MULTIPLE AGENT FORMALISMS FOR COORDINATION IN ORGANIZATIONAL PROBLEMS

by

Hardeep Johar
Leonard N. Stern School of Business
New York University
New York, NY 10003

and

Vasant Dhar
Leonard N. Stern School of Business
New York University
New York, NY 10003

November 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-36

Abstract

Many organizational problems are ill-structured where the structure of a problem is not apparent at the outset of the problem solving process. Agents responsible for these problems often decompose them into subproblems the solution of which is the responibility of other agents. These problems are only nearly independent in the sense that temporal and technical dependencies exist between the different subproblems. Since the problems are interdependent, coordinating the activities of the different agents is important for ensuring that the partial solutions discovered by these different agents are not conflicting in terms of global consistency. Usual mechanisms for coordination include communication and negotiation between agents of interrelated problems. In this paper we describe a formalism for coordination in multiple agent ill-structured problems based on four properties of tasks, atomicity, serializability, completeness and soundness. We examine how these properties are essential for handling conflict resolution. We also outline some requirements for control.

1 Introduction

Many organizational problems are ill-structured in the sense that the specifications of the problems emerge *during* the problem solving process [25]. Often, these problems require the coordination of many developers (agents) usually working reasonably independently. In such problems, although there may be identifiable global states and goals of the problem-solving process, each agent has only a partial and inexact view of these [17]. Furthermore, each agent is required to create and execute plans for the purpose of goal (or sub-goal) fulfillment. Since these activities are never completely independent [26, 25], coordination of the activities of these agents is an important aspect of purposeful work. A key aspect of this coordination is to ensure that the object under development remains consistent to the extent possible, i.e. attainment of the overall goal stays plausible, in the face of the purposeful activities of each agent. Many development projects are ill-structured [25], i.e. they are characterized by incremental specification and changes in specifications. Changes in goals, subgoals, commitments, and constraints could lead to complex dependencies between activities of different agents, and raises issues of control that include maintaining consistency, reallocation of resources (money, men, machines, and material), revision of subgoals and plan specifications, and recognition of subgoal fulfillment. Examples of these kinds of problems include engineering design [1], software design [3], and, in practise, many organizational problems.

2 The nature of tasks

The solution process in ill-structured problems usually requires that the problem be decomposed into sub-problems, either to reduce complexity [5, 25, 26] or to exploit differences in agent expertise (spatial distribution of knowledge cf. [13]). These decomposed subproblems are then the responsibility of other agents for whom the subproblem becomes a task with a goal to be fulfilled. These latter agents may recursively decompose their subproblems into smaller subproblems and contract other agents to fulfill the task objectives. For example, in the design of a software application, the entire task may be decomposed into analysis, design and other subtasks (figure 1) and the design task may be further decomposed into subtasks (figure 2). We refer to the agent decomposing the task as the *contracting agent.*

The basic unit of activity that each agent works with is a task. Since tasks are recursively decomposed, the entire set of tasks can be viewed as forming a tree where the goals of each task are a subset of the goals of its parent. Each task should be independent in the sense that no two subtasks of a task should have goals in common and subtasks of one task should not have any goals in common with subtasks of other tasks, though subtasks do have goals in common with their supertasks. We call a set of tasks that satisfy this property *decomposition independent.* Decomposition independence requires that if two tasks have any goal in common, then the goals of one task must be a subset of the goals of the other and ensures that each task has atmost one supertask. Decomposition independence is important in multiple agent problems since problems are recursively decomposed and two agents may end up working on the same goal. While, in single agent domains this would not occur, and in group problem solving domains this may be desirable, in a multiple agent domain when the focus is on a single overall goal, distributing goals may result in the wastage of resources. For example, in a software development project, it would be counter-productive to have more than one agent working independently on any of the tasks in figure 2. Formally[1]:

Definition 1 Decomposition Independence: *A set of tasks* $T_1.....T_n$ *is* **decomposition independent** *if and only if* $\forall T_i \exists T_j goal(T_i) \cap goal(T_j) \neq \phi \Rightarrow goal(T_i) \subseteq goal(T_j) \bigvee goal(T_j) \subseteq goal(T_i)$

2.1 Task Outcomes as Commitments

A task is usually deemed complete when an agent has selected actions that fulfill the current goals of the task and satisfies current constraints. At this point the agent declares an end to the problem solving

[1]Some notational conventions: T_i refers to a task, $agent(T_i)$ refers to the agent responsible for the task, and $goal(T_i)$ refers to the set of goals of the task T_i.

