DEPENDENCY BASED COORDINATION FOR CONSISTENT SOLUTIONS IN DISTRIBUTED WORK

by

Hardeep Johar
The Leonard N. Stern School of Business
New York University
Information Systems Department
New York, NY 10012

and

Vasant Dhar
The Leonard N. Stern School of Business
New York University
Information Systems Department
New York, NY 10012

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

<u>**Working Paper Series**</u>

STERN IS-92-38

Proceedings of the First International Conference on Information and Knowledge Management 1992, ISMM Press - 1992

Dependency Based Coordination for Consistent Solutions in Distributed Work*

Hardeep Johar
Information Systems Department
New York University
New York, NY 10003

Vasant Dhar
Information Systems Department
New York University
New York, NY 10003

Abstract

Many organizational problems can be decomposed into *nearly independent* subproblems the solution of which is the responsibility of independent agents. In this kind of work, which we call distributed work, the problems are only nearly independent since dependencies exist between the commitments required from each agent. As a consequence of these dependencies, the coordination problem becomes one of maintaining a consistent global solution in the face of the possibly conflicting activities of each agent. We define a normative model for coordination protocols that indicates the formal requirements for maintaining a globally consistent solution. The model identifies several properties that the protocol must enforce, namely *serializability*, *atomicity*, *completeness*, and *soundness*. We show that these properties are desirable in coordination protocols for distributed work problems.

1 Introduction

Many organizational problems are ill-structured in the sense that the specifications of the problems emerge *during* the problem solving process [26]. Often, these problems require the coordination of many developers (agents) usually working reasonably independently. We refer to work involving these kinds of ill-structured problems as *distributed work*. 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 [19]. 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 [27, 26], 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. Examples of these problems include engineering design [1], software design [3], and many other organizational problems. For example, the design of the Boeing 777 involves 5600 designers scattered over 18 locations each working on independent parts of the aircraft that must fit together. In addition, Boeing subcontracts pieces of the design to external entities who must also design parts that fit with the parts being designed by Boeing [22].

Inconsistencies in the object under development arise because agents search for locally consistent solutions which may conflict with the locally consistent solutions found by other agents. Agents search locally rather than globally because of two kinds of uncertainties associated with their tasks: (1) uncertainties due to incomplete knowledge about what states of nature will prevail [29] which arise partly because the agent does not know what design decisions other agents will make; and (2) uncertainties that result from the ambiguous nature of task specifications [11] which arises because tasks are ill-structured. These locally found solutions lead to global inconsistencies because there may be dependencies between the tasks of agents, and they may make assumptions that conflict with the assumptions made by other agents. It is possible that either the ramifications of these dependencies goes unnoticed, or these dependencies themselves remain unidentified. The problem of local control versus global coherence has been recognized as an important problem in the literature [16] and we present a model that addresses this issue utilizing dependencies to aid coordination activities. The model outlines a set of properties that a coordination protocol must satisfy and we show that these properties are useful in ensuring globally consistent solutions.

*We thank Alex Tuzhilin and Albert Croker for their constructive comments on early versions of this paper.

