Distributed Search and Conflict Management Among Reusable Heterogeneous Agents

Susan Ellen Lander

Computer Science Technical Report 94-32
April 1994

This work was supported by DARPA contract N00014-92-J-1698, Office of Naval Research contract N00014-92-J-1450, and NSF contract CDA 8922572. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Distributed Search and Conflict Management Among Reusable Heterogeneous Agents

Susan Ellen Lander

Computer Science Technical Report 94-32
April 1994

Distributed Search and Conflict Management Among Reusable Heterogeneous Agents

A Dissertation Presented

by

Susan Ellen Lander

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

May 1994

Department of Computer Science

This work was supported by DARPA contract N00014-92-J-1698, Office of Naval Research contract N00014-92-J-1450, and NSF contract CDA 8922572. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

ABSTRACT

DISTRIBUTED SEARCH AND CONFLICT MANAGEMENT AMONG REUSABLE HETEROGENEOUS AGENTS

MAY 1994

SUSAN ELLEN LANDER, UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Victor R. Lesser

Committee Members: John R. Dixon, Graham Gal, and Robert N. Moll

The current state of knowledge-based technology is such that most application systems are built from scratch. In order to move beyond the prohibitive cost of constantly reinventing, rerepresenting, and reimplementing the wheel, researchers are beginning to examine the feasibility of building application systems with heterogeneous and reusable agents. In this dissertation, we explore a comprehensive model of distributed-search systems comprising multiple agents where each agent is a complete and independent system that represents a specific area of expertise. Our approach acknowledges the inevitability of conflict among the agents, and exploits that conflict to drive agent interaction and guide local search.

A primary objective of the research is to understand how to build distributed-search systems that can integrate multiple heterogeneous computational agents. Another objective is to explore the role of conflict from two perspectives: how it affects the quality of solutions within a system and how conflict management can be integrated into coordination strategies for distributed search. To these ends, we introduce the conceptual model we have developed for distributed search and conflict management among reusable heterogeneous agents. We present an implemented framework that provides flexible architectural support for agent integration, coordination, conflict management, and for multiagent solution evaluation. We also describe two implemented application systems built within the framework, and report results from our experimental investigation of the efficiency and effectiveness of agent configurations within those systems.

ACKNOWLEDGEMENTS

I would first like to extend my heartfelt appreciation to my advisor, Victor Lesser. The opportunity to work with him has been tremendously rewarding — he is always willing to explore new frontiers and has a contagious enthusiasm for learning. He gave me a great deal of freedom to choose my own course and yet was always there to pull me back when I wandered off on some unproductive track. Throughout my term as a graduate student, he has been a source of trusted advice. Thank you Victor!

Each of my committee members has made a unique contribution to this dissertation. Robbie Moll helped me to clarify ideas about search and the interaction of agents. He also pushed me to be more concrete in my descriptions which no doubt greatly improves the dissertation. John Dixon supplied domain knowledge and code during the development of STEAM and was able to quickly identify issues that would later become important in understanding how to evaluate solutions and classify cooperation. Graham Gal has a slightly different skew on cooperation and negotiation, and his perceptive comments raised issues that I simply didn't see from my own perspective. It has been a privilege to work with this committee.

There are many career and family issues that continue to affect women who choose research occupations more strongly than men in equivalent positions and, alas, Computer Science is not exactly a hotbed of feminist activity. However, the Computer Science Department at UMass is fortunate enough to have a particularly strong representation of women in leadership roles. I would like to especially thank Lori Clarke and Susan Landau for their efforts in making this department open to and comfortable for women.

I'd like to thank several people who have provided technical assistance to the project. Marg Connell helped design and implement an initial version of the STEAM system—a major effort. Dan Corkill offered occasional, but always on-target, constructive criticism. Kevin Gallagher contributed some (awesome!) programming assistance. Michele Roberts has helped out many times with administrative tasks.

On a personal note, Norman Carver and Carol Broverman have been friends and an inspiration to me as well, proving that all grad students do indeed move on. Malini Bhandaru and Carla Brodley have shared some interesting lunches and have provided both friendship and thoughtful advice. Malini, Marg Connell, Ming Fang, Isazkun Gallastegi, Dave Hildum, Frank Klassner, Dorothy Mammen, Zack "I can fix

anything in 5 minutes" Rubinstein, Tuomas Sandholm, and Toshi Sugawara have all contributed to making the office a pleasant and cooperative place to be. Outside of academia, Elaine Aldrich, Vicky Baum-Hommes, April Stein, and Gina Tucker have provided many bright moments over the years and have brightened many a difficult moment.

For my extended family, I should build a flashing neon thank-you billboard. My parents, in-laws, and siblings have done so much to give me a little extra now and then, whether I needed time, money, help with the children, or just a sympathetic ear. Gloria and Gene Brennan, Adelle and David Corkill, Pam, Karl, and Chris Richardson, Janelle and Mark Lauterbach, Scott Lander, Laurie and Amanda Robert, Jon Lander, and Tammis Pompilli, thank you all.

A very special acknowledgement belongs to my two beautiful children who inspire me with their energy and creativity and who bring great joy to our home. Dani and Kellen, thanks for being so understanding, you guys are definitely the best!

Finally, I thank my husband, Daniel, for his calm and unfailing companionship, support, and, luckily, his wonderful sense of humor. I could not possibly have achieved this without his steadying presence. So Daniel, well—you know.

Table of Contents

APPENDICES

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

Can systems of cooperative, heterogeneous, and reusable agents interact coherently and effectively to solve search problems when the required information and capabilities are distributed among the agents? At some level, the answer must be yes, since humans have great proficiency in doing exactly that. Systems comprising reusable computational agents, however, have yet to achieve the skills required to effectively represent and share information, represent and apply appropriate capabilities, manage conflict, and coordinate their actions. This dissertation investigates mechanisms for capturing and implementing the required expertise in a flexible and domain-independent way.

We begin with an overview of the major contributions of the dissertation and the class of problems that will be addressed. A discussion of the themes to be addressed includes defining cooperative distributed search, describing the role of conflict in directing problem-solving activity, and enumerating the types of problems that can be handled. We continue by providing definitions for some of the terms that will be used throughout the dissertation. The last two sections identify the contributions of the dissertation and detail its structure.

1.1 The Team Concept

In this dissertation, we concentrate on systems that comprise a set of logically-independent modules called *agents* and the structures and mechanisms needed to support their integration.[1] Each agent represents a specific area of expertise. It solves problems in its area of expertise and produces local solutions or criticisms of solutions. Many problems are too large and too complex for a single agent to solve alone. This is certainly true for human agents. Examples of human problem-solving teams can be found in many enterprises such as design, research, business management, and human relations. On the surface, it may not be as apparent that this is true for

[1]Each agent is itself a complete system. However, to avoid confusion, we use the term *agent* to mean a component system and the term *system* to mean an application system comprising an agent set and its integrating framework.

computational agents since computing power has come a long way and has a long way yet to go. No matter how much hardware power is available however, there are arguments for the modularity of a multi-agent based approach that go beyond processing power.

Logical modularity reflects the knowledge-acquisition process in which information is gathered from experts with specialized knowledge. Since the information itself is often inconsistent across areas and possibly from one expert to another even within the same area, it is unlikely that a large system can be built without conflicts in the knowledge being represented. If all conflicts must be resolved at system-development time, the system will be brittle and unable to respond to unanticipated situations since no dynamic contingency capability exists. Furthermore, new knowledge cannot be added without a costly consistency-checking and revision effort. Modular decomposition of expertise allows the software-development effort to focus on critical parts of the software without requiring modifications to be explicitly propagated throughout the entire system. Modular agents represent smaller and more manageable chunks of information, maintaining consistency within local boundaries, but not enforcing congruity across boundaries. Because of their size and because of the uniformity of expertise within an area, modular agents are more easily designed, programmed, debugged, and maintained than larger, monolithic programs.

Systems composed of modular agent sets are more robust than monolithic programs and can show graceful performance degradation in situations where only a subset of the agents are performing correctly. The loss of an agent results in a loss of its particular expertise or viewpoint. Depending on the role of the agent in problem solving, the impact on the overall solution can be large or small. However, the other agents can continue processing without interruption.

Newell suggests that there is a level, the *knowledge level*, at which the behavior of an intelligent agent can be modelled [Newell, 1982]. At this level, knowledge is viewed as being distinct from the representation an agent incorporates to access and manipulate that knowledge. The knowledge level is presented as an extension to a more traditional view of a computer system hierarchy: the device, circuit, logic, register-transfer, program, and configuration levels. It provides a means to abstract away the internal details of an agent and instead concentrate on how the agent interacts with the environment.

> If a system has (and can use) a data structure which can be said to represent something (an object, procedure,...,whatever), then the system itself can also be said to have knowledge, namely the knowledge embodied in that representation about that thing.

> The knowledge level permits predicting and understanding behavior without having an operational model of the processing that is actually being done by the agent.

When working with sets of agents, this knowledge level is paramount in the development of integrating machinery. Important questions at this level include: What can the agent do? What does it need to be told in order to perform some task? What can it tell me? In some cases, perhaps, what *will* it tell me? How can it be motivated to do a task? Will it do its task on time? These types of issues are the foundation of agent interaction.

At the knowledge level, the agent's internal workings are a black box. Representational issues such as the architecture, algorithms, language, and inference engine of an agent are irrelevant to its role within a problem-solving set of agents. However, although it can be safely ignored at the knowledge level, representation is of the utmost importance in building an agent that can manipulate its knowledge efficiently and effectively. Agent modularity allows us to abstract the behavior of an agent to the knowledge level for purposes of system integration while leaving agent developers free to make appropriate implementational choices locally for how to best represent the expertise of an agent. In other words, agent modularity permits the separation of how to best capture some domain of expertise within an agent from its role as a member of an interacting agent set.

Given the advantages of logical modularity and the complexity of the problems that are beginning to be investigated, it is clear that multiagent processing is and will be an important and pervasive problem-solving technique. As is the case when humans approach a complex problem, modular agents must be able to work cooperatively in teams. Teams solve problems that are beyond the scope of any individual agent. As stated by Durfee [Durfee, 1991]:

> Artificial intelligence (AI) has emphasized building "stand-alone systems" that can solve problems with minimal help from other systems (computer or human). These systems have traditionally been brittle, in the sense that they fail miserably when presented with problems even slightly outside of their limited range of expertise. The predominant AI answer to brittleness is to inject more knowledge into a system...
>
> A more powerful, extensible strategy for overcoming the inherent bounds of intelligence present in any finite AI (or natural) system, is to put the system in a society of systems, so that it can draw on a diverse collection of expertise and capabilities in the same way that people overcome the limitations of individuals by banding together into societies that are designed to accomplish what individuals cannot. The ability to flexibly team up and coordinate group activities toward individual and collective goals is a hallmark of intelligence.

1.1.1 Agent Reusability in Team Problem Solving

When information is represented in a modular and logically independent way, the information is potentially *reusable* [Neches *et al.*, 1991]. The computational equivalent to teams of human specialists is the reusable-agent system, a multiagent system for which the agents can be dynamically selected from an existing library and integrated with minimal customized implementation. With reusable agents, diverse types of knowledge can be applied in situations that were not explicitly anticipated at agent-development time. Reusable agents can be integrated into a system based on the requirements of that system. For example, the system developer might take into account such factors as the specific expertise needed to solve the problem, the degree of accuracy required, and the desired level of resource investment.[2] A system developer will assemble stored agents rather than programming each agent from scratch, thereby conserving resources. Furthermore, when agents are stored in a library, it can be assumed that they will be tested rigorously and used repeatedly, thereby leading to increased confidence in their reliability.

In order to build a reusable-agent system, the system developer must ensure that the agent set selected is able to actually achieve all subtasks that are required for the overall task. In other words, the system developer must find a suitable problem decomposition that can be handled by the available agents. Information about each agent's input requirements, processing capabilities, and output content and format must be explicitly and declaratively available. In Chapter 3, we discuss how to select an agent set that covers the required solution space without undue redundancy.

1.1.2 Agent Heterogeneity in Team Problem Solving

A consideration to be included in any discussion of team problem solving is the level of heterogeneity among members of the team. Heterogeneity can be characterized by the types of differences that exist such as:

- *Representation heterogeneity* in
  - architecture
  - algorithm
  - language

[2]The term *system developer* will be used throughout this dissertation to refer to the agent, human or computational, that is in charge of the off-line process of building an integrated agent set in the TEAM framework described in Chapter 4 and supplying the integrating information required.

  - inference engine
  - hardware requirements

- *Knowledge heterogeneity* in:

  - declarative knowledge
  - evaluation criteria for solutions
  - priorities
  - goals
  - capabilities

Agent heterogeneity influences system development in many ways. For example, different languages force translation mechanisms. Knowledge heterogeneity makes conflict resolution an integral part of problem solving and requires the explicit definition of what constitutes an "acceptable" solution across the boundaries of expertise. Heterogeneity in agent capabilities or heterogeneous priorities on the orderings of agent activities necessitate some explicit strategy for the coordination of interacting activities.

1.1.3 Conflict in Team Problem Solving

Team problem solving has some inherent difficulties. The most prominent difficulty lies in trying to integrate the work of heterogeneous agents that may not speak the same language, follow the same procedures for generating solutions, or agree on either the facts that go into creating a solution or the value of a solution when it is proposed.

This research project originally began with an investigation of conflict among heterogeneous expert agents: how and why conflicts occur and how they can be resolved. As the work evolved, it became apparent that conflict does not exist in a vacuum but is always part of a larger search process. We found that:

- Conflict-resolution techniques can be used to direct and coordinate search across the set of agents in the face of inconsistencies. However, resolution of conflict does not guarantee the termination of search.

- Conflict-resolution techniques can be used to guide the relaxation of solution requirements at some agents, thereby selectively restructuring the local search at an agent to meet the needs of the global search problem.[3]

- Search characteristics of individual agents including their local search capabilities and their evaluation criteria for solutions, influence the applicability and effectiveness of specific conflict-resolution techniques.

- Characteristics of interagent relationships, such as the existence of complementary search capabilities, or interacting criteria for solution evaluation, influence the applicability and effectiveness of specific conflict-resolution techniques.

