the various states of negotiation. The coordination engine view (see Figure 3) shows the state graph of the negotiation, with failures in red (in monochrome copy, dark gray). In most states, there are multiple places where some condition will cause them to return a failure. When this occurs, we see the failure but not its cause.

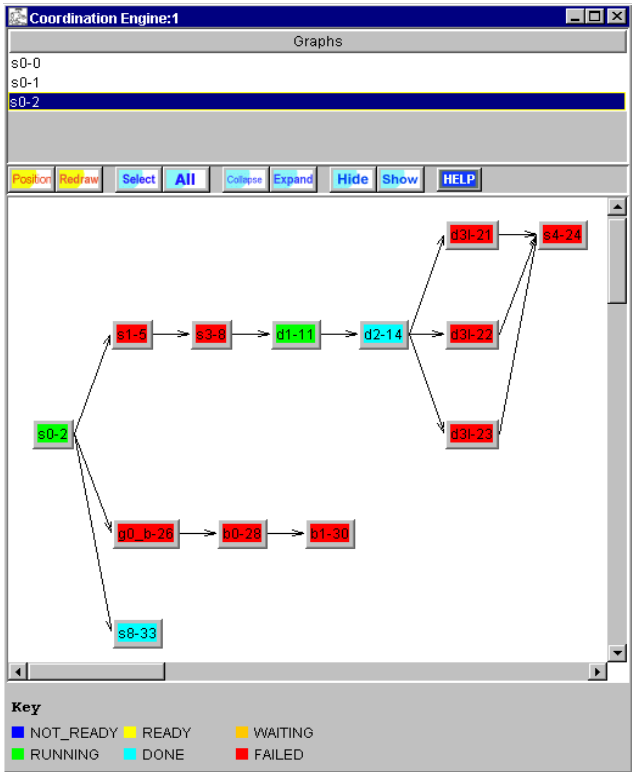


Figure 3. Guardian view of failed job

4. TESTABILITY ENHANCEMENTS

The Foundation for Intelligent Physical Agents (FIPA) specifies that reject and refuse messages should contain a reason field in the content. Slightly paraphrased, the specification reads: The agent receiving a refuse act is entitled to believe that the (causal) reason for the refusal is represented by the third term of the tuple, which may be the constant true. [4]

We modified Zeus 1.02 to add a reason field to the content and present this information in the coordination engine view. We also attached reason codes to local failures that do not generate messages between agents. Subsequently, the Guardian's view of the job that failed (see Figure 4) showed that it failed because all three workcells refused it as infeasible. Good and Fast's views of the first job (see Figure 5) showed that they bid on the contract but did not win – the first job was awarded to Cheap. Fast was awarded the second job because both Good and Cheap refused it, as well as the third job, as being infeasible (see Figure 6). Fast refused only the third job.