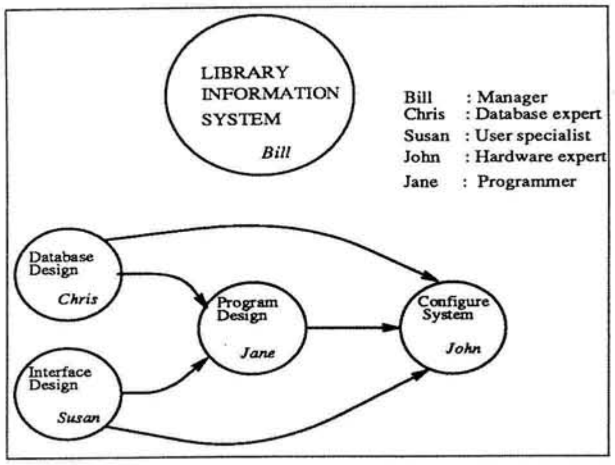

Figure 1: Task Decomposition in the Library Information System

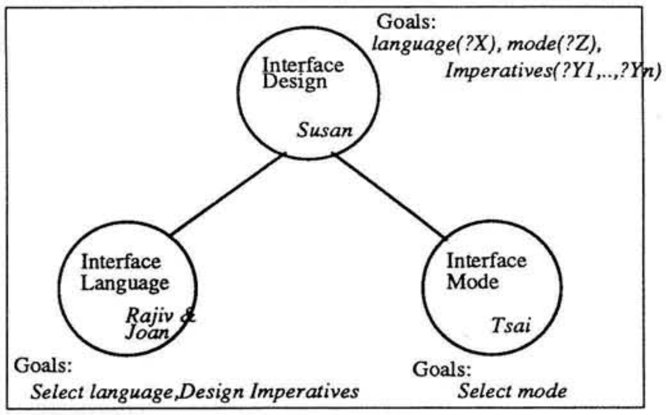

Figure 2: Goals and Tasks

2 Formal Model

2.1 Tasks

The basic unit of activity in distributed work is a *task*. Each task has two identifying characteristics, a set of related goals the achievement of which signifies that the task is completed, and an agent responsible for the task. An *agent* is a problem solving entity that utilizes domain knowledge to find a solution that satisfies all goals of the task. Each task has one agent associated with it, though the agent may be a single human, a group of humans, or a computer program. For example, in the design of an aircraft, the agent responsible for wing design could be the wing design group, the agent responsible for deigning trailing edge panels could be Mr. Smith, and the agent responsible for selecting landing gear could be a computer program (perhaps an expert system).

An agent may either complete a task on his or her own, or may decompose the task either to reduce complexity [5, 26, 27], or to exploit differences in agent expertise [16]. The *Library Information System* example in Figure 1 illustrates both these reasons for task decomposition. The decomposition process involves identification of goals that need to be satisfied for task completion, and parceling out groups of goals to other agents in the form of subtasks. In Figure 2 the *Interface Design* task has three goals that have been handed over to two agents to satisfy. Any of these agents may further subcontract the goals. Susan's task will be complete when all three of her goals are satisfied[1]. Since these problems may be ill-structured, the goals could change over time. For example, perhaps Susan's original goals were *language(?X), Mode(?Y)* and the $imperatives(?Y_1, ..., ?Y_n)$ goal was added later when Rajiv and Joan decided on a command driven interface. In the remainder of this paper T_i refers to a task, $agent(T_i)$ refers to the agent responsible for the task, and $\mathcal{GOAL}(T_i, \mathrm{t})$ refers to the set of goals associated with the task at time t.

2.2 Commitments and dependencies

The problem solving process for each agent's task can be viewed as a process of identifying alternative solutions and the choosing of a subset of these alternatives as a solution. The set of alternative solutions is referred to as a *choice set* [8], and we call the subset of this set that the agent identifies as a solution, a *commitment*. Because of the two types of uncertainties associated with tasks, each agent makes assumptions about possible states of the world that may occur either resulting from the activities of other agents, or from the ill-structured nature of the task. Therefore, each choice in a choice set has associated with it a set of *assumptions* under which the choice is feasible. Since assumptions are associated with uncertainties, the truth or falsity of a particular assumption may change, invalidating an existing commitment and forcing the agent to make a new commitment. $\mathcal{CHOICE}(T_i) = \{CHOICE_1(T_i), ..., CHOICE_n(T_i)\}$ represents the choice set for task T_i, $C_{T_i,\mathrm{t}}$ represents the commitment made by $agent(T_i)$ at time t, and $\mathcal{A}(CHOICE_j(T_i))$ represents the set of assumptions under which

[1]Allowing multiple goals in a task is useful because (1) it allows agents to make partial commitments and (2) often it is desirable to package related (conjunctive) goals together and hand them to an agent (perhaps a planner such as **TWEAK** [6]).

$CHOICE_j(T_i)$ is valid. We provide a formal definition of commitment below and then explain the intuition behind the definition.

Definition 1 Commitment: *Let $\mathcal{A}(x)$ be the set of assumptions for a choice x, S_s be a state of the world, $\mathcal{GOAL}(T_i, \mathrm{t})$ be the set of goals of task T_i at time* t. *A choice $C_{T_i,\mathrm{t}}$ is a* **commitment** *if*
*(i) (*intention*) agent(T_i) reasonably intends to fulfill the choice [7]*
*(ii) (*multiple commitments*)* t *is the time at which the agent first communicates the choice to at least one other agent*
*(iii) (*local feasibility*) $\exists_s \forall_{i,j} S_s \wedge CHOICE_j(T_i) \in C_{T_i,t} \Rightarrow \mathcal{A}(CHOICE_j(T_i)) \subseteq S_s$, where S_s represents a consistent state of the world.*
*(iv) (*goal satisfying*) If S_f is the state resulting from the application of $C_{T_i,\mathrm{t}}$ to S_s then $\forall_j goal_j \in \mathcal{GOAL}(T_i, \mathrm{t}) \Rightarrow \exists_k goal_k \in S_f \wedge goal_j = goal_k$ (i.e. the commitment satisfies all goals of the task that existed at time* t*)*