From these observations, it became apparent that the study of conflict must include the more general exploration of the synergy between search and conflict resolution.

Given a set of heterogeneous agents that are expected to be reusable, it is not possible to anticipate and engineer out all potential conflicts at development time since it is not known what knowledge will be contained in the complete system [Hewitt, 1986]. Instead of dealing with conflict through knowledge engineering, an agent must consider conflict to be an integral part of the problem-solving process and address it explicitly as it occurs. Conflicts can occur not only as a result of hostile behavior but also because they are inherent in the nature of an agent set. Conflicts can be due to:

- inconsistent knowledge among agents
- incomplete knowledge and/or incorrect assumptions
- different problem-solving techniques
- different goals, solution evaluation criteria, or priorities

Consider a team of human experts, consisting of a sales manager and an accountant, cooperating in the task of choosing a telephone company for their business. They have the shared goal of selecting an appropriate system for their office, but each individual would like to insure that her own priorities receive top consideration. Unfortunately, many of the individuals' local goals and priorities are conflicting from a global viewpoint. For example, the accountant would like to try Company 1, a company with low long-distance rates, to save money. The sales manager is concerned

[3]The term *local search* is sometimes used to denote a search that is contained in some bounded region of the search space. In this dissertation, we use the term differently to mean a search performed by a single agent in its own locally perceived search space, in other words, a search performed locally by an agent.

about the quality of service and would rather choose Company 2, a company with a known high level of quality.

In this situation, it is very difficult to judge what solution is the "correct" solution since it depends on the criteria used to decide. In truth, there is no right or wrong answer: the company may need the level of service provided by Company 2, but it may also need to save on costs through Company 1. The problem is compounded because some agent(s) may have incomplete knowledge. For example, the sales manager may be assuming that level of quality provided by Company 1 is inadequate because she saw a television commercial that intimated that inadequacy. However, she has no real information on which to base her conclusion. In this case, the other agent may be able to fill in missing information and thereby convince the sales manager that Company 1 is a reasonable choice.

A different scenario occurs when the agents have explicitly conflicting preferences. For example, the sales manager believes the company should select based on quality. The accountant, on the other hand, believes the company should select based on cost. Assuming that a real difference in service quality exists between Companies 1 and 2, in this case, filling in missing information will not remove the conflict. The two team members must reconcile their differences and reach a decision, taking into account both viewpoints. Since solution correctness is an elusive concept, the experts strive for balance. One strategy that can be used for reconciliation is to continue searching for alternate solutions. Perhaps a telephone company exists that would come closer to satisfying both agents though it might be less satisfying to either individually. A solution that is acceptable to all, though not ideal for some (or possibly any), agents is the best that can be hoped for in this situation.

If extending the search does not result in a mutually acceptable solution, one or both of the agents will have to relax some requirement or preference in order to reach agreement. Negotiation techniques have been developed in the realm of human relations (e.g., international relations, market relations) for minimizing the emotional and practical costs of relaxation [Fisher and Ury, 1981, Pruitt, 1981]. We choose to ignore emotional costs for the time being with the assumption that our agents will be purely rational. On the practical side, we have implemented basic relaxation techniques within specific applications (see Chapters 6 and 7). More sophisticated preference-manipulation techniques have been developed in other domains [Fox, 1987, Sathi and Fox, 1989, Sycara, 1985] and can be incorporated into application systems developed within our approach. We will examine the effects of rudimentary constraint-directed bounding and refinement of the mutual solution space in Chapter 8.

There are many potential problems to which a reusable agent can apply its knowledge, each problem requiring a different combination of agents. Bringing together diverse knowledge is a source of robustness and balance which is extremely important in many real-world situations: a structural engineer and an architect work together to design and build a safe and functional building, or a pediatrician and a cardiac specialist consult to help an infant with heart disease. Solutions are generated from a rich and varied body of knowledge, providing the potential for creativity and innovation. Innovative solutions are often the result of combining existing knowledge in new ways [de Bono, 1971]. Despite the benefits of diversity however, conflicts will arise when trying to merge multiple perspectives, goals, priorities, and evaluation criteria for a common good. Conflict is not a completely negative force when viewed in this light however.

In this dissertation, we will be discussing conflict resolution in the context of distributed search. There are two basic ways to characterize approaches to conflict resolution in distributed search: *extended search* and *relaxation*. The first category, *extended search*, is applied by an agent when it recognizes a conflict with another agent in an existing solution. The agent sidesteps the conflict by extending its local search until a solution is found that does not conflict. Extended-search methods are used when an agent believes that an alternate solution that avoids the conflict can be developed if it continues to examine its local solution space. Some general methods that can be implemented as local operators to facilitate extended search in specific situations include integrating external constraints to refine the local view of the composite solution space, finding alternate goal expansions, and using case-based reasoning to find ball-park solutions that can then be refined.

The second distributed-search method, *relaxation*, occurs when an agent relaxes some requirement on a solution, thereby expanding its local search space. Relaxation may lead directly to a solution: a history of previously-generated, but unacceptable, solutions can be kept and one of these solutions may become immediately acceptable. If not, extended search can now be applied in the expanded space with a better chance of finding a solution. Relaxation methods are utilized when it seems likely that the solution space is overconstrained or when the expense of further local search is unjustified. Some general methods that can be implemented as relaxation operators include relaxing or relinquishing constraints, relaxing or relinquishing goals, manipulating sets of constraints (e.g., unlinking, bridging [Pruitt, 1981]), or manipulating evaluation criteria [Sycara, 1985].

Multiagent systems do not traditionally acknowledge or exploit the role of conflict among agents as a driving force in the control of problem-solving activity. However,

for some types of problem solving, conflict is the focal point of interaction and must be explicitly addressed. In heterogeneous, reusable-agent systems, an agent cannot rely on models of other agents to make local decisions since the full agent set is not known at agent-development time. This is a nontrivial characteristic of reusable-agent systems since it seriously undermines the ability of agents to coordinate long-term behavior based on predictions of the activity of others. An agent can plan its own behavior and can make assumptions about interactions that will be required with other agents (for example, an agent may depend on some value being available from an external source). However, although planning for interactions among agents can greatly improve the coherency and efficiency of an agent network, it does not address the short-term dynamic coalitions that form around specific conflicts. In reusable-agent networks, conflicts are unpredictable and require explicit attention. The tasks required for conflict resolution have to be attended to as well as the originally planned tasks. Coordination plans may be preempted or revised, communication of information is reactive rather than scripted, and the amount of communication required intensifies. In other words, conflict stimulates agent interaction and is itself the basis of coordination of conflict-resolution behavior. The resolution of conflict dictates what information is to be exchanged and specifies a set of conflict-resolution tasks that must be added to the agent's scheduled activities.

Klein [Klein, 1991] has suggested that conflict should be elevated to a first-class status: that conflict- and domain-level knowledge should have separate but equal roles in problem-solving activity. We agree that conflict knowledge plays a distinct role and would further suggest that it should be one of the motivating factors in control. Conflict may point out misconceptions, inconsistencies, or incomplete knowledge that agents have about the world and/or the state of problem solving. It may indicate that some agents have unrealistic requirements for solutions given the requirements imposed by the other members of the team. In this dissertation, we explore the role conflict plays in driving team problem-solving activity. To this end, we will look at methods for detecting and resolving conflict, how to select a method that fits the situation, when and how to apply a specific method, and the effect of explicit conflict resolution on system performance.

1.2 Defining the Class of Problems Addressed

This dissertation addresses distributed-search problems that have a natural subproblem decomposition based on either knowledge or representation heterogeneity where each subproblem is solved by an agent. No run-time task decomposition or task allocation capabilities are inherently part of our problem-solving model, although

there is nothing in the techniques we use that would prevent these capabilities from being effective. We are primarily concerned with logical distribution and do not address issues concerning the physical distribution of processes or the distribution of the composite solution space.[4] We assume that the environment is unchanging and that no new data is introduced to the agents during problem solving.[5] Search is initiated by a problem specification that details values, preferences, and/or constraints for some attributes of a complete solution. Typical applications that fall into this classification include layout, design, diagnosis, and contracting.

The agents in the problems that we can solve are non-hostile and cooperative. By non-hostile, we mean that agents will not attempt to mislead each other or to deliberately sabotage another agent's reasoning. This type of agent is sometimes referred to as *benevolent*. The term "cooperative" has been defined in different ways with respect to multiagent systems. It is sometimes used to mean a system of agents that have completely separate local goals and that interact only when interaction can potentially further those goals [Rosenschein and Genesereth, 1985, Zlotkin and Rosenschein, 1990]. An example of this is a consumer/producer system in which some agents need to acquire resources and other agents have resources available to sell. The agents interact occasionally and serendipitously when doing so helps them to achieve their local goals. As an intuitive example, consider two agents, one that wants to buy a car and one that wants to sell a car. The agents are cooperative because they each want what the other has, one a car, the other money. Note that they are also competitive since the buyer wants to spend the least amount of money possible and the seller wants to gain the most amount of money possible. They are willing to search for a mutually acceptable solution because it is possible that both agents can achieve local goals through cooperation. There is certainly no guarantee that they will find a mutually acceptable solution however. We will call this type of system *locally cooperative* and will describe an example buy/sell contract-negotiation system that has this character in Chapter 7.

Another type of cooperation occurs when agents have a global goal to fulfill that overrides their local goals when necessary. We call this *globally cooperative*. It occurs in situations where agents integrate subproblem solutions into a composite solution that is distinct from any single agent's contribution. In this case, interaction is not casual, but rather is an integral part of each agent's agenda. A typical application problem of this type is the design of an artifact where each agent takes responsibility

[4] In Chapter 10, we discuss possible future work in physical agent distribution.

[5] This is in contrast to systems where new data arrives continuously such as a sensor network.

for some subsystem of the complete artifact. We will describe a steam-condenser design system with this characteristic in Chapter 6. Each agent designs a component of a steam condenser and the components are integrated into a complete design. The complete steam condenser is evaluated based on its composite attributes, for example, the total price of the condenser, rather than on the attributes of individual components. Notice, however, that even in this tightly cooperative situation, agents still have local goals that guide their individual search processes and these local goals may still be competitive. To illustrate, each agent may have the local goal of developing a minimum-cost component. However, when the agents attempt to integrate these minimum-cost components it may be found that the components cannot be merged due to inconsistent requirements. One or more of the agents will have to produce a locally higher-cost component in order to find a globally consistent solution. With global cooperation, however, the utility of a particular compromise can be evaluated by its effect on the global solution, whereas in local cooperation, the utility of any compromise is evaluated individually by each agent and may be different for each agent.

1.3 Agent Definitions

In this dissertation, the term *agent* is used to mean a knowledge-based system. Each agent is a complete stand-alone system with specific capabilities that allow it to be included as part of an agent set in the TEAM framework, a flexible and general framework specifically developed for the integration of heterogeneous and reusable agents. TEAM will be described in Chapter 4. We assume that agents can have both knowledge and representation heterogeneity. Agents are benevolent and are either locally or globally cooperative, meaning that an agent is willing to contribute both knowledge and solutions to other agents as appropriate and may be willing to accept solutions that are not locally optimal in order to find a mutually-acceptable solution.

Each agent can solve problems in its limited domain independently. The agent does internal scheduling of its own activities and has private data, knowledge, and history mechanisms. Although it can operate as a separate problem-solving entity, it has specific capabilities which allow it to act as a member of a team. These include:

- A declarative specification of information required to interface to an integration framework.
- The ability to communicate in a shared language. (The agents in the implemented systems that will be presented in later chapters do some syntactic translation to and from a shared language as discussed in Section 8.2.3 of Chapter 8. However we do not deal with determining the semantic equivalence of heterogeneously represented information.)

- Provisions for sharing information in a timely manner during problem-solving.
- Solution initiation, extension, or criticism capabilities, expressed as distributed-search operators.
- Local solution-evaluation capabilities.
- The ability to coordinate an internal agenda of local tasks to be performed with external events.
- (optionally) The ability to learn about other agents' restrictions and preferences.

In the design of an agent, some primary considerations are: 1) what capabilities should the agent have with respect to composite solutions? and 2) what are the potential interactions between this agent's local solutions and other agents' local solutions? An agent must have at least one of the following capabilities (but can have any combination):

- *Solution initiation:* The ability to generate a proposal that solves a local subproblem and that can be used as the beginning of a composite solution.
- *Solution extension:* The ability to generate a proposal that solves a local subproblem and that will extend a partially-specified composite solution. The partially-specified solution may constrain parameters that will influence the local proposal.
- *Criticism:* The ability to criticize a partially- or fully-specified composite solution by offering negative and/or positive feedback. An agent that criticizes a solution need not generate proposals.

These capabilities will be detailed in Chapter 5.