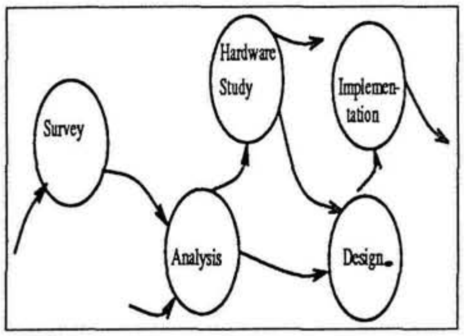

Figure 1: System development process

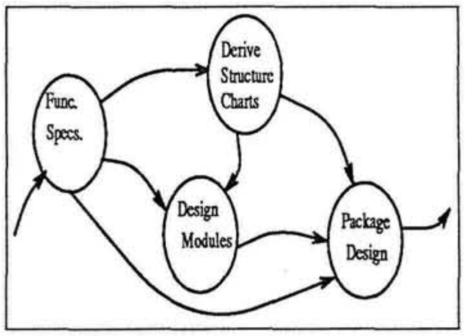

Figure 2: Details of design process

process. When the agent makes such a declaration, he or she *commits* (in the sense of [6]) to that course of action and other agents can reasonably expect that the commitment will be kept [13]. Commitments made by one agent constrain the activities of other agents (whether or not the commitments are carried out), and in this sense the outcome of any task can be viewed as a constraint generating commitment

Commitments made by one agent constrain choices of other agents in two ways: through *resource constraints* and through *technical constraints.* Commitments made by one agent constrain resources available to other agents since they involve either the consumption of resources, or the planned consumption of resources from a finite set of available resources. Commitments made by one agent may constrain the options available to other agents because dependencies may exist between the choices available to the two agents. For example, in the design of a fluid pump, the choice of a metal for the casing reduces the set of fluid choices and vice versa [18].

Because of resource and technical constraints, two types of dependencies arise between the commitments required of agents, *temporal dependencies* and *design dependencies. Temporal dependencies* arise because commitments made by one agent constrain resources available to agents that commit in the future. If the sequence of commitments is reversed, the dependency will be reversed. Temporal dependencies are often expressed by identifying sequences for commitments, as for example in project networks such as a PERT chart, in programming flowcharts, and in data flow diagrams. The arrows in figure 1 reflect temporal dependencies between activities. *Design dependencies* reflect the inherent relationship between the commitments required from the agents. For example, a design dependency exists between the task of an agent (perhaps human or robot) that adds a new job to queue in a machine shop environment and the agent that picks the next job from the queue because both need to have similar assumptions about the type of queue they are dealing with (LIFO/FIFO/priority-based). This dependency is independent of the sequence of their activities in the sense that even if there were no temporal relationship between the activities, the dependency would still exist, though often design dependencies are expressed temporally.

2.2 Commitments and conflicts

Conflicts arise when, because of temporal and design dependencies, two or more agents make commitments that are incompatible. Agents may make incompatible decisions because of several reasons well documented in the literature: because problem solving control is local, but local decisions have to be globally coherent [25, 13]; there is a great deal of task uncertainity [28] and task ambiguity [8] in ill-structured tasks, and; agents may make assumptions arising from incompatible world views or microtheories [16]. Conflict detection and resolution is important because conflicts often go undetected until very late in the process when resolution is expensive. For example, in software development it has been estimated that, because of these incompatible design decisions, maintenance costs could be as much as 70% of total costs [23]. When conflicts occur, the process of conflict resolution involves identifying the agents affected by the conflicts, communicating the causes of the conflict to these agents, and changing one or more commitments until the conflict is resolved.

3 Tasks and dependencies

Because of the possibility of conflicts, whenever an agent executes a task, he or she may have to communicate information about the task to other agents, determine if a conflict has arisen, and seek to resolve the conflict. At each instance of a communication need, the agent needs answers to three questions: *what should be communicated, to whom should the communication be addressed*, and *when is a communication necessary.* The answers to these questions are important because if conflict related information is not communicated, then an inconsistent artifact that needs extensive modification may be the result. If, on the other hand, communication is excessive, then the coordination costs may raise the overall cost of the project prohibitively. For example, Durfee and Montgomery [9] show that the cost of planning could be excessive in either a no communication, or an over communication scenario, and the implication is that the ideal amount of communication depends upon task characteristics. Since the need for communication arises from dependencies between tasks, coordination protocols that handle communication should use these dependencies to determine what, when, and to whom to communicate. In this section we examine four properties of task dependencies: *atomicity, serializability, completeness*, and *soundness* that are necessary for answering these questions.