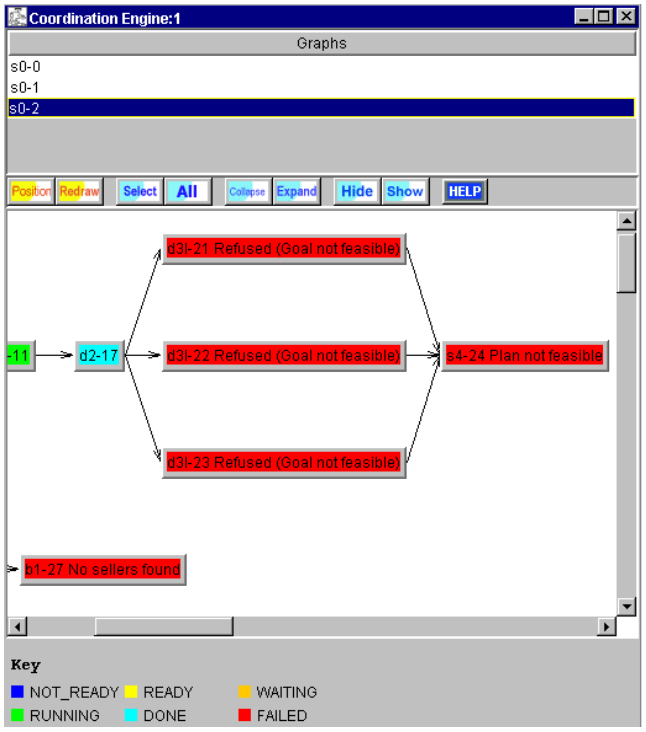


Figure 4. Guardian view with reasons added

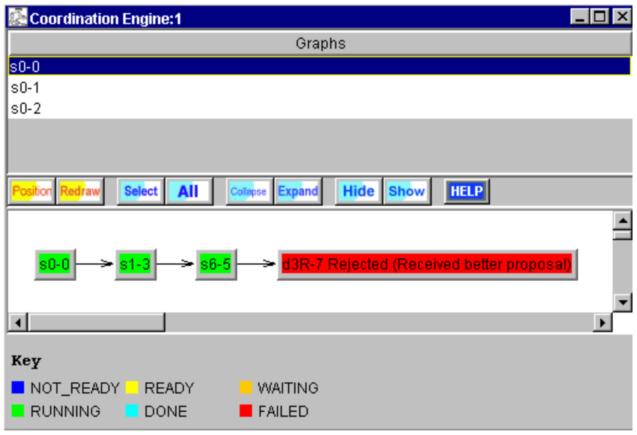


Figure 5. Workcell view of rejection

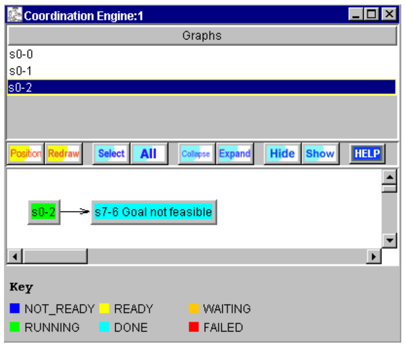


Figure 6. Workcell view of refusal

The behavior of the agents was then more transparent, and we could guess what was going on:

1. Job 1 arrived first, was proposed on by all three workcells and awarded to Cheap. The other two workcells responded to the Guardian's Reject-Proposal messages with Refuse messages, obfuscating the fact that it was the Guardian who called off negotiation. After the modification, it was obvious from the reason embedded in the refusal: "Rejected (Received better proposal)."

2. Job 2 arrived second and was refused by Good and Cheap because they already had Job 1 tentatively on their schedules. Fast proposed on it because it theoretically had time to do both Job 1 and Job 2, and was awarded the job.

3. Job 3 arrived last and was immediately refused by all three workcells as not feasible.

Even though we ran it as multiple processes on the same computer, the distributed nature of the simulation caused the ordering of messages to be semi-random. In rare instances, these random perturbations led Fast to bid on jobs 2 and 3 instead of 1 and 2, in which case Fast could be awarded two jobs.

The root of the problem, then, was that tentatively scheduled jobs were sufficient to cause the immediate refusal of other jobs that would require the same time slot, even though the subsequent rejection of the tentative job would make the refused job feasible. But we could not simply change the workcells to allow overbooking of tentative jobs, because we would then accept contracts that were not feasible.

5. FORMALIZATION OF THE PROBLEM

We now model a simplified version of our scenario using a Temporal Calculus of Communicating Systems (TCCS) [5][6] (one of several that exist by the same acronym). This model is then fed into the Edinburgh Concurrency Workbench [2] (CWB) version 7.1 for analysis.

For a minimal introduction to TCCS notation, please refer to the Appendix of this paper.

We originally modeled the complete scenario with three workcells and Guardian. The resulting model exceeded 100K states before we had even completed it and could not be analyzed on a workstation having a full gigabyte of memory. The formulas and machine-readable TCCS for the original model are available on request. Here we present a simplified model where the workcell Good has been removed. The remaining two workcells are sufficient to demonstrate a range of system behaviors.

The other simplifications are as follows:

1. We revert to the Default-No-Negotiation and Default-Fixed-Margin strategies for the Guardian and workcells respectively, which reduces the complexity of the interactions but does not change the nature of the "failure" or the underlying "fault."

2. We treat negotiation messages as if they were instantaneous, whereas workcell labor takes time.

3. We pretend that work on accepted proposals begins immediately and must be completed by $t = 2$.

4. We do not model the unnecessary refuse messages that are sent by workcells in response to reject-proposal messages.

5. We do not model the many alternative modes of failure, such as agents failing to respond.

6. We do not allow for indefinite future scheduling of jobs, but only deal with the scheduling of the three jobs from our scenario.

Let C, F, and N refer to the agents Cheap, Fast, and Guardian respectively.