Figure 1.1 describes the potential interactions of each pair of agents in an agent set, i.e., the different ways in which subproblem solutions can interact. In Figure 1.1.a, each oval represents the set of solution attributes of an agent. The two attribute sets overlap, meaning that the two agents share some solution attributes and will need to find consistent values for those attributes. For example, if we have a two-agent system in which one agent ($A1$) has attributes x, y, and z and the second agent ($A2$) has attributes a, b, and z, the value of z must be acceptable to both agents and consistent with the values of x and y at $A1$ and a and b at $A2$. Similarly, in Figure 1.1.b, the attribute set of one agent is completely subsumed by that of the second agent. In this case, the two agents must find consistent values for all attributes of the subsumed agent. In Figure 1.1.c, the two agents must find consistent values for all attributes of both agents. Figure 1.1.d shows the situation that occurs when two agents share no solution attributes while Figure 1.1.e describes how agents can indirectly affect each

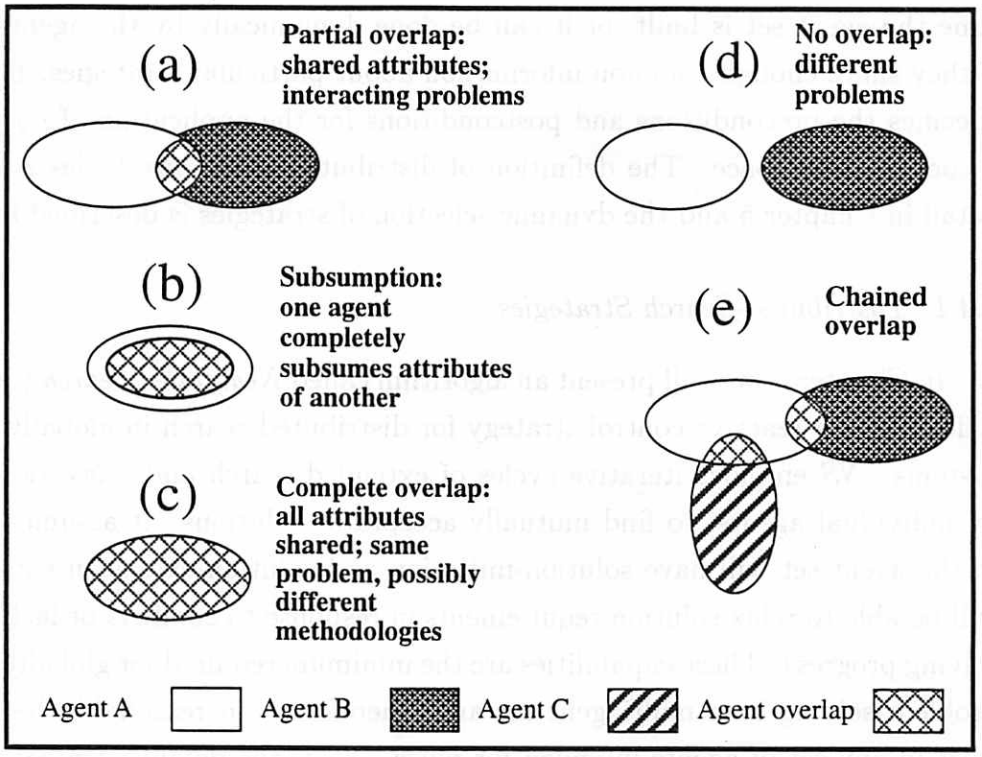


Figure 1.1. *Pairwise Agent Interactions*

other's solutions by each constraining a "middleman" that must find consistent values for solution attributes it shares with each of them even though they don't directly share any attributes.

1.4 Distributed Search

In our formalization of distributed search among cooperative, reusable agents, there are four interwoven processes occurring at each agent: 1) local search for a solution to some subproblem; 2) integration of local subproblem solutions into shared solutions; 3) information exchange to define and refine the shared search space of the agents; and 4) assessment and reassessment of emerging solutions. In the formalization, agents may be simultaneously working on alternative solutions to a problem. This section describes how these activities can be coordinated across a set of heterogeneous agents using distributed-search strategies that specify the role of each agent in deriving a solution. In general, a *distributed-search strategy* specifies a global domain-independent and agent-independent algorithm for the application of a set of search operators. When the algorithm is realized in an agent set, the operators will be distributed across the agents. The selection of a strategy to use and the assignment of operators to agents can be done either by a system developer at the

time the agent set is built, or it can be done dynamically by the agents themselves if they share enough common information about particular strategies. Each strategy specifies the preconditions and postconditions for the application of its operators in a coordination space. The definition of distributed-search strategies is discussed in detail in Chapter 5 and the dynamic selection of strategies is described in Chapter 7.

1.4.1 Distributed-Search Strategies

In Chapter 5, we will present an algorithm called *Negotiated Search* (*NS*) that uses a flexible and reactive control strategy for distributed search in globally cooperative systems. *NS* employs iterative cycles of extended search and relaxation, controlled by individual agents, to find mutually acceptable solutions. It assumes that agents in the agent set will have solution-initiation and solution-extension capabilities and will be able to relax solution requirements in response to conflicts or lack of problem-solving progress. These capabilities are the minimum required for globally cooperative problem solving in a multiagent set and, therefore, can reasonably be expected to exist in any set of agents intended for reuse. *NS* is the default strategy of the TEAM framework, described in Chapter 4, due to its generality and flexibility.

Although *NS* can be widely applied and is powerful enough to find solutions in many different situations, it is a weak search method. It does not exploit specific information or capabilities that may be available to customize the search process. More efficient and effective behavior is likely to result by tailoring search and relaxation mechanisms to the skills of the agents and the desired dynamics of the group. In Chapter 7, we will introduce customized distributed-search strategies that take advantage of the intra- and inter-agent features of a particular agent set and describe when and how they can be applied. In order for a strategy to be effective, the participating agents must exchange enough information to guarantee that each agent understands 1) that a strategy is in effect; 2) what its role is in the strategy; and 3) whatever domain-level knowledge is required to facilitate decision-making.

1.4.2 Distributed-Search Operators

In the TEAM framework, a *distributed-search operator* implements a particular task at an agent. The decision of an agent to apply a particular operator to a problem-solving situation is made within the context of the distributed-search strategy currently in force. Distributed-search strategies and operators will be examined in detail in chapters on negotiated search (Chapter 5) and on customized strategies (Chapter 7).

As a simple and intuitive example, consider a two-agent collaborative design effort in which one agent proposes designs and the second agent extends those designs. The two roles might be called *solution initiation* and *solution extension.* Each of these roles is assigned to one or more agents in the agent set and realized by the agent as one or more implemented distributed-search operators. In a very simple system, the solution-initiation role might have only one operator associated with it, e.g., *initiate-solution.* In a more sophisticated system, there might be several operators associated with that role, e.g., *initiate-solution*, *relax-solution-requirement*, and *terminate-search.* The actual operators associated with each role are defined by the search strategy being used. Notice that an agent may play more than one role simultaneously, either because that is permitted within a single instantiation of the strategy, or because the strategy is instantiated multiple times within the system with each instantiation representing a particular assignment of roles to agents. For example, in the latter case, an agent might act as a solution-initiator for some solutions and as a solution-extender for other solutions.

One of the application systems that we have developed (**STEAM**) does collaborative mechanical design of steam condensers using the *negotiated-search strategy.* The primary roles in this strategy are *solution initiation* and *solution extension* as described above and *solution criticism.* To help make the discussion of search strategies, operators, and roles more concrete, the operator applications of the seven agents in **STEAM** can be traced through a system run in Chapter 5, Section 5.2.

1.4.3 Solution Evaluation

A difficult problem in any multiagent system is to define solution acceptability. We present an evaluation process with two components: first, an agent-level evaluation component and, second, an optional framework-level evaluation component.

Despite the potential for conflict among agents, local evaluation criteria represent the *expertise* of an agent and must be respected. From the individual perspective, this evaluation may be quite complex and detailed. However, from the team perspective, all that matters is whether each agent finds a particular solution acceptable or unacceptable. In order for a solution to be mutually acceptable, all agents must individually find the solution to be acceptable. The mechanisms for collecting and combining agent evaluations will be described in Chapter 4.

In locally cooperative situations, such as consumer/producer contracting, no domain-level evaluation of a final solution is necessary. It is enough that all parties to the transaction are willing to accept the proposed solution. However, in globally cooperative situations, such as multiagent design, it may be that no one agent has

enough global expertise to reasonably evaluate a complete solution. Consider a multiagent system designing a bicycle, where each agent is responsible for the design of some subsystem, e.g., the frame, brakes, gears, etc. Each agent can evaluate proposals for its own subsystem and can evaluate the local impact of external proposals with which it shares attributes. With this form of problem decomposition, however, even though each component of the bicycle might be individually acceptable, the complete bicycle might have unacceptable flaws. For example, it might be unattractive (and therefore unmarketable) or it might be more expensive than the market will bear. These attributes are associated with complete bicycles rather than with subsystem components.

The TEAM framework permits two possible remedies to the problem of fragmented expertise. The first is to include an agent in the agent set that specifically embodies a more global perspective: it has expertise in criticizing either complete solutions or partial solutions comprising multiple components. The second remedy is to permit the system developer to define an evaluation function specifically for complete solutions. This function will be applied by the framework to complete solutions that are acceptable at the agent level. The function can either be used to prune solutions that fall below some threshold or to rank solutions relative to one another.

The two remedies described above are complementary. Defining an agent is more complex and costly but can support highly sophisticated reasoning about acceptability. Defining an evaluation function is easier and faster and may be perfectly satisfactory. The choice of whether to use global evaluation depends on the type of system: a locally cooperative system will not, by definition, use global evaluation. In a globally cooperative system, however, some form of global evaluation should be used and the selection of which type is dependent on factors including the complexity of evaluation within the domain, the accuracy required, and the relative cost of the two mechanisms.

1.4.4 Organizational Structure and Task Decomposition

The heterogeneous agents described in this research are reusable, and as such, they are designed to work without any explicit knowledge of a predefined agent set or agent organization. Rather than relying on information about the existence and capabilities of the other agents that it will be working with to predict interactions and behavior, an agent is developed with a flexible, reactive approach to cooperation. It can then be incorporated into a dynamically-formed set of agents to work on any problem that requires its expertise. However, although TEAM does not rely on the existence of predictive knowledge, it does not preclude the use of organizational

information about the existence and capabilities of the agents in the network when it is available. In fact, any information that can be represented in the common language of a system and that can be formulated and understood by multiple agents within the system can be potentially employed in decision-making.

Conflict-resolution mechanisms are closely tied to the organizational/authoritative structure of the agent set. Information can flow either hierarchically or laterally within the structure as shown in Figure 1.2. We primarily examine issues of *peer*

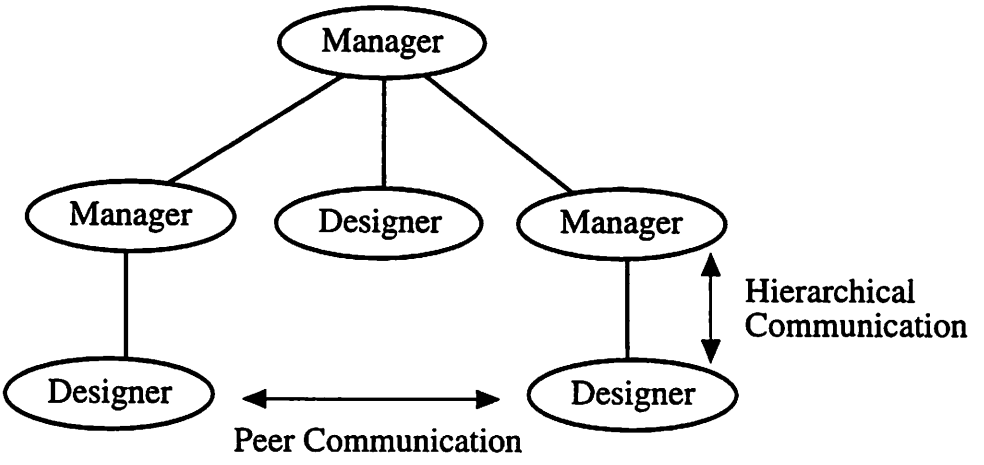


Figure 1.2. *Information Flow Within an Agent Set*

(lateral) conflict resolution. However, the framework we have developed is a highly parameterized, flexible testbed. It can be initialized so that some agents in the organizational structure are given a higher priority, creating a hierarchy in which high-priority agents control the search process. Similarly, the assignment of roles to operators can impose a hierarchical organization across the agent set. Looking at the *solution-initiation* and *solution-extension* roles, for example, it is clear that an agent that initiates solutions has more control over the direction of the search process than an agent that extends existing solutions. In Chapter 9, we investigate the effect of agents taking on different roles within an agent set, and identify characteristics of agents and inter-agent relationships that predict the effectiveness of particular role assignments.

1.4.5 Search-Termination Criteria

In many search domains, solution optimality cannot be guaranteed because the search space is too unstructured and too large. In this type of *satisficing* search [Simon, 1969], it is difficult to know when to stop looking for a better solution. This

problem is exacerbated in multiagent environments where solutions must be considered from both the local viewpoints of agents and from a global perspective. In Chapter 4, we discuss the criticality of search-termination decisions as an inherent requirement of multiagent search and describe some simple approaches to termination.

1.5 Contributions of the Research

In the last several years, there has been a surge of interest in multiagent systems and a number of researchers have been investigating various aspects of these systems. Major areas of investigation include:

- Ontologies, knowledge representations, and languages (interlingua) that support shared knowledge semantics among agents ([Finin and Wiederhold, 1991, Guha and Lenat, 1990, Neches *et al.*, 1991])
- Integration of heterogeneous databases and knowledge resources ([Finin and Wiederhold, 1991, Wiederhold, 1992])
- Knowledge modularity that will support reusability ([Neches *et al.*, 1991])
- Communication languages and networking protocols to support distributed agents ([Genesereth, 1990])
- Multi-viewpoint, multi-criteria evaluation of solutions ([Sycara, 1987, Werkman, 1992])
- Formal models of cooperation, negotiation, and consensus ([Ephrati and Rosenschein, 1991, Genesereth *et al.*, 1986, Rosenschein and Genesereth, 1985, Zlotkin and Rosenschein, 1990, Zlotkin and Rosenschein, 1991])
- Computational models of cooperation and negotiation ([Adler *et al.*, 1989, Bond, 1989, Cammarata *et al.*, 1983, Conry *et al.*, 1991, Davis and Smith, 1983, Durfee *et al.*, 1987, Klein, 1991, Robinson, 1991, Sathi and Fox, 1989, Sycara, 1985, Sycara, 1988, Werkman, 1992])
- Computer-supported cooperative work in which computational agents interact with human agents ([Lee *et al.*, 1993, Reddy *et al.*, 1993, Sriram *et al.*, 1991])
- Cooperative control and coordination of sophisticated systems ([Cammarata *et al.*, 1983, Corkill and Lesser, 1983, Decker and Lesser, 1992b, Durfee and Montgomery, 1990])
- Distributed constraint processing ([Conry *et al.*, 1991, Fox *et al.*, 1989, Mammen and Lesser, 1992, Sycara *et al.*, 1991, Yokoo *et al.*, 1992])

In this dissertation, we address most of these issues in either a general way or with respect to a specific system. One topic we will not address is that of determining

a shared semantics among agents that have essentially different representations or models of the world. We assume throughout the dissertation that there is a shared communication language known to all agents. Every agent either uses that language internally or can translate local concepts to and from that language. There are several major ongoing projects in shared semantics, languages, and shared use of heterogeneous knowledge bases as noted above. The results from those projects can be integrated with a complete multiagent framework when work in both areas is more mature.