3.1 Atomicity: What to Communicate

Consider the example in figure 3. Agent A has the task of writing a program that gets an input from the keyboard and places the input in a queue. Agent B's task is to write a program that is a consumer of queue members. The way in which agent B's program will access the queue depends on agent A's design choices. For example, agent A may choose between using a LIFO, FIFO or priority-based queue. Thus a dependency $T_B \longleftarrow T_A$ exists between the two tasks. The existence of this dependency implies that when agent A completes or makes a modification to the task, a message should be sent to agent B indicating relevant design decisions or assumptions. For example, when agent A completes the task, a message of the form "Global Queue X is a FIFO queue because I assume that in the absence of any reason otherwise a FIFO queue is preferable" should be sent to agent B. If, however, the dependency is set up to include agent A's entire task in T_A, then along with the queue decision message, agent B may also receive a message of the form "I assumed that the keyboard is an IBM PC keyboard" which is an unnecessary communication. The reason for this is that the dependency $T_B \longleftarrow T_A$ is too general and it does not indicate the aspect of T_A on which T_B is dependent. T_A is defined too generally. Tasks in any dependency should contain only those aspects that are useful. We call such tasks *atomic* tasks.

Atomicity refers to the property that the tasks that are considered in coordination should be atomic. Atomicity is a desirable property in the knowledge base of any system [2], but is hard to define independently of dependencies. For example, in software modification literature atomicity is defined as being met if "all or none of the changes of a modification activity are involved in a merge" [15] indicating that atomicity is a property related to effects rather than to an artifact itself. When multiple agents are present, tasks contain subtasks and are therefore not purely atomic in terms of goals. We provide a definition for atomic tasks that makes use of task dependencies and acknowledges the presence of multiple agents. In a pure sense, atomicity guarantees that tasks have single outcomes, however, our definition requires atomicity only at the level of dependencies. For example, agent A may be required to write a program that accepts input from a keyboard and creates a FIFO queue from this input. Since this task has two outcomes, the collection of input, and a FIFO queue, it would have to be divided into two to satisfy the property of atomicity.

Definition 2 Atomic Tasks: *Given a set of task dependencies F and a set of tasks $T_1 \ldots T_n$; A Task T_i is an atomic task if and only if*

$(i) \forall T_j \longleftarrow T_i \in F \; \nexists T_k goal(T_k) \subset goal(T_i) \bigwedge T_j \longleftarrow T_k.$

$(ii) \forall (T_l \longleftarrow T_k \in F) T_j \longleftarrow T_i \bigwedge goal(T_l) \cap goal(T_j) = \phi \bigwedge goal(T_k) \subseteq goal(T_i) \bigwedge agent(T_i) = agent(T_k) \Rightarrow T_k = T_i$.

$(iii)\ \forall (T_j \longleftarrow T_i \in F)\ \nexists T_k goal(T_k) \subset goal(T_j) \bigwedge T_k \longleftarrow T_i$.

$(iv)\ \forall (T_j \longleftarrow T_i \in F)\ \not\exists T_k T_l goal(T_k) \subset goal(T_i) \bigwedge goal(T_l) \subset goal(T_j) \bigwedge T_l \longleftarrow T_k \bigwedge agent(T_i) = agent(T_k)$.

Statement (i) says that if T_j is dependent on T_i then T_j is not dependent on any proper subset of T_i. Statement (iii) says that if T_j depends on T_i then no proper subset of T_j depends on T_i. Statement (ii) says that dependencies should be minimal across other dependencies. For example, if agent A's task involves getting an input, coding the input, and updating the queue; and agent B has to write a program that decodes a queue element, then agent B's task depends on both the coding task and the queue update task. What statement (ii) says is that if there is any other task that is dependent on either the coding or the queue updation, then the dependency between agent A's two subtasks, and agent B's task should be decomposed into two dependencies. This ensures that dependencies are transitive. Statement (iv) allows the existence of dependencies between subtasks when a higher level dependency exists, provided the agents are different[2].

Atomicity ensures that only relevant dependencies are recorded. Dependencies are created because commitments made by one agent constrain other agents. Commitments made by one agent must therefore be communicated to agents responsible for tasks constrained by that commitment. On the other hand, communication is not a costless activity, and should therefore take place only when relevant. The communication problem is therefore one of communicating only appropriate information. The advantage of recording only relevant dependencies and using these as a mechanism for determining communication requirements is that communication can be kept within relevant levels. Atomicity provides an answer to the question *what should be communicated*?