Actions:

– Call for proposals to workcell n for job j: $cfp_{n,j}, n \in \{C, F\}, 1 \le j \le 3$

– Workcell n proposes to do job j: $propose_{n,j}$

– Workcell n refuses to do job j: $refuse_{n,j}$

– Guardian accepts proposal for workcell n to do job j at the proposed price: $accept_{n,j}$

– Guardian rejects proposal for workcell n to do job j at the proposed price: $reject_{n,j}$

5.1 Guardian

Let CFP_j be the Guardian subprocess trying to obtain labor for job j.

$$N \overset{def}{=} \prod_{1<=j<=3} CFP_j$$

$$CFP_j \overset{def}{=} cfp_{C,j}.\delta.0 \mid cfp_{F,j}.\delta.0 \mid Feedback_j$$

Each job produces two CFP messages. Each of the two workcells can send either a proposal or a refusal in response to a CFP, giving four distinct cases. These are multiplied by two permutations of the CFP responses. The actual order is unimportant, but failing to accept all permutations leads to deadlocks in the TCCS. Although we have not parameterized the messages with the proposed price of the labor, the Guardian's preference for cheaper labor is captured in the decision tree below based on the known properties of the workcells.

$$Feedback_j \overset{def}{=} \left(\begin{array}{l} \overline{propose}_{C,j}.\left(\begin{array}{l} \overline{propose}_{F,j}.obsCF_j.(accept_{C,j}.\delta.0 \mid reject_{F,j}.\delta.0) \\ + \overline{refuse}_{F,j}.obsC_j.accept_{C,j}.\delta.0 \end{array} \right) \\ + \overline{refuse}_{C,j}.\left(\begin{array}{l} \overline{propose}_{F,j}.obsF_j.accept_{F,j}.\delta.0 \\ + \overline{refuse}_{F,j}.obsNil_j.\delta.0 \end{array} \right) \\ + \overline{propose}_{F,j}.\left(\begin{array}{l} \overline{propose}_{C,j}.obsCF_j.(accept_{C,j}.\delta.0 \mid reject_{F,j}.\delta.0) \\ + \overline{refuse}_{C,j}.obsF_j.accept_{F,j}.\delta.0 \end{array} \right) \\ + \overline{refuse}_{F,j}.\left(\begin{array}{l} \overline{propose}_{C,j}.obsC_j.accept_{C,j}.\delta.0 \\ + \overline{refuse}_{C,j}.obsNil_j.\delta.0 \end{array} \right) \end{array} \right)$$

The actions of the form *obs…* are included for the technical reason that system traces from the CWB show communications between agents as opaque *tau* (internal) actions of the system. We require observable actions to be able to interpret the traces that lead to any given state.

5.2 Cheap

$$C \overset{def}{=} Free_C$$

(Had we a Good workcell, it would be defined as $Free_G$.)

$$Free_n \overset{def}{=} \delta.\begin{pmatrix} \overline{cfp}_{n,1}.propose_{n,1}.Tentative_{n,1} \\ +\overline{cfp}_{n,2}.propose_{n,2}.Tentative_{n,2} \\ +\overline{cfp}_{n,3}.propose_{n,3}.Tentative_{n,3} \end{pmatrix}$$

The scheduler policy at the root of our problem is codified in *Tentative:*

$$Tentative_{n,1} \overset{def}{=} \begin{pmatrix} \overline{accept}_{n,1}.Confirmed_{n,1} \\ +\overline{reject}_{n,1}.Free_n \\ +\overline{cfp}_{n,2}.refuse_{n,2}.Tentative_{n,1} \\ +\overline{cfp}_{n,3}.refuse_{n,3}.Tentative_{n,1} \end{pmatrix}$$

Finally, in *Confirmed*, we simulate a task requiring two time slices. The following is sufficient for our purposes, but does not allow for scheduling of additional future jobs while the task is running.

$$Confirmed_{n,1} \overset{def}{=} (2).Free_n \oplus \begin{pmatrix} \overline{cfp}_{n,2}.refuse_{n,2}.Confirmed_{n,1} \\ +\overline{cfp}_{n,3}.refuse_{n,3}.Confirmed_{n,1} \end{pmatrix}$$

Analogously for $Tentative_{n,2}$, $Tentative_{n,3}$, $Confirmed_{n,2}$, $Confirmed_{n,3}$.