For example, the choice set for *Rajiv and Joan* (Figure 2) may be *{interface(command-driven), interface(menu-driven)}*. Alternative *interface(command-driven)* may be valid under the assumptions *{desired-interface(flexible), user-literacy(high)}*. Since the problem is ill-structured, they may be forced to make assumptions about the user. For example, they may assume *user-literacy(high)* and may commit to a command-driven interface as their commitment. If, at some later point, it emerges that this assumption was not valid, they would retract this commitment (it would not satisfy the local consistency condition) and make a new one.

Definition 1 states first that a commitment is an intention. Commitments as intentions is fairly common in distributed work settings. For example, in engineering design a commitment corresponds to a part drawing or a process plan for manufacturing, in software development a commitment corresponds to a program specification or to data flow diagrams. A choice becomes a commitment when it is communicated or made available to some other agent. At this point the second agent has the expectation that the commitment will be met and can use this knowledge in making its own commitments (see [7] for a discussion on commitment and intention in plans). The definition includes a temporal component because commitments may be tentative. A commitment may need to be changed because of a change in goals, because of the introduction of a new constraint, because of conflicts with other commitments, or because the agent discovers a better solution. Both engineering and software design problems exhibit this behavior where designs are often modified or extended for the reasons listed above. Commitments must also be valid local solutions in the sense that the assumptions on which they are based should not be mutually inconsistent, and the commitment should satisfy all goals of the task at the time the commitment is made.

That assumptions form the basis for reasoning about problem solutions and therefore are a basis for selecting a commitment has been well documented in the literature. For example, assumption surfacing as a basis for conflict resolution has been the focus of some distributed architectures [23], the use of preconditions and filters to guide operator selection in automated planners [14, 6] is an example of assumption-based commitment (the preconditions, and the current state embody the assumptions), and the use of assumptions has been empirically validated in some distributed problem solving domains such as software development [9].

Commitments made by one task may constrain the possibilities for other tasks. For example, once a particular database software has been selected by Chris, the hardware platform choices available to John may be constrained. Commitments constrain tasks options because dependencies exist between tasks. We define two types of dependencies, *temporal dependencies* and *design dependencies*, below.

Definition 2 *A dependency $T_i \longrightarrow T_j$ is a* **temporal dependency** *if there is an a-priori ordering of commitments such that task T_i must commit before task T_j.*

Definition 3 *A dependency $T_i \longrightarrow T_j$ is a* **design dependency** *if there is at least one $CHOICE_l(T_i)$ that invalidates (is inconsistent with) some element $CHOICE_m(T_j)$.*

Temporal dependencies are usually based either on relationships that have been recognized by the designers and made explicit, or on the creation of an ordering of tasks to reduce complexity. PERT charts and data flow diagrams reflect these dependencies by directed arcs between tasks. In Figure 1 the directed arcs reflect these dependencies.

Design dependencies, on the other hand, reflect inherent relationships between tasks that have not necessarily been identified by an agent and may or may not be temporally specifiable. The essential relationship between the tasks is that the outcome of one task may have the potential to clobber another task (perhaps by invalidating one of the second tasks preconditions [6]). In ill-structured problems, however, this relationship between two commitments may be hard to detect. For example, in Figure 2 it may not be obvious that there is a dependency between the

Interface Language and *Interface Mode* tasks. However, there is a dependency between the tasks since some choices of interface invalidate some choices of language. For example, the selection of a touchscreen interface would invalidate the possibility of using a command driven interface language. This dependency would be detected only if a constraint of the form $\neg language(command) \vee \neg mode(touchscreen)$ were explicitly modeled in the system.