When dependencies are not recorded atomically, an *over-communication* or a *no-communication* scenario may occur. Suppose one of agent B's tasks is a consumer of the queue created by agent A. If agent A's task is not decomposed to the atomic level then a dependency will be set up between the task "get input from keyboard and insert into queue" and "get next candidate from queue". A change in the program that gets input from the keyboard (perhaps because of a change in keyboard specification) will necessitate a message to agent B giving information on the change since agent A will have no way of knowing that the dependency really relates to the second task. This is an example of an *over-communication* scenario.

An even more serious problem arises when agent B does not know of agent A's decisions. Suppose agent B's program requires a LIFO queue rather than a FIFO queue and the agent assumes that the queue type is LIFO. If T_A is not decomposed then agent A may not even recognize that the type of queue is important. In this situation it is possible that the dependency is never recognized and a *no-communication* scenario results with negative effects on consistency.

3.2 Serializability: When to Communicate

Consider again the example in section 3.1 (figure 3). Since task T_{B1} is dependent on task T_{A1b}, for every occurrence of task T_{A1b} there must be an occurrence of task T_{B1} in the future. The occurrence may result in no change to the program, but agent B must have at least considered the ramifications of agent A's commitment. Thus, to ensure consistency of the overall artifact, the entire sequence of tasks carried out over the development lifetime should have the property that for every dependent activity in the set of final dependencies, there should be no activity that it depends on following its last occurrence. We call a sequence of tasks that satisfies this property *serializable*. The reference to final dependencies is important because in an ill-structured problem the dependencies could change over time. Serializability provides an answer to the question *when to communicate*. A communication is necessary whenever the execution of a task results in a non-serializable task sequence.

[2] We assume here that if if the agent responsible for a task is the same person or body as the contracting agent, then the different roles makes it possible to treat the agent as two entities.

While serializability has been discussed in the context of concurrency [27, 15], that research does not adequately address inconsistencies that arise from dependencies. The literature on concurrency focuses on concurrent modifications and ensures serializability by the use of locking protocols. Locking protocols may not, however, be enough in a organizational problems. For example, when agent A has task {A1: Write a program to update the queue} and agent B has task {B1: Write a program to get next queue item} and a dependency $T_{B1} \longleftarrow T_{A1}$ exists, assume that the agents have completed their tasks under the common assumption of a FIFO queue. Now agent A revises the program so that the queue created is a priority queue. Locking will not help unless agent B attempts to access the queue type concurrently, and the resulting inconsistency will go undetected. Locking is insufficient because the dependency between the state of the queue update program when B1 is completed and the design decision made in completing task A1 is not recorded. To ensure serializability, either B1 needs to be reactivated so that it's commitment occurs later than that of A1, or agent A should be prevented from revising it's commitment. This can be made possible by recording the dependency between the two tasks, and by using this dependency to detect situations where serializability is compromised. Since dependencies are created by commitments, recording the rationale behind commitments can be used to resolve the conflict or to support negotiation for conflict resolution.

Definition 3 *Let $S=\{ T_1 \ldots T_n \}$ be a temporally ordered sequence of tasks, F be a set of task dependencies, and $t(x)$ represent the time at which event x occurs. S is a* **serial sequence** *if and only if $\forall T_i \longleftarrow T_j \in F, t(commitment(T_j)) \leq t(commitment(T_i))$*[3]*. S is* **serializable** *if and only if it is equivalent to some serial sequence.*

3.3 Completeness and Soundness: To Whom to Communicate

Completeness refers to the property that it should be possible to propagate causal relations between activities to other related activities. If dependencies are recorded between atomic tasks then the only requirement on the inference procedure for propagating dependencies is that it should be transitive. For example, if $T_i \longleftarrow T_j$ and $T_j \longleftarrow T_k$ then completeness requires that $T_i \longleftarrow T_k$, and that it should be possible to make this inference.

Completeness is important in a multiple agent scenario because whenever conflicts have to be resolved, all interested parties need to be involved in the resolution process. Consider, for example, a situation where three agents, a hardware analyst, a systems software engineer, and an applications developer, are involved in developing a system for creating business reports for an organization. Their tasks and goals are { T_{HW}: Select a print device}, { T_{SS}: Create a language for writing reports}, and { T_{AS}: Create report definition formats based on user requirements}. Dependency $T_{SS} \longleftarrow T_{HW}$ exists because the capabilities of the language depend on the printer available (graphics/no-graphics, layout options, etc.). $T_{AS} \longleftarrow T_{SS}$ exists since the capabilities of the language constrain report format possibilities. These dependencies are explicit in the sense that they are declared by the agents. There is also, however, an implicit dependency $T_{AS} \longleftarrow T_{HW}$, since the capabilities of the printer affect the language primitives which affect the report format possibilities and therefore both the applications developer and the software engineer may be affected by a change in the printer.