We also do not address the intricacies of adding human agents to a system, although we believe that humans should be in the loop and will be part of systems that move from the research laboratory to real-world acceptance. We recognize the tremendous amount of work that needs to be done in building interfaces that translate to and from comfortable human representations and working style to computationally manageable representations and coordination. In developing systems that support only computational agents, we are laying the foundation for computational communication and representation of information, and synchronization of agents. We see this work as a key ingredient in building consolidated systems in the future.

There are two approaches to understanding multiagent interactions in general: formal mathematical approaches that attempt to develop theories of interaction and cooperation and system-building approaches that attempt to build systems that support real or simulated agent interaction. Unfortunately, in their current states, neither approach is completely satisfactory. Formal approaches work with abstract problems under assumptions that severely limit their applicability. System-building approaches produce working interactions but often with no real understanding of how or why they work the way they do and without any effective method for validating or even evaluating the systems that result. It is our hope that over time these approaches will converge: formal approaches will become richer and less restrictive, and system-building approaches will become more concerned with developing underlying theory and promoting evaluation.

What has been lacking in multiagent work, both formal and informal, is the attempt to build multiagent systems that can handle the richness of detail that comes from using actual reusable agents with heterogeneous knowledge and that can, at the same time, represent and adaptively achieve formal cooperation protocols. The framework that we have developed, while still a prototype, is modular and flexible enough to use formal methods that are developed for various pieces of the problem. For example, if a formal model of negotiation is developed that is rich enough to capture characteristics of a real agent environment, it can be incorporated into our framework

as a distributed-search or conflict-resolution strategy, depending on the purpose of the protocol. We can also incorporate multi-criteria decision making into the local and global evaluations of solutions and can use various types of search-termination decision policies. Local search can use well-known and well-understood search algorithms. As distributed search is investigated more, it may be that better algorithms will be developed that explicitly integrate multiple local searches. The framework we have developed for this dissertation provides the capability to modularly address each of these pieces of the overall problem-solving process. We provide specific algorithms for each piece but can flexibly adapt to the state-of-the-art as it changes and improves.

As a last point, a recent trend in artificial intelligence systems has been to recognize that one algorithm will not be best in every situation and to attempt to choose algorithms that fit the situation well or to concurrently apply multiple algorithms to a problem [Adler *et al.*, 1989, Brodley, 1993, Hogg and Williams, 1993, Lesser *et al.*, 1993, MacLeod and Moll, 1991, Orelup *et al.*, 1988]. This point is particularly germane in reusable-agent systems because each agent has to participate in some globally coherent algorithm but that algorithm cannot be selected until the problem and agent set are known. Therefore, the framework supports the dynamic agent-based selection of appropriate strategies for distributed search based on agent capabilities and intra- and inter-agent attributes and relationships. We will also discuss dynamic selection of conflict-resolution mechanisms for specific conflict situations although we have not yet implemented this capability.

As noted above, there has been a great deal of work done in the field of multiagent systems in recent years. However, for the most part, each project has focused on a small part of the overall problem. In this dissertation, we take an integrated view of the whole problem. We have developed a flexible and general framework that can support different search strategies, cooperation styles among agents, evaluation techniques, and search-termination policies. It can be used to integrate agents with heterogeneous knowledge and evaluation criteria, and with different search and information-assimilation capabilities. The framework enables rapid prototyping of multiagent systems, as evidenced by the development of the **STEAM** and **AGREE** systems. It can support effective problem solving as shown in our comparison of solution quality and problem-solving efficiency between **STEAM** and a similar system not built within the **TEAM** framework as discussed in Chapter 6. The framework does not preclude the use of formally specified techniques for conflict-resolution, solution evaluation, or distributed search, and it can support either flat or hierarchical problem-solving styles. Although we don't go deeply down any of the many possible paths, we provide

the foundation required to deal with realistic multiagent environments in a flexible and general way.

1.5.1 Implementational Achievements

The major implementational achievements that will be presented in the dissertation are:

1. **TEAM**: A testbed framework that supports the requirements for distributed search among cooperative heterogeneous and reusable agents
2. **STEAM**: An application system built within the **TEAM** framework for multiagent cooperative design of steam condensers
3. **AGREE**: An application system built within the **TEAM** framework for developing equitable buy/sell contracts among competing agents

TEAM is a generic and flexible framework that was developed specifically to support distributed search among heterogeneous and reusable agents. **STEAM** is an application program implemented in the **TEAM** framework for parametric design of steam condensers. It comprises seven specialized knowledge-based systems, each responsible either for the creation of a component of the steam condenser or for criticizing some aspect(s) of an emerging solution. The components of a steam condenser are a pump, heat exchanger, motor, platform, shaft, and v-belt. The domain-level knowledge in **STEAM** was originally developed by Meunier and Dixon for use in an *iterative respecification* system [Meunier, 1988]. The implementation of **STEAM** was a major effort that was interleaved with the design and development of the **TEAM** framework.

The **TEAM** framework was originally developed in the context of **STEAM**. We have demonstrated its generality and flexibility by using it as the framework for the **AGREE** system. **AGREE** is a smaller system than **STEAM**, comprising only two agents. It is also a conceptually simpler system and required considerably less domain-level implementation effort. Because **TEAM** was fully developed at the time **AGREE** was created, the development of **AGREE** was very fast with most of the effort going into implementing the agents. The integration effort included defining the common agent language and data structures, defining and implementing the agent structures that link the domain-level knowledge to the framework, and implementing distributed-search operators for the various strategies defined at each agent. Through analysis of the performance (in terms of solution quality and processing time) of **AGREE** under different strategies, we are able to show the dramatic impact of the use of customized distributed-search strategies.

1.5.2 *Experimental Investigation*

Two types of experiments have been performed using the systems described above. The first type validates the feasibility and sophistication of both the complete TEAM framework and particular features of the framework. We demonstrate the feasibility of the framework as an architecture for application systems by directly comparing results from STEAM to those obtained by Meunier [Meunier, 1988] in the original steam-condenser design application in Chapter 6. We then demonstrate the sophistication of the framework by noting specific types of behavior that have been documented in other multiagent frameworks. State-of-the-art cooperative distributed problem solving systems exhibit behaviors, such as distraction and skepticism, that stem from the high degree of local autonomy, intelligence, and interaction of the agents. In TEAM, as in other systems, some of these behaviors are explicitly addressed within the framework and some must be addressed at the level of agent design. As results are presented and analyzed, we will be pointing out instances of cooperative behavior issues that have been documented in the analysis of other frameworks. We view the existence of these instances as a very positive indication of the sophistication of the framework and of the level of realistic interaction that is being achieved by the agents in the application systems.

As noted above, the first type of experiment investigates the feasibility of theoretical and architectural concepts. The second type of experiment that will be presented examines performance characteristics of distributed search, for example, analyzing the effect of customized search in a particular domain. System performance is measured in terms of the quality of solutions (as determined by some global measure of quality such as the cost of a design) and by the efficiency of the system in finding acceptable solutions. Efficiency has three components: 1) the amount of processing time spent by each agent that is directly related to problem solving; 2) the amount of time spent at each agent to support focus-of-attention, communication, and other local overhead processes; and 3) the amount of time spent on framework-level overhead processing in TEAM. Experiments that will be described in following chapters include:

- Comparison of the STEAM system and a previously developed system from which its domain knowledge was taken [Meunier, 1988]. These two systems have identical domain-level knowledge but their agents are organized differently and use different control and coordination.

- Comparison of system performance under the *negotiated search* strategy and a customized search strategy in the AGREE system.

- The influence of an agent's local solution-space characteristics on system performance. In any distributed-search strategy, each agent plays a role (or roles) in the development of a solution. We look at how solution-space characteristics of agents influence their effectiveness in particular roles and identify key characteristics that can be used to predict effectiveness.

- The effect of shared constraining information within the *negotiated-search* strategy. In these experiments, we control whether or not simple boundary constraints are shared by agents and analyze how information sharing affects solution quality and system performance.

1.5.3 Summary of Contributions

The following list provides a compact summary of the contributions of the dissertation as described above:

1. the development of a generic framework, **TEAM**, to support effective integration of reusable-agent systems.

2. the implementation of two prototype application systems (**STEAM** and **AGREE**) within the **TEAM** framework to demonstrate its properties of flexibility and generality.

3. the implementation of a widely applicable, default, distributed-search strategy, *negotiated search*, that demonstrates the representation and application of strategies within the system.

4. the development of a comprehensive conflict-management package, including conflict avoidance and resolution techniques, within the context of *negotiated search*.

5. empirical evaluation of the negotiated-search approach through comparison of solution quality and processing time in the **STEAM** system with a system developed in the Mechanical Engineering Department that contains identical domain-level knowledge but different control-level knowledge.

6. the implementation of a narrowly applicable distributed-search strategy, *linear compromise*, to demonstrate the dynamic strategy selection capabilities of the framework.

7. investigation of the effect of utilizing customized distributed-search strategies in specific situations.

8. investigation of the effectiveness of exchanging and assimilating external information that restricts the local search space of an agent.

9. an analysis of the costs of information sharing within reusable-agent sets.

10. investigation of agents' abilities to undertake different roles within a search strategy and characteristics of agents that influence their effectiveness within a role.

11. discussion and design of solution-acceptability and termination policies within reusable-agent systems.

1.6 Guide to the Dissertation

This section provides pointers to the rest of the dissertation. Chapter 2 contains a survey of related work. (We also point out specific relationships to other work throughout the dissertation in the context of particular features or problems.) Chapter 3 presents a formal view of distributed search and the role of information sharing within the search. Chapter 4 provides details about the philosophy, architecture, and implementation of the TEAM framework.. Chapter 5 introduces *negotiated search*, a specific strategy for coordinating agent interactions in globally cooperative search. Chapters 6 and 7 contain details about the application systems that have been developed, as well as experimental investigation of properties of those systems. Chapter 7 also describes customized distributed-search strategies. Chapter 8 describes experiments on information sharing and assimilation. Chapter 9 discusses agent/role assignments within a strategy and agent set. Chapter 10 summarizes the main points of the dissertation and points the way to future work.

- *Chapter 2:* A survey of related work in cooperative distributed problem solving, multiagent systems, distributed search, negotiation, and conflict resolution.

- *Chapter 3:* A formal perspective on the local and shared information in distributed search.

- *Chapter 4:* A multiagent testbed framework for cooperative distributed search among heterogeneous and reusable agents (TEAM).

- *Chapter 5:* The representation and realization of distributed-search strategies and the introduction of *negotiated search* as a general and widely applicable strategy. An extended example of the application of negotiated search in the STEAM application system.

- *Chapter 6:* Presentation of STEAM, an application program implemented in TEAM for design of steam condensers. Empirical evaluation of STEAM's performance.

- *Chapter 7:* Presentation of AGREE, an application system for buy/sell contract negotiation among two agents. Introduction of customized distributed-search strategies

with a description of an implemented strategy (*linear compromise*) and designs for other strategies. Demonstration of the dynamic strategy-selection capabilities of **TEAM**. Empirical investigation of system performance and solution quality under the linear-compromise strategy as compared to that under negotiated search.

- *Chapter 8:* Experimental investigation of the effect of information sharing and information assimilation in **STEAM** on solution quality and system performance. This includes classification of information that can and cannot be shared, and a breakdown of the costs of information sharing.

- *Chapter 9:* Empirical investigation of agents' abilities to undertake different roles within a search strategy and characteristics of agents that influence their effectiveness within a role.

- *Chapter 10:* A summary of research results and conclusions based on those results as well as a look at the contributions of the dissertation in hindsight. Discussion of future extensions to the current research and possible paths for future exploration.

CHAPTER 2

THE RESEARCH CONTEXT

In this chapter, we survey related research that provides a context for the dissertation. We examine both the historical foundations and the current trends in multiagent problem solving, distributed search, negotiation and conflict management, and the coordination of agents in multiagent systems. The dissertation research is related to many different subareas and can be tied in to many different projects. On the one hand, there is a vast amount of relevant literature to be reviewed. On the other hand, none of the literature addresses the issues described in this dissertation as a comprehensive package of concepts and techniques, and there is no project that can be directly compared to TEAM. With this in mind, we categorize and present the relevant areas of research below.

2.1 Multiagent Systems

In [Bond and Gasser, 1988], Bond and Gasser define research in multiagent systems (MASs) as follows:

> ...(multiagent system) research is concerned with coordinating intelligent behavior among a collection of (possibly pre-existing) autonomous intelligent "agents", how they can coordinate their knowledge, goals, skills, and plans jointly to take action or to solve problems. The agents in a multiagent system may be working toward a single global goal, or toward separate individual goals that interact.

Under this definition, a number of different projects can be classified as related to the development of multiagent systems. Agents can be heterogeneous or homogeneous, reusable or system-specific, locally cooperative or globally cooperative.

An active research area is *hybrid systems*, the integration of systems with different architectures (e.g., neural nets vs. production systems) or algorithms (e.g., linear optimization vs. heuristic search) where each agent implements the most appropriate technology for the class of problems it is designed to handle [Skinner, 1992]. Hybrid systems are typically not considered to be multiagent systems because all the control knowledge about how they should interact is built into the integrating framework

rather than into the agents themselves, that is, the agents' intelligence is limited to domain rather than control knowledge. The range of systems with agents that do exhibit control or coordination intelligence, however, is quite broad. In the rest of this chapter, we categorize issues and approaches in multiagent research and discuss representative projects in each category.

2.2 Managing Knowledge in Multiagent Systems

One of the most fundamental concerns in multiagent systems is how knowledge is maintained, shared, and learned by agents. In this section, we discuss two aspects of knowledge management: knowledge sharing and knowledge consistency.