In a normative sense, design dependencies arise because agents may make incompatible assumptions when attempting to make a commitment. If the problem were not ill-structured, then these assumptions would have taken on the status of premises and though the dependency would remain, it would not be a conflict causing dependency. For example, the assumption behind the selection of a touchscreen interface device could be that *Tsai* believes that the target user of the *Library Information System* is not good at using computers, while the selection of a command-driven interface may depend on the assumption that the user is good at using computers and would therefore want a flexible interface. The two agents have made incompatible assumptions that results in a conflict causing design dependency. If the problem had been clearly specified, the type of user would have had the status of a premise, and would have been known to both agents, and the dependency would not have been conflict causing. Definition 4 formalizes this idea.

Definition 4 *Let $a(x,y)$ be an assumption of the form object* x *is a* y. *A Dependency $T_i \longrightarrow T_j$ is a* **conflict-causing design dependency** *if* $\exists_{l,m} CHOICE_l(T_i) \in C_{T_i,\mathrm{t}} \wedge CHOICE_m(T_j) \in C_{T_k,\mathrm{t}} \Rightarrow \exists_{x,y_1,y_2} a(x,y_1) \in \mathcal{A}(CHOICE_l(T_i)) \wedge a(x,y_2) \in \mathcal{A}(CHOICE_m(T_j)) \wedge y_1 \neq y_2$.

The above definition states that a design dependency is conflict-causing only if the two tasks make different assumptions. If they make exactly the same assumptions then the assumption is elevated to the status of a premise. The only design dependencies of interest are the ones that cause conflicts and in the remainder of this paper we use the term *design dependencies* to refer to *conflict-causing design dependencies*.

2.3 Coordination Protocols

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. Whenever an agent makes a commitment, three questions need to be answered: *is a communication necessary*, *what needs to be communicated*, and *to whom should the communication be addressed*. A *coordination protocol* is a system designed to help the agent answer these three questions. 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 [12] 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 conflict-causing design dependencies, 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 related to task dependencies: *atomicity*, *serializability*, *completeness*, and *soundness* that coordination protocols need to satisfy so as to provide answers to these questions.

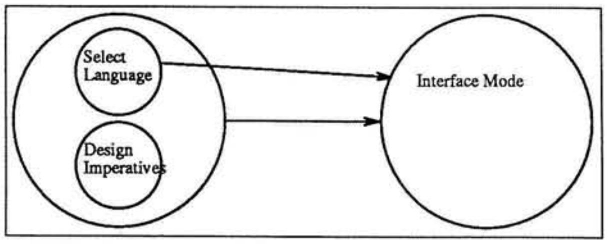

Figure 3: Atomic Dependencies

2.3.1 Atomic Dependencies

The ability to represent relationships concisely has been recognized as a desirable property of any system [2]. In distributed work problems the primary relationship between tasks is the presence of design and temporal dependencies. A coordination protocol should, therefore, ensure that the dependencies in a system are expressed concisely. We refer to this conciseness property as *atomicity*.

The importance of ensuring that dependencies are concisely expressed is illustrated by the example in Figure 3. The two tasks in the figure are the *select language* and *interface mode* tasks of Figure 2. Assume that while decomposing the problem, Susan indicated a temporal dependency between the two tasks as shown in Figure 3 perhaps recognizing that the interface language affects the mode. Rajiv & Joan, the agents responsible for the *interface language* task decompose the task further. As stated in the previous section, a design dependency exists between the *select language* and *interface mode* tasks and domain knowledge shows that no real dependency exists between *design imperatives* and *interface mode*. The existence of the temporal dependency implies that each time a commitment is made (or changed) by any of the subtasks of *interface language*, the impact of the commitment has to be evaluated by the agent responsible for the *interface mode* task even though this should not be necessary for commitments made by the *design imperatives* task. Thus, non-atomic dependencies could lead to unnecessary communication which result in wasteful or damaging problem solving activity.

Definition 5 *A dependency* $T_i \longrightarrow T_j$ *is an atomic dependency if and only if*
(i) $\neg\exists_k T_k \wedge \mathcal{GOAL}(T_k) \subseteq \mathcal{GOAL}(T_i) \wedge T_k \neq T_i \wedge T_k \longrightarrow T_j$,
(ii) $\neg\exists_k T_k \wedge \mathcal{GOAL}(T_k) \subseteq \mathcal{GOAL}(T_j) \wedge T_k \neq T_j \wedge T_i \longrightarrow T_k$,
(iii) $\neg\exists_{k,l} T_k \wedge T_l \wedge T_k \longrightarrow T_l \wedge \mathcal{GOAL}(T_l) \subseteq \mathcal{GOAL}(T_i) \wedge \mathcal{GOAL}(T_k) \cap \mathcal{GOAL}(T_i) = \phi \wedge T_l \neq T_i$, *and*
(iv) $\neg\exists_{k,l} T_k \wedge T_l \wedge T_k \longrightarrow T_l \wedge \mathcal{GOAL}(T_k) \subseteq \mathcal{GOAL}(T_j) \wedge \mathcal{GOAL}(T_l) \cap \mathcal{GOAL}(T_j) = \phi \wedge T_k \neq T_j$