Completeness in a coordination protocol ensures that the question "to whom should a communication be addressed" is correctly answered. In the example above, a communication indicating a potential change in the printer, implications of this change, and the possibility of a conflict needs to be generated and sent to the applications developer and the software engineer. Since the change affects all three agents, before deciding on a course of action, the system should have the ability to reason about the implications of the change to all affected tasks.

Soundness refers to the property that every dependency that follows from a given set of dependencies is a valid dependency. If the dependencies determined by the consistency detection mechanism are not sound, then messages may be sent to inappropriate agents and this could result in serious *over-communication* problems. If dependencies are recorded on atomic tasks, and the inference procedure uses transitivity,

[3]The equality is necessary because tasks may be mutually dependent.

then soundness is guaranteed. The coordination protocol must therefore ensure that dependencies are recorded atomically[4].

Definition 4 Soundness and Completeness: *Given a set of dependencies F, A coordination protocol is* **complete** *iff* $\forall T_i \longleftarrow T_j F \models T_i \longleftarrow T_j \Rightarrow F \vdash T_i \longleftarrow T_j$, *and* **sound** *iff* $\forall T_i \longleftarrow T_j F \vdash T_i \longleftarrow T_j \Rightarrow F \models T_i \longleftarrow T_j$.

4 Coordination

4.1 Conflict Resolution

Conflicts may occur whenever a sequence of tasks becomes non-serializable. The process of conflict resolution, therefore, involves the creation of a serializable sequence of tasks. For example, consider the example involving agents HW, SS and AS where a change in a budget constraint forces agent HW to select a new printer. The resulting new commitment could potentially render the commitments of agents SS and AS inconsistent creating a conflict. The conflict could be resolved either by reactivating T_{SS} and T_{AS}, or by retaining the old commitment made by HW by relaxing the budget constraint. Each of these alternatives results in a serializable task schedule and thus the process of conflict resolution involves selecting one serializable schedule from the options available.

It is important to note that conflict resolution should be restricted only to the cases where serializability is compromised. Global coherence requires that the artifact under construction be consistent, but does not require that individual assumptions made by agents be consistent if global coherence is not compromised. For example, it is legitimate for the hardware agent to assume that high resolution graphics are not necessary, while an applications software agent assumes that high resolution graphics are absolutely necessary if the printer acquired does support high resolution graphics anyway, since global coherence is not compromised. The coordination effort sbould support conflict detection at the level of commitments, not at the level of assumptions, but should support conflict resolution at both levels.

A coordination protocol must provide support for conflict resolution at two levels, by identifying the agents affected, and by supporting the process of negotiation. Conflict resolution usually involves negotiation between the agents affected by the conflict [11, 7, 16]. The coordination protocol must provide support for conflict resolution by identifying the minimal set of agents involved in a conflict. This requires that the procedure for determining dependencies be complete and sound, and that dependencies are atomic. Since agent AS is a final consumer of the printer, negotiations involving a change in printer specifications must involve that agent[5].

4.2 Coordination and Control

While serializability provides a set of outcomes for the negotiation process, the basis for negotiation really depends on examining the assumptions behind the design decisions of the parties involved, the cost of redoing tasks that would be affected by the resolution, and the need to confirm to global coherence requirements. Choosing an alternative for resolving a conflict would be made on the basis of which best achieves global objectives. In any event, the outcome of the process may involve the reallocation of resources between the affected agents. The problem of control is *how best can this resource allocation process be managed* and a coordination protocol must provide support for managing this process.

[4]Definition 2 defines atomicity in terms of the entire set of dependencies. Satisfying definition 2 does not guarantee pure atomicity and therefore does not ensure soundness though it does ensure completeness. As explained in section 3.1 pure atomicity is hard to determine and therefore we are forced to make do with the weaker requirement that soundness be satisfied with respect to dependencies.

[5]Note that negotiation protocols [7, 11] usually require that only the contracting parties be involved in a negotiation. They do not attempt to involve all the affected parties.

The problem of managing resources has received very little attention in the literature, with the possible exception of Kornfeld and Hewitt's work on resource sponsors in distributed problem solving [19]. Most other research examines resource management from the perspective of time-constrained problem solving in real time systems [22, 20] where the focus is on finding a solution in a reasonable amount of time rather than on managing a set of resources to make resource constrained decisions.