Knowledge Sharing: When agents interact, they must be able to understand each other, i.e., they must have a mutual understanding of interacting objects and events [Finin and Wiederhold, 1991, Wiederhold, 1992]. In some kinds of multiagent systems, it is possible that all agents will "speak the same language", and understanding each other will be a trivial matter. However, in heterogeneous or reusable-agent systems, shared languages cannot be taken for granted. Languages can be literally different because agents use different names for objects or agents can represent their knowledge such that the semantic meaning of an object in one agent's language may not be equivalent to an object of the same name in another agent's language. This is an important problem in the development of multiagent systems, and one that we do not address in the dissertation research. A major effort is underway to develop techniques to support sharing of knowledge among systems, The Knowledge-Sharing Effort, sponsored by the Air Force Office of Scientific Research, the Defense Advanced Research Projects Agency, the Corporation for National Research Initiatives, and the National Science Foundation [Neches *et al.*, 1991]. As work progresses in this area, it will be possible to use the results from that project to understand how to structure knowledge in particular multiagent systems.

Knowledge Consistency: Another fundamental concern in multiagent systems is how to manage inconsistencies among agents. In [Huhns and Bridgeland, 1991], Huhns describes a precise truth-maintenance approach to distributed database management. We do not intend to imply that we use distributed truth-maintenance techniques or even that it would be possible or desirable to, given the heterogeneity of the agents involved. However, it is useful to examine the heterogeneous-agent concepts of privacy and sharing with respect to those presented by Huhns to provide insight into the underlying philosophy of our work. The terms in italics are used consistently with the definitions provided by Huhns.

In our approach, each agent maintains private databases that contain long-term knowledge about its domain of expertise and short-term knowledge about the current state of problem solving. All local information is initially private, meaning that neither the system, nor any other agent can examine the local databases. As problem solving progresses, information may be communicated with other agents, and that information then becomes *shared*. Long-term knowledge is maintained in a *locally consistent* knowledge base.[1] However, as information is communicated, consistency between local knowledge and received information is not expected.

There are several ways to circumvent the problems associated with inconsistent internal and external knowledge: 1) ignore any externally obtained knowledge that conflicts with internal beliefs; 2) replace internal knowledge with conflicting external knowledge when it is received; 3) allow conflicting internal and external knowledge to co-reside and track what came from where; or 4) always resolve any inconsistency that becomes apparent when information is communicated so that the entire agent set maintains *local-and-shared consistency*. Maintaining local-and-shared-consistency implies that any time an inconsistency is found, some decision will be made as to which information is correct and that decision will be propagated through all agents that have shared the knowledge. *Global consistency*, on the other hand, implies that all knowledge is consistent across all knowledge bases, whether or not the agents interact.

With reusable, heterogeneous agents, we claim that global consistency is neither feasible nor desirable. From a feasibility standpoint, it is impossible to knowledge engineer reusable agents in a large application system that are created independently by different people at different times with different expertise. Furthermore, global consistency requires that different perspectives and beliefs cannot co-exist within an agent set. However, as we have previously discussed, different perspectives can actually be quite beneficial, particularly in creative problem-solving situations.

The question then becomes whether or not local-and-shared consistency is achievable and desirable in this type of system. To maintain this level of consistency, it must be the case that every disagreement is fundamentally resolved when it is noticed and the effects of that resolution are propagated throughout the set of participating agents. To do this would be a very large burden, both computationally and philosophically. Computationally, we would want to try to resolve the conflict based on some concrete evidence that one or the other perspective is more valid: perhaps by invoking some deep model of the domain, applying naive physics, maintaining measures of certainty of each agent for a particular piece of information, or finding collaborating evidence.

[1]We use this term in a nontechnical sense to mean simply that there are no explicitly known conflicts in the declarative knowledge contained in the database.

Although there are certainly situations in which the validity of some agent's belief should be tested, it can be computationally intensive to do so. Furthermore, in many situations, there is no absolute measure of correctness. And again, philosophically, it is often not desirable to enforce consistency among agents. It is not necessary that all conflicts be resolved in order to find a mutually acceptable solution.

For the reasons given above, we have decided against enforcing any type of consistency across the borders of agents. Since conflicting information is not expected to be resolved, when an agent is aware of an explicit inconsistency it must select one viewpoint to apply in its reasoning. Selecting which information to use is often non-trivial and we treat it as agent- and strategy-specific.

2.3 Distributed Search

Although many systems have been built that do some form of distributed search, in many of these systems, the search algorithm is implicit in the architecture and control paradigm. In this section we describe work in which the search process is itself a focus of the research. In all of the distributed-search research described in this section, agents have homogeneous local knowledge representations and tightly integrated system-wide problem-solving strategies that are assumed to hold across all agents [Conry *et al.*, 1991, Fox *et al.*, 1989, Mammen and Lesser, 1992, Nishibe *et al.*, 1992, Sycara *et al.*, 1991, Yokoo *et al.*, 1992].

In [Yokoo *et al.*, 1992], Yokoo looks at developing formal representations for various categories of distributed-search algorithms for constraint-directed search: distributed constraint satisfaction, distributed constraint optimization, distributed constraint optimization with local state-space search, and distributed constraint optimization with local and/or-graph search. For each category, he describes associated research issues, current work that provides at least some initial suggestions on how to deal with those issues, and classes of problems that fall into the search category. In [Nishibe *et al.*, 1992], Nishibe looks at reducing the often intractable computational cost of complete distributed-search algorithms for constraint-satisfaction problems. He particularly investigates developing satisficing algorithms that take advantage of heuristic orderings of variable/value instantiations. Although both of these projects are in a preliminary stage, the work shows promise of leading to classifications of problems and associated mappings from problem classes to effective algorithms for those classes.

Distributed-search systems have typically used variants of backtrack search. In backtracking, a solution is incrementally built by instantiating one variable or attribute of the solution at a time. When an attribute cannot be assigned a value

because of conflicts with previously instantiated attributes, one or more of the previous attributes are undone and search continues from that point. Several mechanisms for determining the order of attribute instantiation (variable ordering), the order in which potential values are assigned to attributes (value ordering), and the order of agent invocation in distributed systems have been explored in [Sycara *et al.*, 1991, Yokoo *et al.*, 1992].

A heuristic satisficing algorithm can only be as good as the heuristics that are used. Currently, most heuristics are developed intuitively—someone decides that it seems reasonable to apply a particular rule to a particular situation. A more rigorous approach to heuristic development would provide both more robust systems and greater user confidence. For example, Mammen [Mammen and Lesser, 1992] describes heuristics for improving problem-solving efficiency in multi-agent systems based on the use of the variable tightness texture measure developed by Fox et. al. for single-agent systems [Fox *et al.*, 1989]. Mammen extends this texture measure to distributed systems by defining a new measure, *Imbalance in Variable Tightness*: "the extent to which some agents have variables with higher or lower variable tightness measures on average." Because the measure is domain-independent and well-defined, the effect of imbalance in variable tightness can be rigorously explored along with heuristics for improving performance.

Centralized Constraint-Directed Reasoning: Fox [Fox, 1987, Fox *et al.*, 1989] investigates constraint-directed reasoning in the job-shop scheduling domain. The representation of constraints in Fox's work is more complex than that in `TEAM` and the use of constraints is more integral to the operation of the system. Conflicts occur among constraints because of interactions such as resource/due-date, resource/cost, due-date/cost, and resource/resource. The use of constraints in the `ISIS` system developed by Fox and in other systems using the constraint-directed reasoning paradigm is highly sophisticated, in fact, it is sophisticated to the point of making it unsuitable for a reusable-agent system. Constraints are selected, relaxed, and applied through the interpretation of numeric weights on *a priori* and relative importance, sensitivity, and utility. Problem-solving occurs through a tightly integrated hierarchical-reasoning search paradigm that depends on required constraining information being available at every level.

It follows that constraint-directed reasoning places stringent requirements on the similarity and global understanding of knowledge representations, architectures, and search paradigms that must exist among agents. These requirements make the formalism too limited for use in systems of heterogeneous and reusable agents. We

note though that these limitations may be completely reasonable in the context of the application domain of job-shop scheduling.

Distributed Constrained Heuristic Search: In [Sycara *et al.*, 1991], Sycara, Roth, Sadeh, and Fox describe research in applying constraint-directed reasoning to systems of asynchronous agents. This work is also done in the domain of job-shop scheduling, where multiple orders comprising multiple activities, possibly using multiple resources, must be scheduled over the existing set of resources over a constrained time interval. Agents are homogeneous units where each agent has a set of orders to schedule. Scheduling is done asynchronously and concurrently. A heuristic texture measure for variable ordering, variable tightness, is used to focus the search. Agents interact by contributing local demand profiles for the resources that are to be allocated. A monitoring agent for each resource computes and distributes the aggregate demand for the resource. The aggregate demand is then used by variable and value ordering heuristics to select an activity to schedule and determine a tentative resource reservation for that activity.

The paradigm used to control the search in this work is a type of dependency-directed backtracking called *Distributed Asynchronous Backjumping* (DAB). The DAB paradigm takes into account the effect of agent asynchrony by recognizing that other agents can make intervening resource reservations that invalidate current local reservations. The results obtained from the use of the DAB paradigm show that in situations where texture measures are updated and propagated frequently, system efficiency approaches that of centralized search. However, solution quality is not a focus of this work and there is no mechanism for using quality to guide search: a solution is either acceptable or infeasible with no gradation. We believe that although the DAB paradigm offers an effective search mechanism from a system-efficiency perspective, some combination of DAB and a more comprehensive evaluation/constraint-optimization package would be required to handle situations where quality optimization is also important.

We should note that the use of a search strategy such as DAB is not inherently incompatible with the TEAM framework and that it would be possible to implement the strategy. However, with heterogeneous or reusable-agent systems, there is some question as to whether the agents could support such tightly integrated texture measures. These measures require the calculation, exchange, and assimilation of a highly specific abstraction of the local state spaces of agents and it is unlikely that general reusable agents would implement them. However, the idea of communicating abstract information that will guide other agents' local searches is extremely important in developing effective distributed search. Agents may be able to communicate some

information about either their search spaces or their search strategies that can be used by other agents to guide and coordinate their local searches. The more an agent understands the search space of another, the less time is wasted in following unproductive paths, and the more an agent understands about how another agent is intending to search, the more coherent their mutual behavior becomes. The knowledge gained from information exchange can also be used to guide the relaxation of local solution requirements in a productive way when the search space is overconstrained. Therefore, to whatever extent communication of constraining information is possible, it should be exploited (see Chapter 8).

The Distributed Vehicle Monitoring Testbed: Rather than using backtracking to examine alternative solutions, the Distributed Vehicle Monitoring Testbed (DVMT) used an opportunistic incremental-extension algorithm in which multiple potential solution paths were simultaneously [Lesser and Corkill, 1983]. When an a value couldn't be assigned to an attribute because of conflicts, instead of undoing some attribute(s) of that solution, a new solution was started. This allowed for the possibility that the existing solution, though not completely satisfying all agents' requirements, might eventually be recognized as better than any other solution found. This type of problem solving is a forerunner to the TEAM approach which also simultaneously maintains multiple solution paths. However, when DVMT started a new solution path, it did not take into account any feedback information about what was wrong with the old path. The agent that developed the new solution simply tried its next best possible partial solution. In contrast, with the TEAM approach, feedback information is used to refine the agent's view of the mutual search space before a new solution is proposed.

2.4 *Negotiation and Conflict Management*

In [Gasser, 1991], Gasser states that the term *negotiation* has been used in dozens of different ways in the DAI literature. Several definitions of negotiation from various researchers, within and without the field of DAI, are provided below:

> Negotiation is a process of communication established between two conflicting agents in which they try to develop or refine their plans jointly so that the goals of each are satisfied. [Adler *et al.*, 1989]

> ...often negotiation is optional. Other methods are available for dealing with the divergence of interest including efforts to influence the other by overt moves (e.g., launching a military attack), taking the other to court, etc. In contrast to such struggles, negotiation can be viewed as a form of coordination, in which the two parties talk to one another in search of an agreement. [Pruitt, 1981]

> Negotiation is composed of two phases: a *communication* phase where information relevant to the negotiation is communicated to participating agents, and a *bargaining* phase where "deals" are made between individuals or in a group... The negotiation process can be viewed as constraint directed search in a problem space. [Sathi and Fox, 1989]

One of the primary motivations for the dissertation research was to examine how conflict management and negotiation could be used effectively in a multiagent search system. A major challenge we faced in applying negotiation, however, was that the term negotiation is used in many different situations and is difficult to pin down [Lander and Lesser, 1993]. Artificial intelligence researchers have previously used the term negotiation with respect to conflict resolution and avoidance [Adler *et al.*, 1989, Cammarata *et al.*, 1983, Klein, 1991, Lander *et al.*, 1991b, Sycara, 1988, Werkman, 1992], task allocation [Davis and Smith, 1983, Durfee and Montgomery, 1990, Zlotkin and Rosenschein, 1989], and resource allocation [Adler *et al.*, 1989, Conry *et al.*, 1991, Sathi and Fox, 1989, Sycara *et al.*, 1991]. Negotiation has been examined under a game-theoretic approach in which every agent knows all relevant information about other agents [Genesereth *et al.*, 1986, Kraus and Wilkenfeld, 1990, Zlotkin and Rosenschein, 1991] and under conditions where agents are hostile and completely unwilling to share private information [Sycara, 1988]. Various points along the cooperation/hostility continuum are examined in [Zlotkin and Rosenschein, 1990, Zlotkin and Rosenschein, 1991]. Negotiation can occur among peers [Cammarata *et al.*, 1983, Lander and Lesser, 1991], through a mediator or arbitrator [Sycara, 1988, Werkman, 1992], or hierarchically through an organization [Davis and Smith, 1983, Durfee and Montgomery, 1990]. It can occur at the level of domain problem-solving or at the level of control problem-solving.

Laasri et. al. describe the *recursive negotiation model*, a general model of multi-agent problem solving that details various situations that can potentially benefit from negotiation [Laasri *et al.*, 1992]. In examining this model, it becomes clear that negotiation is a pervasive process and remains relatively untapped by current computational systems. For example, negotiation could be used to select a group of agents that would then negotiate over how to allocate a set of tasks among a possibly different set of agents. Those agents would in turn execute the tasks and negotiate a mutually acceptable solution. In all of these situations, negotiation is used to either generate potential solutions even though multiple agents' requirements on the selection may overconstrain the search space or to select one of a set of possible solutions in the face of potentially conflicting local evaluations. In the following sections, we look at negotiation and conflict management in several different categories: human models, formal models, and computational models.