Statement (i) says that if T_j is dependent on T_i then T_j is not dependent on any subtask of T_i. Statement (ii) says that if T_j depends on T_i then no subtask of T_j depends on T_i. Statement (iii) and statement (iv) require that a dependency should be minimal with respect to other dependencies that involve subtasks of the tasks in the dependency.

2.3.2 Complete and Sound Protocols

Completeness refers to the property that it should be possible to propagate causal relations between tasks to other related tasks. For example, if $T_i \longrightarrow T_j$ and $T_j \longrightarrow T_k$ then it follows that there is an implicit dependency between T_i and T_k. Completeness in a protocol requires that it should be possible to make this inference. If the coordination protocol is not complete then there is the possibility that inconsistencies in the artifact will persist because of relationships that remain undetected. Formally:

Definition 6 Complete protocols: *Given a set of dependencies F, A coordination protocol is* **complete** *iff* $\forall (T_i \longrightarrow T_j)\ F \models (T_i \longrightarrow T_j) \Rightarrow F \vdash (T_i \longrightarrow T_j)$.

Soundness refers to the property that every dependency identified by the coordination protocol is a valid dependency. If the dependencies determined by the protocol are not sound then inappropriate instances of coordination may be identified resulting in higher coordination costs. Formally:

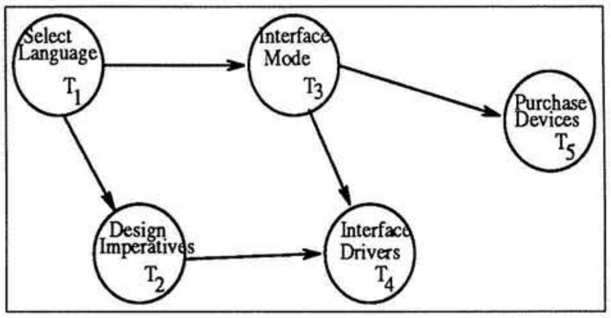

Figure 4: Design dependencies

Definition 7 Sound protocols: *Given a set of dependencies F, A coordination protocol is* **sound** *iff* $\forall (T_i \longrightarrow T_j)\ F \vdash (T_i \longrightarrow T_j) \Rightarrow F \models (T_i \longrightarrow T_j)$.

2.3.3 Serializability of commitments

The literature on database concurrency control discusses the notion of *serializability* as a test for ensuring the integrity of a database that is accessed concurrently [28]. Recent research in software design has extended the notion of serializability to the modification of software artifacts [24, 18]. Serializability in database literature [28] and software development literature [18] focuses on the effects of transactions or modifications to an artifact (a database or a software artifact) and uses techniques such as locking protocols to ensure integrity. However, in distributed work problems, inconsistencies arise because of dependencies between tasks, rather than because two or more agents are attempting simultaneous modification of a database or a program module. For example, Figure 4 illustrates a dependency network for the *Library Information System* described earlier. A change in commitment by the $agent(T_1)$ may affect the commitment possibilities for $agent(T_2)$, perhaps forcing a change in commitment. A locking protocol would lock access to T_1, and perhaps also to T_2 while the commitment was being revised, but would do nothing to ensure that T_2 modified its commitment.

The notion of serializability as discussed above does not consider dependencies or commitments. A sequence of commitments containing T_1 and T_2 will be serializable only if there is at least one commitment made by the $agent(T_2)$ after every occurrence of a commitment by $agent(T_1)$. More generally, a sequence of commitments will be serializable only if the above relationship holds for all dependencies present in the set of tasks. Formally:

Definition 8 *Let,* $\mathcal{T} = \{T_1, ..., T_n\}$ *be a set of tasks and* $\mathcal{C}(\mathcal{T})$ *be the sequence of temporally ordered commitments of tasks in* $\mathcal{T}$. *Let F be the set of dependencies* *defined on $\mathcal{T}$ and F^+ be the closure of F. $\mathcal{C}$ is a* **serial sequence** *iff* $\forall_{i,j}\ (T_i \longrightarrow T_j \in F^+) \Rightarrow \neg\exists_{l,m} C_{T_i,t_l} \in \mathcal{C} \wedge C_{T_j,t_m} \in \mathcal{C} \wedge t_l \leq t_m$. *$\mathcal{C}$ is* **serializable** *if it is equivalent to some serial sequence.*

In Figure 4 one possible serial sequence of commitments is $\{C_{T_1,1}, C_{T_3,5}, C_{T_2,5}, C_{T_5,6}, C_{T_4,7}\}$. A serial sequence is a dependency preserving sequence of commitments. A sequence $\{C_{T_1,1}, C_{T_3,2}, C_{T_2,3}, C_{T_1,4}, C_{T_2,5}, C_{T_3,5}, C_{T_5,6}, C_{T_4,7}\}$ is serializable since it is equal to the serial sequence above if only the final commitments of a task are considered. Thus *equivalence* in the above definition has the same meaning as in database notions of serializability (c.f [28]).

3 Properties of the model

The normative model of coordination described above can be used to show the validity of some intuitive aspects of coordination. In this section we state some of these results and outline intuitive proofs.

Theorem 1 *If a coordination protocol is complete and sound, the protocol will detect a* minimal *set of agents potentially involved in a coordination activity.*

This follows from the following lemmas:

lemma 1 *If a coordination protocol is complete, the protocol will detect a* covering *set of agents potentially involved in a coordination activity.*

lemma 2 *If a coordination protocol is sound, the coordination protocol will detect only agents actually involved in the coordination activity.*

Completeness guarantees that given a particular dependency, all dependencies that follow from it can be determined. For example, if $agent(T_1)$ (Figure 4) makes a new commitment, effective coordination requires that this commitment be communicated to the agents responsible for all other tasks in the figure. This will be possible only if the protocol is capable of detecting $T_3 \longrightarrow T_4$ and $T_3 \longrightarrow T_5$ (presuming that the other two dependencies have been independently detected). (The possibility of cycles is briefly taken up below.) Since completeness does not guarantee that every dependency detected will be a true dependency, the set of agents may contain agents not affected by the commitment. Soundness ensures that only true dependencies will be detected and that the set is minimal.

While the possibility of communicating commitments to the entire set of agents may appear to impact communication costs adversely, it is important to note that the *purchase devices* task is affected by the outcome of the *select language* task, and that, based on some previous commitment by $agent(T_1)$, $agent(T_5)$ may have already purchased some devices or expended problem solving resources. Thus $agent(T_5)$ is *potentially* affected by the change in $agent(T_1)$'s commitment. Completeness and soundness help answer the question *to whom should a communication be sent.*

Theorem 2 *If dependencies are atomic then* minimal *information will be communicated in a coordination activity.*

Intuitively, atomicity guarantees that the tasks in a dependency will be as specific as possible. For example, in Figure 3, if the true dependency is between *select language* and *interface mode* only information relating to the *select language* task should be communicated. If the dependency was not atomic, information relating to the entire task would be communicated adding to the communication cost. Atomicity helps answer the question *what should be communicated.*

Theorem 3 *Serializability of a sequence of commitments is a sufficient condition for global coherence.*

If a sequence of commitments is serializable, then there exists some equivalent serial sequence which contains exactly one instance of a commitment for each task. By the definition of serial sequences, this equivalent serial sequence orders tasks according to dependencies and for every $T_i \longrightarrow T_j$, the commitment by $agent(T_i)$ precedes the commitment by $agent(T_j)$, and $agent(T_j)$ would have incorporated $agent(T_i)$'s commitment as a constraint, thus, ensuring a consistent artifact. Serializability helps answer the question *when is a communication necessary*, and a simple answer is *whenever serializability is compromised.* A serializable sequence can be constructed to remove the potential inconsistency. Note that serializability is defined on the closure of the set of dependencies and therefore we also have the following lemma:

lemma 3 *Completeness is a necessary condition for serializability.*

Often a dependency network may contain cycles. In fact, most design dependencies will be bi-directional. For example, the *select mode* and *select language* tasks in Figure 4 are each, in a true sense, dependent on the commitments of the other. However, in most distributed work problems, this problem is alleviated by the presence of temporal dependencies provided at the time the problem is decomposed (as in PERT charts and Data Flow Diagrams). These temporal dependencies can be used to force a direction on those lower level dependencies that are bi-directional, as has been done in Figure 4 where T_4 belongs to the *program design* subtask, and T_5 belongs to the *configure system* subtasks in Figure 1. However, cycles cannot be completely eliminated and Figure 5 illustrates a simple cycle.