In Kornfeld and Hewitt's terminology, *resource sponsors* are agents that provide resources to problem solving agents under the principles that resources should not be wasted on establishing what is already known, and that resources should be apportioned amongst competing approaches since sponsors do not necessarily know which approach is right [19]. The notion of resource sponsors corresponds somewhat to that of contracting agents introduced in section 2. By making control decisions, either by acting as the final arbiter during negotiation, or by unilaterally resolving conflicts (depending on the form of the control organization), contracting agents distribute resources in the same way as resource sponsors. Kornfeld and Hewitt incorporate the notion of sponsors into problem solving representations by attaching a sponsor to each activity, and withholding sponsors from goals that are already established or are fruitless to pursue. Including contracting agents in task objects is analogous to attaching sponsors to each goal. There are, however, significant differences between the two concepts that make it difficult to apply Kornfeld and Hewitt's representations to decision making problems. First, contracting agents are a part of a hierarchy of resource sponsors and may not actually own the resources that they distribute. Second, it may be possible for them to negotiate more resources from their contracting agents. Third, and more significantly, while sponsors are external entities with no problem solving function, contracting agents are themselves a part of the problem solving system and are responsible for the goals that they and their agents are pursuing. When sponsors or contracting agents are reponsible for the goals for which they allocate resources, they need mechanisms to reason about allocation decisions, and the main limitation of the work on resource sponsors is that it does not provide any mechanisms for representing and reasoning about resources.

In order to reason about the different alternatives available, contracting agents need to have the ability to evaluate the resource implications of the different alternatives. The implication of this is that it is not enough to record knowledge about the resource consumption and commitment of tentative decisions made by agents, but it is also necessary to record knowledge about resource commitments required by options that an agent explored but rejected (either because they were inferior, or because they violated some constraint). These resource commitments form the basis for choosing amongst alternatives. Since each alternative involves a change in a commitment made by one or more agents, agents need a measure for the cost of changing a commitment. We call this measure the *strength of commitment.*

The *strength of a commitment* incorporates resource implications of commitments and some estimate of the alternate commitments that an agent could make. When commitments are viewed in terms of resources, it becomes possible to distinguish between *strong commitments*, where resources have already been consumed, and *weak commitments* where resources are commited but not consumed. Similarly, if there is only one way in which an agent can satisfy a global goal, then the commitment is strong. Mechanisms for measuring the strength of a commitment will come from the domain of interest and here we give an example from the software development domain.

Consider the situation where a contracting agent has to resolve a conflict between an applications agent who, because of certain user requirements, has decided on a report that requires the printing of 3-dimensional piecharts, and a hardware selection agent who, because of a budget constraint, has selected a printer that is incapable of printing graphs (Figure 4). The conflict could be resolved by relaxing the budget constraint and selecting a graphics printer, or the user has to make do with a tabular report, or more resources are allocated to one or both of the agents to explore new alternatives. Each option has different resource and goal implications. For example, relaxing the budgetary constraint or exploring new alternatives involves the consumption of more resources, while dropping the 3-dimensional piechart may compromise a global service requirement. The choice would depend on the strength of the commitments made by the two agents which would depend on resources consumed and on options available. For example, if the printer has already been purchased, then the commitment made by the hardware selection agent is a strong commitment and if the number of alternatives available to the applications agent are few (or only one) then that commitment is strong.