2.4.1 Human Models of Conflict Management

Psychological Studies: Pruitt discusses human negotiation and conflict management in international relations from a background of organizational and occupational psychology [Pruitt, 1981]. Fisher and Ury describe tactics to be used for reaching agreement in day-to-day experiences [Fisher and Ury, 1981]. These authors describe various techniques used by expert negotiators that can be applied to human bargaining situations. Although there are many useful insights to be gained, much of the work in human negotiation deals with psychological issues such as how to make concessions without losing power or 'face'. Many of the psychological factors are not immediately relevant to computational negotiation and we do not take them into account. However, some of the tactics described in these works can be carried over to computational environments. In [Sathi and Fox, 1989], Sathi and Fox describe the implementation of several negotiation operators that were directly inspired by negotiation tactics suggested by Pruitt. For example, unlinking is a negotiation tactic in which a bundled set of goals of a negotiation participant is decomposed into smaller components which can then be addressed according to the priority of the components rather than the priority of the whole.

Lateral Thinking: Although work on lateral thinking is not directly concerned with conflict resolution, one approach to resolving conflicts is to search for alternative solutions. DeBono defines *lateral thinking* as a mechanism for generating innovative solutions in human problem solving [de Bono, 1971]. In lateral thinking, different approaches to a problem are explicitly sought through strategies which are designed to elicit unusual connections between problem components. Although we have not pursued the use of lateral-thinking techniques for computational conflict resolution, it is an area that may offer interesting insights into how to construct mutually acceptable solutions in some situations. In [Sathi and Fox, 1989], Sathi describes a *reconfiguration* operator for resolving conflicts that changes the physical configuration of a resource in order to adapt the resource to more closely match a request. This is a simple example of the type of creative problem solving proposed by DeBono.

2.4.2 Formal Models of Negotiation and Conflict Management

Several formal models of negotiation have been developed [Ephrati and Rosenschein, 1992, Genesereth *et al.*, 1986, Khedro and Genesereth, 1993, Kraus and Wilkenfeld, 1990, Rosenschein and Genesereth, 1985, Zlotkin and Rosenschein, 1989, Zlotkin and Rosenschein, 1990]. These models have some highly desirable properties such as the ability to guarantee convergence, pareto-optimality, or equilibrium (where

no agent can benefit from lying). For example, Zlotkin and Rosenschein describe a formal model of negotiation in [Zlotkin and Rosenschein, 1989] that can be used to guarantee the selection of pareto-optimal, utility-maximizing solutions from a set of potential solutions. However, a drawback in the use of the formal models has been that they are computationally intractable in many situations and they often fail to capture the richness of detail that would be necessary to apply the model in a realistic domain. In this section we describe several models that have been developed.

In much of the work on formal models of negotiation, negotiation is viewed as a separate process used to select a solution from a set of candidate solutions. Each agent involved in the negotiation attempts to select the candidate that will maximize local or global utility with the understanding that all agents must ultimately agree on a single candidate. The negotiation process does not itself define the set of candidate solutions. It is usually assumed that only two agents are involved in the negotiation, that agents are *rational*, meaning that they will not select an action that will result in an avoidably poor payoff, and that each agent knows the other agent's potential payoffs for all candidate solutions.

Cooperation without Communication: Genesereth, Ginsberg, and Rosenschein argue that with rational agents in restricted domains, no communication is needed to achieve cooperative behavior [Genesereth *et al.*, 1986]. Instead agents maintain local copies of a "payoff matrix" for the problem that defines the payoff each agent will receive for every possible outcome of the problem. Using this payoff matrix, an agent can determine what action it should take to get its best payoff, assuming that other agents also know the matrix and will act rationally. Although this research isn't cast as a model of negotiation, it does point out a limited situation where negotiation is implicit in the problem-solving model. In other words, in a two-agent situation where rational agents have to mutually select a solution and where both agents know all possible solutions and the payoffs to both agents for each possible solution, negotiation can take place with no communication. This is obviously a highly restricted case but does present some initial insights into what information is required for negotiating the mutual selection of acceptable solutions and where the information requirements are likely to break down.

Negotiation to Select Pareto-Optimal Solutions: Zlotkin and Rosenschein describe a negotiation paradigm in [Zlotkin and Rosenschein, 1989] that can be used to guarantee the selection of pareto-optimal solutions from a set of locally evaluated potential solutions. The paradigm is motivated with "The Postmen Problem" in which two agents have to deliver letters to addresses on a city map. Each has a set of letters to be delivered and they are able to exchange letters. By exchanging letters

they may be able to improve their plight by lessening the distance they need to walk to deliver their set of letters. In fact, each agent wants to find the shortest distance route through a graph of n addresses where each address is the repository for one or more letters, a problem very similar to the classical np-hard traveling salesman problem, for each possible assignment of letters to postmen. Assuming that it is possible to calculate optimal solutions for all possible assignments (where cost=distance), they are ready to begin negotiating a pareto-optimal solution. From a practical standpoint however, guaranteed optimality of local solutions is unlikely to be achievable, which would obviate the possibility of global pareto-optimality.

Although this particular problem is not a good motivator, there are situations in which it is possible to generate and evaluate the complete candidate set and, in this situation, an algorithm such as the one proposed by Zlotkin and Rosenschein can be used to make the best selection possible. However, in many domains, such as design, the complete candidate set is not known and cannot be generated. Furthermore, in globally cooperative systems, the evaluation of a candidate solution is not strictly local, but has both local and global elements. In this case, the problem is not to maximize local utilities over a set of solutions, but rather to generate solutions that maximize both local and global utilities under conditions of uncertainty and incomplete information.

Constrained Intelligent Action: Ephrati and Rosenschein [Ephrati and Rosenschein, 1992] investigate a planning domain in which a subordinate agent attempts to achieve some plan of a supervisor agent. The supervisor has incomplete information about the world as seen by the subordinate and therefore may not be able to provide a complete and reasonable plan. The subordinate must use its own perspective on the world to choose a plan that both satisfies the supervisor's goals and matches the supervisor's expectations of cost. Some of the measures suggested by Ephrati for choosing a plan are: 1) comparing the cost of plans generated by the subordinate to the cost of the supervisor's plan; and 2) comparing the deviation in subordinate and supervisor knowledge bases as a result of the plan; 3) avoiding irreversible consequences; and 4) applying distance metrics to plans by measuring distance between each step in a plan. These measures attempt to provide a foundation for choosing among alternative plans when there is inconsistency between the world models of the supervisor and subordinate and when the goal is to satisfy the supervisor.

Although the supervisor/subordinate relationship of that model is not directly relevant to negotiation, in the TEAM framework, agents are often required to accept externally imposed limitations. Using distance metrics to compare local to external solutions in order to maximize the probability of external acceptability is an area of

open research that, in certain situations, could be used as a conflict management technique. This is particularly true if generating alternatives is not expensive. However, when solutions are expensive to generate, mechanisms that lead to efficient pruning of the search space (such as assimilating information about the local search spaces of other agents) are preferable to mechanisms that allow for detailed comparison of generated alternatives.

Non-Cooperative Models: In [Zlotkin and Rosenschein, 1991], Zlotkin and Rosenschein present the Unified Negotiation Protocol (UNP) for agents in non-cooperative domains: domains in which there may not be any benefit to cooperation. They restrict their model to two-agent situations where the agents are rational and have complete knowledge and symmetric abilities. The protocol is presented within a closed-world, slotted-blocks domain with fixed goals. Three different types of situations are defined: 1) cooperative, in which the cost to both agents of a joint plan that achieves all goals is less than or equal to any individual plan that would achieve a single agent's goals; 2) compromise, in which at least one agent will have to pay more to achieve its goals jointly than it would to achieve them individually but, given the inevitable presence of another agent, a deal can be made that achieves the goals; and 3) conflict, in which at least one agent has to pay an unacceptably high cost for any joint plan and, therefore, no deal can be made.

Zlotkin and Rosenschein show that in some conflict situations it is possible to find a partial plan that will increase both agents' relative payoffs even when all goals cannot be satisfied. The UNP is presented as a mechanism that describes the characteristics of a deal and the meaning of utility. A hierarchy of deal types is shown based on the types of situations these deals are effective for. Finally, multiplan deals which can be used in flexible goal situations are briefly discussed. This work provides initial insights into the process of making deals in locally cooperative domains where mutual benefit is not a given. The integration of formal models such as this one with computationally practical algorithms, representations, and architectures may eventually result in sound and effective conflict management that combines the strengths of formal modeling with the demands of practicality.

2.4.3 Computational Approaches to Conflict Management

Open Systems: Early work in this area is presented in a paper by Kornfeld and Hewitt that describes their exploration of the scientific community metaphor [Kornfeld and Hewitt, 1981]. In this paper, they examine issues of parallelism, pluralism, and concurrency among diverse agents. A language, Ether, was developed to support the type of problem solving embodied in the scientific community metaphor.

This line of investigation has been continued by Hewitt in his research on *open systems* [Hewitt, 1986, Hewitt, 1991]. An open system comprises a set of agents that concurrently interact with the world and that have no global perspective, control, or knowledge consistency. Agents embody "microtheories", small chunks of locally consistent knowledge, and interact through negotiation over mutually inconsistent microtheories.

The HEARSAY-II Speech Understanding System: The HEARSAY-II system provides an early model of multiple specialists working together [Erman *et al.*, 1980]. Cooperation among the specialists occurs implicitly through the incremental extension of globally available hypotheses. Conflicts are not resolved explicitly—instead competing hypotheses coexist and vie for processing resources to improve their believability.

The expertise of a traditional blackboard system as exemplified by Hearsay-II is represented by KSs which, in some ways, parallel the knowledge-based agents of the TEAM framework. Traditionally, KSs have some or all of the following attributes:

1. they are instantiated in response to a particular pattern on the blackboard;
2. they respond only to information that is available at the time of their instantiation;
3. they cannot be suspended during execution or reinstantiated once execution has ended;
4. they keep no history of their actions.

TEAM agents, on the other hand, are fully functional systems which have the property of persistence throughout the problem-solving process. They can generate partial solutions independently and have private data, knowledge, goals, and histories of their own actions. They can suspend work on a particular task at any time and resume it when the situation seems more suitable. Also, HEARSAY-II had a centralized scheduler which made control decisions based on the global situation represented on the blackboard. Each TEAM agent is responsible for scheduling its own tasks based on both its internal problem-solving state and the information available through the global database.

CALLISTO: An Intelligent Project Management System: The CALLISTO system by Sathi, Morton, and Roth is concerned with the management of large projects that require resources and expertise from diverse fields [Sathi *et al.*, 1986]. The goal of the project is to model a good manager and the tasks that she is involved in: specifying activities, organizing personnel, scheduling activities, chronicling activities, projecting resource needs, monitoring project performance, and analyzing and managing

deviations from plans. In the first phase of research, two models of project expertise emerged: constraint-directed negotiation and comparative analysis of project knowledge. Subsequently the mini-Callisto system was developed to pursue investigation of constraint-directed negotiation. The negotiation process is seen as detecting an inconsistency or incompleteness, isolating appropriate constraints, communicating constraints to appropriate parties, and jointly relaxing or strengthening constraints for resolution.

Agents in this system are logically and physically distributed. Each mini-callisto represents a specific organizational entity (a group or department). Each has specialized expertise and local knowledge bases and is motivated by local goals. Cooperation is enforced through explicit contracts. A testbed was developed that supported the creation of several mini-callistos which could negotiate management problems. The testbed was used for experiments on negotiation for resource, activity, and configuration management. This work was fairly early in terms of conflict management. It is significant in that it identified problems that would become major themes in later work: for example, how to detect inconsistency or incompleteness in a network, how to identify the parties involved in a negotiation, how to isolate appropriate constraints, how to relax constraints, and how to agree that a mutually acceptable solution has been found.

Telephone-Network Traffic Control: Adler, Davis, Weihmayer, and Worrest examine conflict management in the context of telephone-network traffic control in multiagent systems[Adler *et al.*, 1989]. Agents are autonomous, geographically distributed, homogeneous and designed to work together. Each agent controls the routing of telephone calls through a network of linked nodes in a bounded geographical area. Agents interact when calls are routed through a link that connects nodes across the geographical boundaries of multiple agents. The problem addressed is for each agent to plan routes for calls originating in or entering its local area in a simulated system. When a particular link becomes overloaded, traffic must be diverted to other local links or to links that end in some other agent's geographical area. Conflicts occur when locally developed plans are merged and jointly place too much traffic on particular links.

One of the goals expressed in this work was to investigate the dynamic selection of conflict-management techniques based on characteristics of the current state of problem-solving. Several protocols are described that vary in the amount and type of information is shared among agents. The first, conflict-driven plan merging, is a backtracking resolution protocol that can be invoked in response to detected conflicts in plans. Others are shared plan development, a conflict-avoidance strategy that

uses hierarchical plan expansion, and mutual accommodation, a resolution strategy in which agents iteratively exchange plans and then attempt to develop local plans that will not conflict with the communicated ones.

A hypothesis of the research is that the latitude available to agents in taking actions that meet their local performance criteria, along with the complexity of the problem (roughly the number of interacting constraints to be dealt with), and the costs of applying a specific interaction protocol determine which protocol should be used. This idea is a precursor to the dissertation research in which we look at the dynamic selection of customized distributed-search strategies in response to agent properties and inter-agent relationships. It is interesting to note that the need for dynamic selection of coordination strategies has been recognized for some time. TEAM is the first architecture that addresses that need in a flexible and generic manner.

Cooperative Design: Klein has developed a conceptual model of conflict resolution in design and has shown that the model is viable through its implementation [Klein and Lu, 1989, Klein, 1991]. The model is based on informal studies of cooperative design performed by human design experts. It has been applied to solar home design and local area network (LAN) design.

Klein maintains that conflict-resolution (CR) expertise exists separately from domain-level design expertise. He suggests a taxonomy of design conflict classes that range from very general to highly specific. Associated with each conflict class is advice for resolving that type of conflict. A conflict-resolution strategy is defined as the mapping from conflict class to associated resolution advice. A strategy's preconditions hold when the conflict class has occurred in an emerging solution. Klein suggests that CR expertise includes control knowledge for selecting a CR strategy to apply from the set of strategies that may have their preconditions satisfied at any given time. He goes on to argue that conflict-resolution expertise is abstract and applies over a variety of design domains.