Although it is feasible to reduce the number of processes by merging the three $Tentative_{n,j}$ into a single $Tentative_n$, and similarly for $Confirmed_n$, we keep them separated here to clarify the intended behaviors of the system.

5.3 Fast

$$F \overset{def}{=} Fr$$

$$Fr \overset{def}{=} \delta.\begin{pmatrix} \overline{cfp}_{F,1}.propose_{F,1}.Te_1 \\ +\overline{cfp}_{F,2}.propose_{F,2}.Te_2 \\ +\overline{cfp}_{F,3}.propose_{F,3}.Te_3 \end{pmatrix}$$

$$Te_1 \overset{def}{=} \begin{pmatrix} \overline{cfp}_{F,2}.propose_{F,2}.Te_{12} \\ +\overline{cfp}_{F,3}.propose_{F,3}.Te_{13} \\ +\overline{accept}_{F,1}.Co_1 \\ +\overline{reject}_{F,1}.Fr \end{pmatrix}$$

Analogously for Te_2, Te_3.

$$Te_{12} \overset{def}{=} \begin{pmatrix} \overline{cfp}_{F,3}.refuse_{F,3}.Te_{12} \\ +\overline{accept}_{F,1}.Te_2Co_1 \\ +\overline{reject}_{F,1}.Te_2 \\ +\overline{accept}_{F,2}.Te_1Co_2 \\ +\overline{reject}_{F,2}.Te_1 \end{pmatrix}$$

Analogously for Te_{13}, Te_{23}.

$$Te_1Co_2 \overset{def}{=} \begin{pmatrix} \overline{cfp}_{F,3}.refuse_{F,3}.Te_1Co_2 \\ +\overline{accept}_{F,1}.Co_{12} \\ +\overline{reject}_{F,1}.Co_2 \end{pmatrix}$$

Analogously for Te_1Co_3, Te_2Co_1, Te_2Co_3, Te_3Co_1, Te_3Co_2.

$$Co_1 \overset{def}{=} (1).Fr \oplus \begin{pmatrix} \overline{cfp}_{F,2}.propose_{F,2}.Te_2Co_1 \\ +\overline{cfp}_{F,3}.propose_{F,3}.Te_3Co_1 \end{pmatrix}$$

Analogously for Co_2, Co_3.

$$Co_{12} \overset{def}{=} (2).Fr \oplus \overline{cfp}_{F,3}.refuse_{F,3}.Co_{12}$$

Analogously for Co_{13}, Co_{23}.

5.4 Analysis

To make the analysis terminate, we include an additional process $T \overset{def}{=} (4).0$ that halts the system after four time slices. The system can now be composed as

$$System \overset{def}{=} \begin{pmatrix} (C \mid F \mid N \mid T)\backslash \\ \{\ldots\ list\ of\ unobservable\ actions\ elided\ \ldots\} \end{pmatrix}$$

Loading the model into the CWB, we find that the system has approximately 2732 states.[*] There are four distinct "deadlock" states, all of which represent intentional haltings of the system at $t = 4$. Using the CWB function to list all observations of length three, we find 66 of them. Modulo the various combinatorics, we only have three distinct system behaviors:

1. C bids on one job, F bids on the other two. There are 18 ways this can happen.
2. F competes unsuccessfully with C for one job and wins another, while the third job is refused by both workcells. There are 36 ways this can happen.
3. F and C collide as in case 2, but F is rejected *before* the CFP for the third job arrives, so F bids on the last job and gets it. There are 12 ways this can happen.

In practice, the order in which CFP messages arrive at workcells is not uniformly random, and it is highly unlikely for one negotiation to run to completion while the CFP for another languishes *en route*, so there is a strong bias in favor of case 2. But it is interesting to note that if any sequence were as likely as any other, one job would still fall through more often than not. This would probably not be true of our original, three-workcell scenario, where there are more ways for all three jobs to get done.

6. EVALUATION OF PROSPECTIVE SOLUTIONS

In a free market, both Guardian and workcells would suffer when jobs fall through unnecessarily. Neither side has a motive to leave this problem unfixed.

[*] Quoting from the Edinburgh Concurrency Workbench user manual (Version 7.1) dated 1999-07-18: "The number of states of an agent is not as clear a concept as you might think: treat the number as a rough indication of size, only."