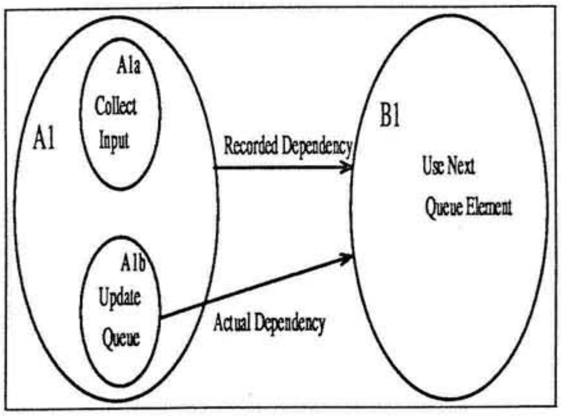

Figure 3: Atomicity and Task Dependency

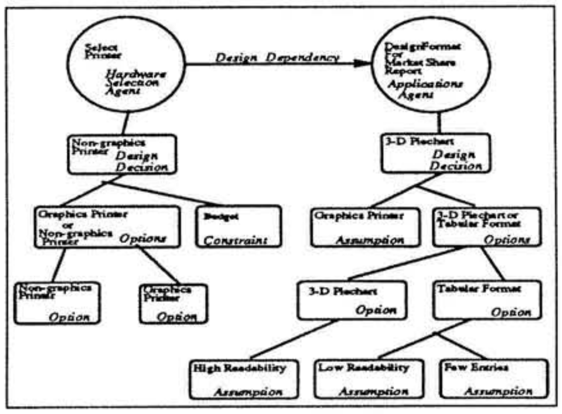

Figure 4: Example of control options

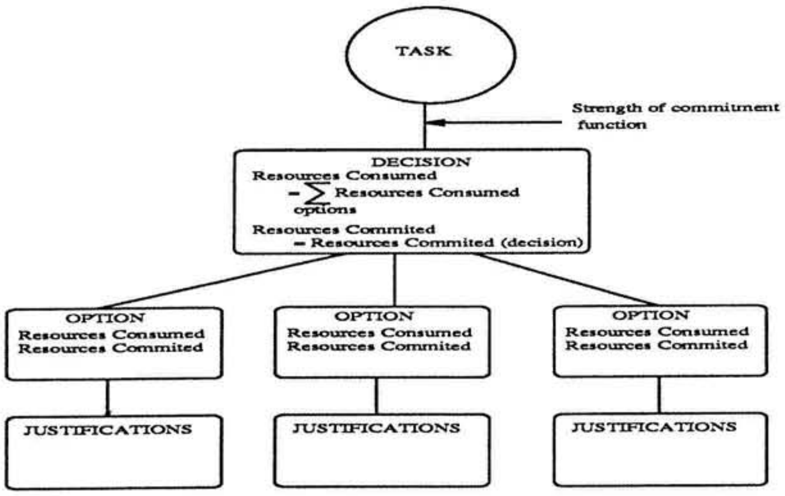

Figure 5: Knowledge Requirements for control

For control, the contracting agent needs to have access to the different options considered by agents as well as knowledge on resource commitments. If the problem is ill-structured, specifications change, and an option that violates a constraint at some point of time, may be a possible solution at a later point of time. Therefore, resource commitments and resource consumption need to be recorded for each option considered whether the option is a possible solution or not. The resource commitment for the task is the resource requirement of the commitment. The resource consumption of the task is the sum of resources consumed by all options in the option set. The strength of a commitment is some function of resources committed, resources consumed, and the number of options in the option set with the from of the function depending on the domain. For example, strength could be a simple function like the ratio of resources consumed to resources commited. Figure 5 illustrates knowledge requirements for control.

5 Related research and discussion

Multiple agent systems have been the focus of much research in recent years though the problem of multiple agent organizational decision making has received little attention. Gasser et al. [14] and Fox [12] examine the organization of distributed systems with the objective of discovering how systems are distributed in organizations using organizational paradigms. Our focus is more on supporting cooperation in organizational problem solving, and on formalisms for reducing the uncertainity and complexity amongst the agents, and in this sense, is complementary to that stream of research. Our research is perhaps closest to the FA/C paradigm [21, 4] where agents exchange agents exchange tentative and partial solutions with the objective of converging on a solution. The FA/C paradigm is, however, better suited for diagnosis applications rather than at finding solutions, and for groups of agents solving overlapping problems (as also are blackboard systems [10]). Similarly, in the DATMS architecture [24], agents exchange knowledge about inconsistencies and results to arrive at an interpretation of data, and this architecture is also better suited for interpretation and diagnosis. A major difference between our work and the research described above (as well as some other DAI approaches e.g. [7, 9]) is that they assume that either the decomposition and problem knowledge is known beforehand, or atleast at the time the problem is decomposed.

We are currently working on an architecture for supporting organizational problem solving using the formalisms developed in this paper. The architecture uses a multiple agent ATMS (MA-ATMS) [18] as a reasoning mechanism and an additional component for managing tasks and task dependencies. The MA-ATMS records all problem solving knowledge and in that sense is different from the DATMS [24] where each agent has an ATMS, and permits agents to have inconsistent assumptions, enforces global consistency at the level of design decisions, and can maintain a record of alternative solutions, and is, therefore different from the DTMS [2].

Finally, we summarize the complexity involved in ensuring that tasks have the properties specified in this paper. An algorithm for decomposition independence would be $O(g)$ where g is the average number of goals per task if the check is made only when a new task is created, and if no new goals are created. If new goals are created then the algorithm is $O(ng)$ in the worst case where n is the number of tasks. Similarly, an algorithm for atomicity would be $O(3n + k)$ in practise where k is the average number of dependencies per task, and atomicity is checked when a dependency is added since only the task dependencies, and dependencies for its supertask need be examined. The serializability algorithm is $O(nk)$.