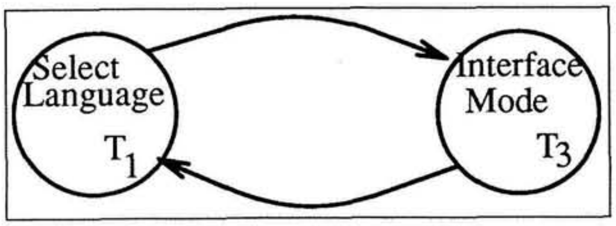

Figure 5: A Simple Cycle.

Theorem 4 *When a dependency network contains cycles, a negotiated commitment is a necessary condition for serializability.*

When cycles are present, a sequence of commitments will be serializable only if, in the equivalent serial sequence, all agents responsible for the tasks in the cycle commit simultaneously. For example, in the simple cycle in Figure 5, the two tasks have to commit simultaneously for Definition 8 to be satisfied, and the equality condition will ensure that it is satisfied. The simultaneous commitment has to be negotiated since each agent has to know the other agent's commitment before making his/her own commitment. The proof extends this reasoning to larger cycles. Interestingly, this result has been a normative guideline in the software development domain where it is recommended that if two programs are interrelated then they should be modified simultaneously (negotiation) and only at the level of specifications (commitment) [25].

4 Related Research

Multiple agent systems have been the focus of research in recent years though the development of protocols that support coordination of the activities of human agents has received little attention. Gasser et al. [17] and Fox [15] examine the organization of distributed systems with the objective of discovering how systems are distributed using organizational paradigms. Our focus is more on supporting cooperation in organizational problem solving, and on formalisms for reducing uncertainty and complexity amongst 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 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 [13]). Similarly, the DATMS architecture [23], where agents exchange knowledge about inconsistencies and results to arrive at an interpretation of data, 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. [10, 12]) is that they assume that the decomposition and problem knowledge is known beforehand, or at least at the time the problem is decomposed. When a problem is clearly specified at the time of decomposition, the coordination problem reduces to one of constructing a serial sequence rather than a serializable sequence of commitments.

5 Conclusion

We have outlined a formal model for protocols that support coordination in activities that involve decomposition of the problem into nearly independent parts, and that require a globally coherent solution. We are working on an architecture for supporting organizational problem solving using the formalisms developed in this paper. The architecture uses a multiple agent assumption-based truth maintenance system (MA-ATMS) [20] as a reasoning mechanism and an additional component for ensuring that dependencies are atomic, and commitments are serializable. We are in the process of developing and testing algorithms for detecting non-serializable commitments, constructing serializable sequences, and for guiding coordination activities to ensure that a sequence is serializable. The MA-ATMS records problem solving knowledge and in that sense is different from the DATMS [23] where each agent has an ATMS. It 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].

References

[1] A. H. Bond. The Cooperation of Experts in Engineering Design. In L. Gasser and M. H. Huhns, editors, *Distributed AI 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. J. 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 Prob-

lem 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] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32:333–377, 1987.

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

[8] A. E. Croker and V. Dhar. A Knowledge Representation for Constraint Satisfaction Problems. Technical Report STERN-IS-90-9, New York University, New York, August 1990.

[9] B. Curtis, H. Krasner, and N. Iscoe. A Field Study of the Software Design Process for Large Systems. *Comm. of the ACM*, 31(11), November 1988.

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

[11] 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.

[12] 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.

[13] 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.

[14] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:198–208, 1971.

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

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

[17] 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.

[18] 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.

[19] M. N. Huhns, editor. *Distributed AI Volume I*. Morgan Kaufmann, Los Altos, California, 1987.

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

[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] J. Markoff. Boeing Takes Off. *The New York Times*, 10th. November 1991.

[23] 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.

[24] C. Pu, G. E. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th. International Conference on Very Large Databases*, pages 26–37, August 1988.

[25] M. Shaw. Abstraction Techniques in Modern Programming Languages. *IEEE Software*, 1(4):10–26, October 1984.

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

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

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

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