6.1 Workcell-Serialized Negotiations

One approach that seems completely wholesome and general at first glance is for the workcells to delay responding to CFPs while their schedules are tentatively full. We can model this in the TCCS by failing to accept CFP messages while in the relevant states:

$$Tentative_{n,1} \overset{def}{=} \begin{pmatrix} \overline{accept}_{n,1}.Confirmed_{n,1} \\ +\overline{reject}_{n,1}.Free_n \end{pmatrix}$$

$$Te_{no} \overset{def}{=} \begin{pmatrix} \overline{accept}_{F,n}.Te_oCo_n \\ +\overline{reject}_{F,n}.Te_o \\ +\overline{accept}_{F,o}.Te_nCo_o \\ +\overline{reject}_{F,o}.Te_n \end{pmatrix}$$

$$Te_nCo_o \overset{def}{=} \overline{accept}_{F,n}.Co_{no} + \overline{reject}_{F,n}.Co_o$$

The resulting system, unfortunately, contains deadlocks. For example:

$$C \Rightarrow Tentative_{C,1}$$
$$F \Rightarrow Te_{23}$$

$$N \Rightarrow \begin{pmatrix} \left(\delta.0 \mid cfp_{F,1}.\delta.0 \mid \begin{pmatrix} \overline{propose}_{F,1}.obsCF_1.(accept_{C,1}.\delta.0 \mid reject_{F,1}.\delta.0) \\ +\overline{refuse}_{F,1}.obsC_1.accept_{C,1}.\delta.0 \end{pmatrix}\right) \\ \left|\left(cfp_{C,2}.\delta.0 \mid \delta.0 \mid \begin{pmatrix} \overline{propose}_{C,2}.obsCF_2.(accept_{C,2}.\delta.0 \mid reject_{F,2}.\delta.0) \\ +\overline{refuse}_{C,2}.obsF_2.accept_{F,2}.\delta.0 \end{pmatrix}\right)\right. \\ \left|\left(cfp_{C,3}.\delta.0 \mid \delta.0 \mid \begin{pmatrix} \overline{propose}_{C,3}.obsCF_3.(accept_{C,3}.\delta.0 \mid reject_{F,3}.\delta.0) \\ +\overline{refuse}_{C,3}.obsF_3.accept_{F,3}.\delta.0 \end{pmatrix}\right)\right. \end{pmatrix}$$

In the general case, this problem would be a show-stopper. In our particular scenario, if we presume that the Guardian has prior knowledge of the price differential between Cheap and Fast and is only inquiring to see if their schedules are clear, we could work around by accepting the proposal from Cheap before Fast even responds:

$$Feedback_j \overset{def}{=} \begin{pmatrix} \overline{propose}_{C,j}.accept_{C,j}.\begin{pmatrix} \overline{propose}_{F,j}.obsCF_j.reject_{F,j}.\delta.0 \\ +\overline{refuse}_{F,j}.obsC_j.\delta.0 \end{pmatrix} \\ +\{\ldots other\ permutations\ are\ unchanged\ \ldots\} \end{pmatrix}$$

The resulting System has (approximately) 1685 states and 18 distinct observations, all of which get all three jobs done.

Even with the stopgap solution, this approach contains an inherent tradeoff in that it delays responses to the later CFPs. Although we have not explicitly modeled the time consumed in negotiations, it is clear that the delay will get worse as additional CFPs pile up. Failure to respond to a CFP is equivalent to a refusal, so in the simple case nothing would be lost, but at some point the backlog of "bad" CFPs would begin to impact the execution of later "good" ones. Moreover, the delays could have unacceptable social consequences in practice, particularly if the Guardian also fails to accept or reject proposals in a timely fashion. Our scenario is too limited to permit analysis of these behaviors.

6.2 Guardian-Serialized Negotiations

In scenarios having only a single Contract Net manager, it would suffice to issue the CFPs one at a time, delaying the next until negotiation on the previous has completed. This is accomplished in our model by letting $N \overset{def}{=} CFP_1$, revising *Feedback*$_1$ and *Feedback*$_2$ to get rid of the parallel operators, and replacing the eight nontemporal deadlocks $(\delta.0)$ remaining in each with *CFP*$_{j+1}$. The resulting system has a mere 36 states and only one observable behavior – both workcells bid on the first job, which goes to Cheap; the second and third jobs are only bid on by Fast.