References

[1] A. H. Bond. The Cooperation of Experts in Engineering Design. In L. Gasser and M. H. Huhns, editors, *Distributed Artificial Intelligence Volume II*, pages 463–486. Morgan Kaufmann, San Mateo, 1989.

[2] D. M. Bridgeland and M. N. Huhns. Distributed Truth Maintenance. In *Proceedings Eighth National Conference on Artificial Intelligence*, pages 72–77, Boston, Ma., July-August 1990.

[3] F. P. Brooks. The Mythical Man-Month. In P. Freeman and A. I. Wasserman, editors, *Tutorial on Software Design Techniques*, pages 35–42. IEEE Computer Society Press, Silver Spring, Maryland, 1983.

[4] N. Carver, Z. Cvetanovic, and V. Lesser. Sophisticated Cooperation in FA/C Distributed Problem Solving Systems. In *Proceedings of AAAI-91*, pages 191–198. MIT Press, 1991.

[5] B. Chandrasekaran. Design Problem Solving: A Task Analysis. *AI Magazine*, 11:59–71, Winter 1990.

[6] P. R. Cohen and H. J. Levesque. Intention is Choice with Commitment. *Artificial Intelligence*, 42:213–261, 1990.

[7] R. Davis and R. G. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20:63–109, 1983.

[8] V. Dhar and M. H. Olson. Assumptions Underlying Systems that Support Work Group Collaboration. In M. H. Olson, editor, *Technological Support for Work Group Collaboration*, pages 33–50. Lawrence Erlbaum Associates Inc., Hillsdale, New Jersey, 1989.

[9] E. H. Durfee and T. A. Montgomery. A Hierarchical Protocol for Coordinating Multiagent Behaviors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 86–93, 1990.

[10] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainity. *ACM Computing Surveys*, 12(2):213–253, 1980.

[11] R. E. Fikes. A Commitment-Based Framework for Describing Informal Cooperative Work. *Cognitive Science*, 6:331–347, 1982.

[12] M. S. FOx. An Organizational View of Distributed Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):70–80, January 1981.

[13] L. Gasser. Social Conceptions of Knowledge and Action: DAI Foundations and Open Systems Semantics. *Artificial Intelligence*, 47:107–138, 1991.

[14] L. Gasser, N. F. Rouquette, R. W. Hill, and J. Lieb. Representing and Using Organizational Knowledge in DAI systems. In L. Gasser and M. N. Huhns, editors, *Distributed Artificial Intelligence Volume II*, pages 55–78. Pitman/Morgan Kauffman, London, 1989.

[15] W. H. Harrison, H. Ossher, and P. F. Sweeney. Coordinating Concurrent Development. In *CSCW 90 Proceedings*, pages 157–168, New York, 1990. The Association for Computing Machinery.

[16] C. W. Hewitt. Open Information Systems Semantics for Distributed Artificial Intelligence. *Artificial Intelligence*, 47:79–106, 1991.

[17] M. N. Huhns, editor. *Distributed Artificial Intelligence Volume I.* Morgan Kaufmann, Los Altos, California, 1987.

[18] H. Johar and V. Dhar. An Extended ATMS for Decomposable Problems. Technical Report STERN-91-3, New York University, 1991.

[19] W. A. Kornfeld and C. E. Hewitt. The Scientific Community Metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):24–33, January 1981.

[20] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Cao, and J. Y. Read. Real-Time Knowledge-Based Systems. *Artificial Intelligence Magazine*, 9(1):27–45, Spring 1988.

[21] V. Lesser and D. D. Corkill. Functionally Accurate, Cooperative Distributed Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):81–96, January 1981.

[22] V. R. Lesser, J. Pavlin, and E. H. Durfee. Approximate Processing in Real Time Problem Solving. *AI Magazine*, 9(1):49–61, 1988.

[23] J. Martin and D. McClure. *The Problem of Software Maintenance and its Solution.* Wiley, 1983.

[24] C. L. Mason and R. R. Johnson. DATMS: A Framework for Distributed Assumption Based Reasoning. In L. Gasser and M. H. Huhns, editors, *Distributed Artificial Intelligence Volume 2*, pages 293–318. Morgan Kaufmann, San Mateo, 1989.

[25] H. A. Simon. The Structure of Ill-structured Problems. *Artificial Intelligence*, 4(1):181–202, 1973.

[26] M. Stefik. Planning With Constraints. Technical Report STAN-CS-784, Stanford University Computer Science Department, January 1980.

[27] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume I.* Computer Science Press, Maryland, 1988.

[28] O. E. Williamson. *Markets and Hierarchies.* Free Press, New York, 1975.