The CR model has been implemented as the Cooperative Design Engine (CDE). In this system, multiple computational design agents attempt to solve a design problem. An agent comprises two components: a design component that can update and critique designs and a CR component that resolves agent conflicts. The CR components of all agents are identical. An agent recognizes a conflict through constraint-violation detection, and identifies possible class(es) of the conflict. Advice associated with each conflict class is instantiated into plans for resolving the conflict. A single plan is selected and executed that may resolve the conflict or that may lead to new conflicts that must be resolved.

Klein's work both demonstrates the ubiquity of conflict and the need for a generic and flexible approach to conflict resolution. The CDE system substantiates the

proposed model by demonstrating that the approach is useful and productive in resolving conflicts that occur naturally in design problems. However, the CDE system is very exploratory in nature and does not address issues that arise from embedding conflict resolution in a larger problem-solving context. For example, solution evaluation, information exchange and assimilation, system termination, focusing, efficiency, coordination, and coherence are all integral parts of effective multiagent problem solving that are not examined in Klein's research.

Our approach, like that used by Klein, recognizes that many different conflict-resolution methods exist and that specific methods are applicable in specific situations. We believe that his conflict-class taxonomy and associated resolution advice provide a general foundation for handling conflict in design application systems. The TEAM framework represents an initial attempt to merge the system qualities required for effective conflict management such as that proposed by Klein with effective coordination and control. TEAM embeds the area of conflict resolution into the more general topic of search. We look at how conflict resolution is entwined with other aspects of the search process such as solution evaluation and system termination. We describe how to handle conflict when agents are reusable, heterogeneous, and don't all have the same level of CR expertise.

Multistage Negotiation: Conry et. al. present multistage negotiation as a coordination protocol for distributed search [Conry *et al.*, 1991]. The application domain is the generation of plans to satisfy service-restoral goals in the control of a communications network. Homogeneous agents control circuits within a particular region in the network. When a circuit is interrupted, planning to restore service is initiated locally by agents involved. Initial plans consist of local plan fragments that represent alternative routes through their own region and external requests for routes through other regions. Agents have no global view of the emerging solution and information is shared by explicit communication over a limited bandwidth. Despite the limitations on communication, agents must be able to recognize when a problem is overconstrained so that they can satisfy at least some subset of goals.

Negotiation, in this research, is viewed as an iterative exchange of information with the purpose of acquiring enough knowledge about plan interactions to make local decisions that will not conflict with the decisions of other agents in the network. Agents make tentative commitments to particular plans and announce their intentions to neighboring agents. When conflicts are detected, agents change their tentative commitments. When there are no alternatives that do not result in conflict, they must choose some subset of goals to satisfy.

The protocol is structured in three phases: asynchronous search, coordinated search, and an overconstrained resolution phase. In the asynchronous search phase,

agents work independently to select pre-enumerated local solutions (plan fragments) that will satisfy their goals. They then propagate these local solutions to their neighbors. When a conflict is found between a local solution and that of another agent, any tentative resource commitments relative to that solution are retracted and another solution is tried. If all of the prespecified local solutions have been found to conflict with the requirements of other agents, the network enters a coordinated search phase. In this phase, agents search for solutions within a context: in other words, agents keep track of which local solutions can be used in conjunction with which non-local solutions in order to determine if there is any combination of solutions that will satisfy all goals. In the previous phase, commitments and retractions were not coordinated, leaving open the possibility that a proposed solution that failed in one context would have worked if it had been tried earlier or later. If no composite solution is found in this phase, the system enters the overconstrained resolution phase to determine which goals must be relinquished.

This protocol is interesting in that it presents a mechanism for the propagation of abstracted context information throughout a physically distributed network. It guarantees a solution will be found if there is a plan that can satisfy all global goals and also guarantees to detect an overconstrained situation if one exists. However, the use of this protocol is limited to situations where all plans and associated plan fragments can be enumerated before its application.

Constraint-Directed Resource Reallocation: Sathi and Fox [Sathi and Fox, 1989] describe constraint-directed negotiation in the domain of resource reallocation in an engineering organization. In this domain, resources are bought and sold among agents in response to changing needs. The exchange of resources can be a *simple transaction* in which a resource is simply sold from one agent to another, a *trade* in which a two-way exchange of resources occurs between agents, or a *cascade* in which a series of transactions are required to fulfill some goal. Negotiation techniques in this work are motivated by negotiation operators developed in the realm of human relations. The three operators used are composition, reconfiguration, and relaxation. This is a locally cooperative domain, meaning that agents are basically competitive but are willing to look for reallocation arrangements that are mutually acceptable.

Sathi evaluates alternative allocations in a two-stage process. In the first stage, the local evaluations of agents are computed for a set of potential alternative reallocations. The utility of each constraint on each possible transaction or cascade is derived and placed in the matrix. The set of constraint utilities for each transaction are combined into an aggregate utility for each transaction/cascade using combinatorial strategies such as *lexicographic semiorder*. This aggregate utility summarizes the

agent's evaluation of a particular transaction. Once agent utility and ranking has been computed, a similar combination procedure is used to derive the global utility and ranking of alternative reallocations. Sathi's work is significant in his recognition of the need for two-level solution evaluation. The particular evaluation procedure he uses is not applicable in a heterogeneous-agent system, however, because it relies on tightly coordinated evaluation.

Several interesting concepts came out of this work. One is its *reconfiguration* operator. Applying this operator changes some attribute of an existing resource in order to meet requirements, for example, reconfiguring a workstation by changing its software or its physical location. This leads to opportunities for true creativity given an appropriate understanding the properties of objects.

A second observation made by Sathi was that negotiators use different strategies for converging to a solution depending on the search space and the topology of the constraint space. This is the same theme we will see in many different research projects, and that is addressed by the dynamic strategy capabilities of TEAM.

Finally, a third interesting observation made by Sathi was that system performance improved when a "mediator" was added. For practical purposes, this means that what started out as a fully distributed system was reimplemented as partially centralized, and performance improved. This is not a surprising result really since distribution of data and problem solving introduces many difficulties in search. In reusable-agent systems, agents are truly heterogeneous in their knowledge and it is not possible to build a central agent that knows how to solve all the subproblems. However, Sathi's observations suggest that distribution should be applied sparingly, and only when the need truly outweighs the cost.

2.4.4 Negotiation and Feedback

In Chapter 1, we describe two types of information that are communicated by agents during negotiation: proposals and feedback. Feedback information can take different forms. When an agent receives a proposal to evaluate, should it just indicate "yes, I like it" or "no, I don't like it". If it is possible to say something more than that about why a proposal is unacceptable, how much should be said? For example, suppose that an agent is scheduling a meeting and proposes a meeting time of 4:00. One of the meeting participants rejects that proposal because it conflicts with another meeting. Should the rejecting agent tell the scheduler that its other meeting will be over at 4:05? Should it mention that the other meeting is not very important? Should it tell the scheduler all the available time slots it has for that day? for the next week? Should it mention that every Wednesday from 3:00-5:00 is reserved for

tennis? Sen is attempting to formally investigate meeting scheduling [Sen and Durfee, 1992]. He assumes that the cost of formulating a list of available time slots is very low compared to the cost of multiple communications. Under this assumption, it is cost-effective to reply to a rejected proposal with a list of other possible times (alternative solutions). Although this assumption is perfectly reasonable for the task he is examining, it is inappropriate for some domains. For example, in the STEAM mechanical design domain that will be described in Chapter 6, the analogy to generating a list of available times would be generating a list of potential component proposals. However, a single component proposal is expensive to generate. Therefore, generating a list of alternative potential components should be a low-preference policy for providing feedback. Clearly, the decision about what feedback to communicate depends to some extent on the domain.

Feedback enables agents to converge on a shared understanding of each other's requirements, which in turn enables them to intelligently form counter-proposals. However, in the world of heterogeneous reusable agents, there is no guarantee that every agent is capable of formulating appropriate feedback or that every agent is capable of assimilating the feedback it receives. Depending on the domain and on characteristics of the shared search space, e.g., density of solutions, these capabilities may or may not be critical to finding acceptable solutions. In a dense space, even random generation of proposals may be a reasonable approach. We will discuss this further in Chapter 9. Knowledge about feedback capabilities can be very important in deciding whether or not to include an agent in an agent set and/or how to organize the set. To achieve the flexibility required for reusable agent sets, the framework must be able to support different policies for information exchange such as the *yes-no* and *alternatives* policies described by Sen [Sen and Durfee, 1992] or the *generate-random-alternatives* policy described by Lander [Lander *et al.*, 1991a].

2.4.5 Mediation and Arbitration

In many human and some computational conflict-resolution systems, third-party intervention is used to help resolve conflicts among agents [Pruitt, 1981, Sycara, 1988, Werkman, 1992]. In mediation, the third party works with the disputants, helping them to reach agreement. In arbitration, the third party listens to arguments from both sides and produces a binding resolution. In either case, the third party is expected to gather information, evaluate the information impartially, break deadlocks, and protect the confidentiality of the conflict participants. When conflict participants are hostile, it is often valuable to have a buffering agent. A mediator or arbitrator can collect information from the negotiation participants without making the information

public, provide an objective perspective, suggest proposals, and possibly enforce relaxations when agents will not relax voluntarily. A mediated approach was taken by Sycara in her work on management/labor negotiations [Sycara, 1988]. Werkman uses both mediation and arbitration in his DFI system [Werkman, 1992]. These systems are both described below.

Although mediation and arbitration are useful mechanisms in domains where the participants in a conflict are adversaries, a general problem with third-party intervention is that the third party does not have deep domain knowledge and cannot be expected to make technically expert decisions. This is especially true when the conflict participants are logically heterogeneous. Because the systems that will be addressed in this dissertation are cooperative, the role of mediator is essentially redundant. The conflict participants themselves are capable of gathering and evaluating information and have the technical expertise to do so intelligently. Deadlocks only occur when agents refuse to move from a favorable position to something less individually favorable. In a cooperative situation, the focus is on finding the most favorable common, rather than local, solution. A deadlock is never a favorable common solution and, therefore, participating agents are willing to change their positions as required. The mechanisms for achieving this in an equitable way will be described in detail in later chapters.

Negotiation to Resolve Goal Conflicts: Negotiation to resolve goal conflicts was explored by Sycara [Sycara, 1987, Sycara, 1988]. In this work, the application domain is labor mediation. Sycara's system, PERSUADER, acts as mediator between a company and a union. This is a locally cooperative domain since solution quality is a function of the utilities of independent agents. Agents are adversary, local goals are heavily interrelated, and agents cannot accurately predict or model the beliefs or utility values of other agents. However, agents' utilities can change over time. In general, problems are overconstrained and complete goal satisfaction is impossible.

The basic tasks that PERSUADER can perform are to generate an initial compromise, repair and improve a rejected compromise, and persuade the parties to change their evaluations of a compromise. The problem solving methods employed are: 1) case-based reasoning (prevailing practice); 2) preference analysis; and 3) situation assessment. Case-based reasoning is used to form initial compromise proposals when suitable precedents are found, otherwise preference analysis is applied. An agent's overall utility is modeled as the sum of weighted goal utilities, $P = w_1u_1 + \ldots + w_iu_i$, where P is the overall utility payoff, w is the weight attached to a particular goal, and u is the utility value for that goal. To change a payoff, either the weight or utility value of a particular goal can be changed. When using preference analysis to generate

proposals, the payoffs of alternative solutions are calculated for each agent and the best, or fairest, alternative is proposed. Once a solution is proposed, the possible responses to rejection are to: 1) persuade the agent to accept the proposal by getting the agent to change its utilities; or 2) modify the proposal to better meet the agent's goals, although this may result in rejection by the other agent.

PERSUADER solicits and maintains goal information on each of the agents involved in the dispute, as would be expected in a mediated arrangement. It uses these goal structures to develop persuasive arguments to get agents to change their positions. For example, it would tell a union representative that a wage increase might cause layoffs if it believes that the layoff issue has a higher priority than the wage issue. The reasoning used by PERSUADER is very sophisticated and highly appropriate to adversary conflicts where agents are unwilling to share their true utilities, where it is unlikely that completely satisfactory solutions exist, and where some agreement must be reached. Furthermore, the use of case-based reasoning by PERSUADER to develop initial ball-park solutions suggests that this approach may be applicable in some situations to help reduce the costs of constructing alternative solutions.

2.4.6 Concurrent Engineering

A primary characteristic of concurrent engineering (sometimes called simultaneous engineering) is that it is important to consider the effect of decisions that will be made throughout the life-cycle of a product at the time the product is designed. In [Winner, 1988], concurrent engineering is defined:

> Concurrent Engineering (CE) is a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support.
>
> This approach is intended to cause the developers, from the outset, to consider all elements of the product life cycle from conception through disposal, including quality, cost, schedule, and user requirements.

When expertise is brought to bear in a sequential fashion, e.g., when a completed design is presented to a manufacturer, important constraints may be overlooked. If the path fails, much of the work that has been done turns out to be irrelevant and, if constraints could be introduced earlier, some of that work could be avoided. In [Sriram *et al.*, 1991], Sriram describes the problem in a product development domain, including a quotation from Business Week, April 30, 1990, p. 111:

> The present method of product development is like a relay race. The research or marketing department comes up with a product idea and hands

> it off to design. Design engineers craft a blueprint and a hand-built prototype. Then, they throw the design "over the wall" to manufacturing, where production engineers struggle to bring the blueprint to life. Often this proves so daunting that the blueprint has to be kicked back for revision, and the relay must be run again – and this can happen over and over. Once everything seems set, the purchasing department calls for bids on the necessary materials, parts, and factory equipment – stuff that can take months or even years to get. Worst of all, a design glitch may turn up after all these wheels are in motion. Then, everything grinds to a halt until yet another so-called engineering change order is made.

The work below describes approaches to concurrent engineering that bring relevant information and constraints to the forefront at appropriate points in the development cycle.