Of course, this introduces more of the unmodeled negotiation delay that we discussed above. A more scenario-specific solution is to alter the Guardian to issue the CFPs to Fast only after they have been refused by Cheap:

$$CFP_j \overset{def}{=} cfp_{C,j}.\delta.0 \mid Feedback_j$$

$$Feedback_j \overset{def}{=} \begin{pmatrix} \overline{propose}_{C,j}.obsC_j.accept_{C,j}.\delta.0 \\ +\overline{refuse}_{C,j}.cfp_{F,j}.\begin{pmatrix} \overline{propose}_{F,j}.obsF_j.accept_{F,j}.\delta.0 \\ +\overline{refuse}_{F,j}.obsNil_j.\delta.0 \end{pmatrix} \end{pmatrix}$$

This is merely an escalation of the "pre-selection" of Cheap that we made previously, where proposals from Cheap were accepted before Fast had a chance to respond, but this one requires no modifications to the workcells' behavior. The resulting system has 431 states and 18 distinct observations, all of which award all three jobs without inter-workcell competition.

6.3 Insistent Guardian

A simple strategy to implement is to have the Guardian try again if a job fails to attract a proposal the first time around. We can attempt this in our modeled scenario by replacing the nontemporal deadlock following each *obsNil*$_j$ in *Feedback*$_j$ with *CFP*$_j$, effectively looping back immediately as soon as a CFP fails. Unfortunately, this creates a cycle that is not guaranteed to terminate. The next best thing is to wait one time slice before trying again: (1).*CFP*$_j$. In order to handle these CFPs, we must extend our *Confirmed* states slightly. We know that Cheap would be incapable of finishing a job by $t = 2$ if its CFP arrived at $t = 1$:

$$Confirmed_{n,1} \overset{def}{=} (1).Confirmed'_{n,1} \oplus \begin{pmatrix} \overline{cfp}_{n,2}.refuse_{n,2}.Confirmed_{n,1} \\ +\overline{cfp}_{n,3}.refuse_{n,3}.Confirmed_{n,1} \end{pmatrix}$$

$$Confirmed'_{n,1} \overset{def}{=} (1).Free_n \oplus \begin{pmatrix} \overline{cfp}_{n,2}.refuse_{n,2}.Confirmed'_{n,1} \\ +\overline{cfp}_{n,3}.refuse_{n,3}.Confirmed'_{n,1} \end{pmatrix}$$

The *Co*$_j$ processes already permit Fast to accept CFPs at $t = 1$. It is not necessary to extend the *Co*$_{jk}$ states for this scenario; entering such states implies that all three jobs will have been allocated.

The resulting system has 2812 states and 66 distinct observations that break down in the same proportions as in the original model, with case 2 revised as follows:

2. F competes unsuccessfully with C for one job and wins another, while the third job is refused by both workcells. Time passes, and then F accepts the third job on the second try. There are 36 ways this can happen.

In practice, the number of retries would need to be constrained in various ways, as retries of proposals having no chance of success tie up valuable resources. More significantly, this approach, while generally applicable, does not help us in all cases. It is only by virtue of the fact that Fast can begin a job one cycle late and still finish on time that we achieve acceptable results. In scenarios where all workcells must begin work at $t = 0$ to achieve acceptable results, trying again later is of no help.

6.4 Better Upward Communication

The workcells could inform the Guardian that two jobs under consideration are mutually exclusive, requiring the same resources at the same time, and force the Guardian to make a choice. In our scenario, this would degenerate to centralized scheduling with many superfluous interactions. Obviously, if there were multiple requestors with conflicting needs, the decision could not be passed upwards in this way.

6.5 Two-Stage Commitment

"Two-stage commitment" as described below should not be confused with the two-phase commit protocol used in databases.

Workcells are no longer obliged to a contract when they send a proposal, and the Guardian is no longer assured of getting a job done by sending an acceptance. Upon receipt of an acceptance, the workcell must either seal the contract or back out of it. This second "commitment" is then firm. If one proposer backs out, the Guardian is able to send an acceptance to another proposer.

To analyze our scenario with two-stage commitment, we add a message type, *agree*, which is defined by FIPA, and remove the *reject* message for becoming redundant. The workcell sends *agree* if it is willing to firm up the commitment, otherwise it sends *refuse*.

$$Feedback_j \overset{def}{=} \left(\begin{array}{l} \overline{propose}_{C,j}. \left(\begin{array}{l} \overline{propose}_{F,j}.accept_{C,j}. \left(\begin{array}{l} \overline{agree}_{C,j}.obsCF'_j.\delta.0 \\ + \overline{refuse}_{C,j}.accept_{F,j}. \\ \quad \left(\begin{array}{l} \overline{agree}_{F,j}.obsC'F_j.\delta.0 \\ + \overline{refuse}_{F,j}.obsC'F'_j.\delta.0 \end{array} \right) \end{array} \right) \\ + \overline{refuse}_{F,j}.accept_{C,j}. \left(\begin{array}{l} \overline{agree}_{C,j}.obsC_j.\delta.0 \\ + \overline{refuse}_{C,j}.obsC'_j.\delta.0 \end{array} \right) \end{array} \right) \\ + \overline{refuse}_{C,j}. \left(\begin{array}{l} \overline{propose}_{F,j}.accept_{F,j}. \left(\begin{array}{l} \overline{agree}_{F,j}.obsF_j.\delta.0 \\ + \overline{refuse}_{F,j}.obsF'_j.\delta.0 \end{array} \right) \\ + \overline{refuse}_{F,j}.obsNil_j.\delta.0 \end{array} \right) \\ + \quad \ldots other\ permutation\ elided \ldots \end{array} \right)$$

Because the two-stage approach is more complex than the original, separating out states that can theoretically be combined now becomes a burden. The following two states enable Cheap to react appropriately to all feasible negotiations, assuming sane behavior on the part of the Guardian. If the Guardian were to attempt something insane, like accepting a proposal that was never issued, the separated processes would block the action, but the merged process would blithely progress into a confirmed state.

$$C \overset{def}{=} Uncommitted_C$$

$$Uncommitted_n \overset{def}{=} \delta. \left(\begin{array}{l} \overline{cfp}_{n,1}.propose_{n,1}.Uncommitted_n \\ + \overline{cfp}_{n,2}.propose_{n,2}.Uncommitted_n \\ + \overline{cfp}_{n,3}.propose_{n,3}.Uncommitted_n \\ + \overline{accept}_{n,1}.agree_{n,1}.Confirmed_n \\ + \overline{accept}_{n,2}.agree_{n,2}.Confirmed_n \\ + \overline{accept}_{n,3}.agree_{n,3}.Confirmed_n \end{array} \right)$$

$$Confirmed_n \overset{def}{=} (2).Uncommitted_n \oplus \left(\begin{array}{l} \overline{cfp}_{n,1}.refuse_{n,1}.Confirmed_n \\ + \overline{accept}_{n,1}.refuse_{n,1}.Confirmed_n \\ + \overline{cfp}_{n,2}.refuse_{n,2}.Confirmed_n \\ + \overline{accept}_{n,2}.refuse_{n,2}.Confirmed_n \\ + \overline{cfp}_{n,3}.refuse_{n,3}.Confirmed_n \\ + \overline{accept}_{n,3}.refuse_{n,3}.Confirmed_n \end{array} \right)$$

$$F \overset{def}{=} Unc$$

$$Unc \overset{def}{=} \delta. \left(\begin{array}{l} \overline{cfp}_{F,1}.propose_{F,1}.Unc \\ + \overline{cfp}_{F,2}.propose_{F,2}.Unc \\ + \overline{cfp}_{F,3}.propose_{F,3}.Unc \\ + \overline{accept}_{F,1}.agree_{F,1}.Co \\ + \overline{accept}_{F,2}.agree_{F,2}.Co \\ + \overline{accept}_{F,3}.agree_{F,3}.Co \end{array} \right)$$

$$Co \overset{def}{=} (1).Unc \oplus \left(\begin{array}{l} \overline{cfp}_{F,1}.propose_{F,1}.Co \\ + \overline{accept}_{F,1}.agree_{F,1}.Co' \\ + \overline{cfp}_{F,2}.propose_{F,2}.Co \\ + \overline{accept}_{F,2}.agree_{F,2}.Co' \\ + \overline{cfp}_{F,3}.propose_{F,3}.Co \\ + \overline{accept}_{F,3}.agree_{F,3}.Co' \end{array} \right)$$

$$Co' \overset{def}{=} (2).Unc \oplus \left(\begin{array}{l} \overline{cfp}_{F,1}.refuse_{F,1}.Co' \\ + \overline{accept}_{F,1}.refuse_{F,1}.Co' \\ + \overline{cfp}_{F,2}.refuse_{F,2}.Co' \\ + \overline{accept}_{F,2}.refuse_{F,2}.Co' \\ + \overline{cfp}_{F,3}.refuse_{F,3}.Co' \\ + \overline{accept}_{F,3}.refuse_{F,3}.Co' \end{array} \right)$$

The resulting system has 2304 states and 72 distinct observations, none of which allow any jobs to fail. We again have three distinct system behaviors:

1. C and F each bid on all three jobs. C is forced to back out of two of them, which are then awarded to F. There are 18 ways this can happen.
2. C and F compete for two jobs, while the third is only bid on by F. C backs out of one of the two contested jobs, which is then awarded to F. There are 36 ways this can happen.

3. C and F compete for one job, which is awarded to C, while the other two are only bid on by F. There are 18 ways this can happen.

F never fails to bid on all three jobs because it cannot possibly win the first awarded contract.

7. CONCLUSION

Our experiences support the accepted wisdom that obtaining globally coherent behavior from autonomous agents is an ambitious goal. Nevertheless, a simple change to two-stage commitment sufficed in this case. The generally applicable result coming from this study is not our particular solution, which might not be appropriate in every coordination scenario, but our approach to diagnosing and troubleshooting the agent coordination. Our experiences suggest that run-time visibility into the logic determining agents' behaviors is crucial to reaching a full understanding of how the coordination failed, which is needed to enable rigorous analysis of possible solutions. Unsophisticated tweaking of agents' logic is not likely to yield a completely correct solution on its own, yet meaningful validation of potential solutions can only occur if the problem itself has been fully understood.

The difficulties that we had in achieving visibility point to areas where emerging standards can be improved. The FIPA specification leaves testability out of scope [3], and the reasons embedded in Reject and Refuse messages are implementation-dependent content. Future standards for agent infrastructures might standardize the communication of reasons to enable the development of interoperable testing and debugging tools and facilitate effective analysis of coordination problems.

8. APPENDIX: TCCS NOTATION

It is not possible to capture accurate and full semantics for TCCS in the space available. For a complete and formal introduction, please refer to the cited references [5][6].

Table 2. TCCS notation guide

Example	Semantics
$cfp_{C,j}$	Send this message to a receiving process. Not possible to proceed if no such process exists.
$\overline{cfp}_{C,j}$	Receive this message from a sending process. Not possible to proceed if no such process exists.
$X.Y$	Do X, then do Y.
$X \mid Y$	Do X and Y in parallel.
$\prod_{1<=j<=3} X_j$	$= X_1 \mid X_2 \mid X_3$
$X + Y$	Do whichever of X or Y is possible at this time. If both are possible, choose one.
$X \oplus Y$	If X or Y can send or receive a message at this time, then do it. Otherwise, progress through time, eliminating whichever process is not prepared to idle.

A process cannot idle unless it is explicitly enabled to do so by one of the following:

$(1).X$	Idle for one clock-tick, then do X.
$\delta.X$	Idle until X becomes possible, then do it.

0 signifies a deadlock process that cannot do anything; hence, $\delta.0$ means "idle forever" and is used in lieu of subprocess termination.

9. REFERENCES

[1] Davis, R., and Smith, R.G. Negotiation as a metaphor for distributed problem solving. Artificial Intelligence 20, 1 (1983), 63-103.

[2] Edinburgh Concurrency Workbench home page. http://www.dcs.ed.ac.uk/home/cwb/.

[3] FIPA Architectural Overview (99/07/09), section 6.1.13. Available from http://www.fipa.org/.

[4] FIPA '97 Specification, Version 2.0, Part 2 (Agent Communication Language), section 6.5.16. Available from http://www.fipa.org/.

[5] Milner, R. Communication and concurrency. Prentice Hall, 1989.

[6] Moller, F., and Tofts, C. A temporal calculus of communicating systems. Lecture Notes in Computer Science #458, Springer-Verlag, 1990, 401-415.

[7] Zeus home page. http://193.113.209.147/projects/agents/zeus/index.htm.

BIOGRAPHY

David Flater is a computer scientist at the U.S. National Institute of Standards and Technology. He came to NIST in 1992 while completing his Ph.D. in Computer Science from the University of Maryland. Since his transfer to the Manufacturing Engineering Laboratory in 1996 he has contributed to a number of projects involving object- and agent-oriented systems for engineering and manufacturing. Presently he is active in the Manufacturing Domain Task Force and the Test and Validation Special Interest Group of the Object Management Group.