Cooperative Design Evaluation: Werkman has developed a negotiation-based approach to multi-perspective evaluation of designs [Werkman, 1992] . The Designer Fabricator Interpreter (DFI) system assists a human engineer by selecting and critiquing alternatives from a set of steel beam-to-column building connection designs. The agents in the system parallel human design, manufacturing, and assembly critics. In building this system, Werkman has addressed many of the same issues that were seen in the development of TEAM, e.g., the use of conflict as the primary controlling factor in the agent set, the need for a common language, local evaluation criteria and acceptability measures, system termination, and the possibility of different power structures and cooperation strategies within the agent set. However, in this work, the agents are redundantly reviewing a fixed set of alternative solutions from different perspectives rather than incrementally constructing solutions with diverse expertise. Werkman has focused on enabling fine discrimination between alternative designs. To do this, the system uses a rigid representation and control structure, incorporating third-party mediation and arbitration to resolve conflicts when necessary. Werkman does not consider issues of agent reusability, specifically the need for local assimilation and relaxation capabilities. The system is particularly appropriate as a support tool for human designers.

DICE: ***D**ARPA **I**nitiative in **C**oncurrent **E**ngineering*: The general problem of integrating diverse technologies in product development has spawned several efforts to develop a less costly and error-prone development cycle, notably the DARPA Initiative in Concurrent Engineering [Jagannathan *et al.*, 1991]. The DICE-WV program[2] attempts to provide computer support for CE in five areas: information

[2]Another CE project, also with the acronym DICE, is in progress at MIT. To avoid confusion, we will call the DARPA project DICE-WV, where the WV indicates that this project resides at the University of West Virginia. The other project will similarly be referred to as DICE-MIT.

sharing, co-locating people and programs, the integration of tools and services with frameworks, team coordination, and the capture of corporate history. Goals of the project include reduced time to market, improved total quality, and lower cost for products or systems developed and supported by large organizations. In conjunction with the project, the Concurrent Engineering Research Center (CERC) has been established at Morgantown, WV to serve as a national repository for CE material including a library, testbed, and a research staff.

Much of the work taking place in the DICE-WV project is concerned with embedding CE techniques in a realistic environment. For example, a major effort might be required to connect heterogeneous platforms so that engineers working with an X-windows application can communicate with engineers and tools running in VMS. In the TEAM application systems, we do not attempt to model real-world environments at that level of detail for the time being. However, it is interesting to note that the DICE-WV approach to integration is to use *wrappers* to standardize data access and user interfaces to and from heterogeneous tools. In TEAM, we are investigating the use of wrappers for standardizing the semantic input and output of operators within a tool or around a tool.

DICE-WV is not yet addressing issues of conflict management in CE since the current focus of the project is to develop the tools to enable interaction in a heterogeneous environment. However, we believe that these issues will become more important as the underlying interaction technology evolves.

DICE: **D***istributed and* **I***ntegrated Environment for* **C***omputer-aided* **E***ngineering*: In [Sriram *et al.*, 1992], Sriram, Logcher, Groleau, and Cherneff describe an architecture for building concurrent engineering systems that we will refer to as DICE-MIT. This architecture is designed specifically to facilitate the cooperation of agents working on diverse engineering problems in an integrated project.

> On July 7, 1981, two skywalks in the lobby of the Hyatt Regency Hotel in Kansas City collapsed. It was cited as the "most devastating structural collapse ever to take place in the United States;" 114 people died and 186 were injured [Marshall and et. al., 1982]. This was not only a failure of a physical structural system, but also a failure of the process by which most projects in the U.S. are designed and built.

The Hyatt failure was attributed to three primary events, each of which served to compound existing errors. Each of these errors should have and could have been caught during the design stage had there been better communication and coordination in the design process. The DICE-MIT architecture attempts to address these issues by providing a framework for the integration of multiple computer-aided tools that

will explicitly inform relevant design participants of potentially important interactions and changes.

The two DICE projects are very ambitious in scope and tackle a number of infrastructure issues that are not examined in TEAM, e.g., the management and storage of potentially huge amounts of data and low-level communication protocols for heterogeneous platforms. On the other hand, the current versions of the projects do not address some of the large conceptual issues such as conflict management that will be required in a comprehensive CE system (the incorporation of this type of knowledge is planned in future work however). Although the TEAM and DICE projects have had different foci to date, there are large areas of overlapping interest that will become more significant as the projects move forward.

Although concurrent engineering as a domain is outside the scope of this dissertation, we believe that it is a highly relevant domain to this project. Therefore, we have provided flexibility in the TEAM framework to support concurrent-engineering approaches to product design in a general way. For example, one technique that can be applied is to allow agents to begin working without complete specification of their input parameters, instead giving them the capability to make assumptions about likely values. If the assumptions are mistaken, they may have to retract some work. However, they have the opportunity to speak up early in the design process and communicate important constraints. In Chapter 5, we will describe an initial investigation into the use of default assumptions for the values of required input parameters. This work both enables more concurrency in agent processing and provides explicit mechanisms for agent notification when assumptions are violated.

2.5 Agent Coordination

The coordination problem in distributed systems is described as the coherent scheduling of tasks over the set of distributed agents working on sets of interrelated problems[Decker and Lesser, 1992a]. Although coordination of TEAM agents fits this definition, the most sophisticated work on coordination deals with the issues that arise due to the impact of dynamically-changing environments, physically-distributed and/or uncertain data, and time and cost constraints on problem-solving.

Because we are looking only at systems that perform distributed state-based search, the coordination protocol to be applied in a particular system is defined to some degree by the search strategy selected. That is, the order of operator application for the evolution of a single solution is at least partially defined by the search strategy, and the search strategy is a static and globally known entity. This reduces the importance of dynamic coordination capabilities among agents but does not entirely

obviate the need for coordination. There are still issues to be addressed around the coordination of agents with respect to multiple solution paths, around the dynamic selection of search strategies, and around the response to specific conflict situations.

The relationships between distributed search, coordination, and negotiation are tightly interwoven and it is often difficult to decide how to characterize a particular research project. The projects described in this section focus primarily on how the actions of each agent can be synchronized with those of other agents to achieve the best possible overall system coherence.

Distributed Air-Traffic Control: Cammarata, McArthur, and Steeb describe strategies for cooperation in the domain of collision avoidance in air traffic control [Cammarata *et al.*, 1983]. Each agent is responsible for a single airplane, and must plan a route that maximizes local utility for that plane while ensuring that it does not get too close to other planes. They developed two cooperation strategies: task centralization and task sharing. Most of the experimental investigation of the cooperation policies involved the task-centralization policy. In this policy, when a conflict (potential collision) is detected, the set of agents select a single agent to revise its plans to avoid the collision. Several methods for selecting that agent were investigated: *selection by shared convention*, *selection of the least spatially constrained agent*, and *selection of the most knowledgeable, least committed agent.*

Contract Nets: Davis and Smith designed the contract net protocol as a control mechanism for the decomposition and allocation of tasks within a network of agents [Davis and Smith, 1983]. They used an iterative bidding process to form contracts between a contractor and a contractee, dynamically building hierarchical organizations of nodes. The protocol is an interesting approach to negotiating contracts where one agent plays the role of the manager and controls the selection of alternative bids. Other agents have some control over the selection process by determining whether or not to bid and what to offer. The protocol uses a single iteration of announce, bid, and select to award contracts.

Coordination as Distributed Search: Durfee and Montgomery have looked at the process of coordination as itself involving distributed search through a hierarchical space of potential behaviors [Durfee and Montgomery, 1991]. They describe five components of this search: hierarchical representation, metrics for evaluating the acceptability of a coordination plan, a distributed-search protocol for structuring agent interaction and communication, local-search algorithms, and control knowledge. To investigate these issues, a simulated robotic task was implemented where robots are responsible for producing, transporting, and consuming materials. The coordination of the transporter robots is the primary focus. In order to deliver the materials,

robots must plan to navigate through a spatial environment where conflicts occur when multiple agents collide. Potential conflicts are resolved through a specific set of authority ratings on agents and priority ratings on behaviors. The approach taken by Durfee and Montgomery examines many of the same issues we have looked at in TEAM including the integration of distributed-search protocols, local search, and solution evaluation metrics. In Durfee and Montgomery's work to date, there are more questions than answers, but the two approaches appear to be complementary and future work could lead to interesting merge of ideas.

Meta-Level Control for Coordination: Corkill and Lesser look at the problem of achieving coordinated activity among the nodes in a distributed sensor network [Corkill and Lesser, 1983]. Internode communication in these networks is a major issue since it is both limited and potentially unreliable. Limited node interaction makes it infeasible to keep every node completely informed of the information possessed by other nodes in the network. Adequate coordination is difficult without a shared view of how the network is attempting to solve the problem and of the general roles and responsibilities of each node. A shared organizational structure is used to give the network some long-term stability and structure while allowing for dynamic local decision-making in response to a changing environment. Agents are allowed to be *skeptical* within the structure and employ sophisticated local control to balance local concerns with the need for global coherence.

Coordinating Plans of Autonomous Agents: vonMartial uses negotiation to coordinate agents' local plans in a locally cooperative environment [von Martial, 1992]. This work looks at both positive and negative plan interactions. A model was developed that is designed to facilitate the coordination of plans through the detection of relationships among the plans, predictive analysis of both harmful and beneficial relationships, and iterative replanning based on detected relationships. The model describes an integrated network of homogeneous planning agents, and an iterative protocol for reconciling plan relationships.

Partial Global Plans: Durfee and Lesser have created a coordination framework, *partial global plans* (PGP), that increases the coordination of a set of homogeneous agents in a network by building and communicating abstracted global plans based on precise local plans[Durfee and Lesser, 1988, Durfee and Montgomery, 1990, Durfee and Lesser, 1991]. The global plans introduce some level of global coherence but allow some cushioning of the effect of changes in local plans. In other words, if a local plan changes in response to some unanticipated event, the global plan may or may not be updated to reflect that change depending on factors such as magnitude of the local change. Decker and Lesser have generalized the PGP planner to ab-

stract and quantify coordination relationships in a domain-independent way among semi-heterogeneous agents[Decker and Lesser, 1992b]. The themes of coordination and conflict resolution are closely related: conflict resolution activities often require coordination and coordination activities often require conflict resolution. In this dissertation, we do not deal with the sophisticated coordination of long-term activity among agents such as those studied by Corkill, Durfee, Decker, and vonMartial, and Lesser since we have focused on search problems in stable environments that can be coordinated through static global strategies. Instead, we look at the problems of coordination that result from the dynamic selection of search strategies. In future work, we will look at issues in coordinating agents around specific conflict-resolution episodes.

2.6 Summary

In this chapter, we have described a number of projects that have some degree of relevance to the dissertation work. It is clear that there has been a great deal of work done in such general areas as distributed search, negotiation, and agent coordination. However we reiterate that, to date, no other work comprehensively examines the many interwoven issues that arise in specifying and building search systems of heterogeneous and reusable agents. We envision the future integration of the general and flexible mechanisms developed for the TEAM project with work being done by other Distributed Artificial Intelligence researchers. For example, the TEAM work is complementary and can be integrated with formal models of negotiation, with the infrastructure mechanisms of the large DICE concurrent engineering efforts, with various distributed-search algorithms, and with taxonomies of conflict types and associated resolution procedures. These projects all contribute to various aspects of the overall problem and we look forward to their eventual synthesis.

CHAPTER 3

DISTRIBUTED SEARCH

In this chapter, we provide a theoretical foundation for understanding the interactions of agents involved in distributed search. We begin with a brief discussion of how the state space of a problem is logically partitioned among a set of agents in distributed search. We describe how an agent set can be constructed given a library of reusable agents. We continue by describing the relationship between the local solution spaces of agents (the subjective view) and the global space in which complete solutions will lie (the objective view). We describe how information flows between agents, what types of information can flow through the agent set, and how agents learn about the larger environment in which they are embedded. Finally, we discuss the coordination of agent actions through a state-based representation of coordination strategies.

3.1 Selecting an Agent Set

Artificial Intelligence has long been concerned with search. In [Barr and Feigenbaum, 1981], three components of search systems are described:

1. The *database* describes the current task-domain situation (or the *state* of the problem) and the goal;
2. A set of *operators* are used to manipulate the database;
3. A *control strategy* is used for deciding what to do next, specifically, deciding what operator to apply and where to apply it.

When all operators reside in a single program or logical entity and have access to the same knowledge and databases, the search is centralized. In [Lesser, 1990], Lesser discusses distributed search:

> A distributed search involves partitioning the state space and its associated operators and control regime so that multiple processing elements can simultaneously perform local searches on different parts of the state space; the (intermediate) results of the local searches are shared in some form so that the desired answer is produced in a timely manner. The requirements for an effective distributed search may necessitate a reorganization of the state space search in ways which would not be optimal for a sequential search. Research in DAI emphasizes the development of approaches for partitioning the search space into a set of local searches and coordinating these simultaneous searches in terms of both information sharing and control.

In systems of agents that are logically distributed, the partitioning of the state space of the system is induced by the *a priori* problem decomposition imposed by the agent set. In globally cooperative domains, the agent set is constructed by selecting agents that have appropriate capabilities and that can produce values for the solution parameters required for global solutions. Agent selection is an iterative process, either working from a problem specification to a solution or vice versa. Assuming the case where agent selection starts with a problem specification, in the first round of selection, the chosen agents must be able to operate given only the information available in the specification. These agents generate outputs that are added to the information available which is then used to select the next round of agents. The process repeats until a set of agents is found that can jointly generate the required flow of information from a specification of the problem to a complete solution (see Figure 3.4 in Section 3.3).

When building agent sets for globally cooperative systems, a likely goal for a system designer would be to find a set of agents that collectively possess all required expertise without any redundant expertise. Redundancy usually introduces extra processing time and may introduce unnecessary conflict, making it a generally undesirable property. There are situations, however, where redundancy is desirable, particularly where reliability is an issue, when an agent's expertise is uncertain or distrusted, when agents have different criteria for evaluating a solution, or when there is an unacceptable amount of uncertainty or ambiguity implicit in the domain or the problem-solving methodology of an agent. When a system developer is choosing from a set of existing reusable agents, the degree of redundancy may not be completely within her control. For example, an agent that can generate a value for x may coincidentally generate a value for y. y may also be produced by another agent that also generates values for z. If x, y, and z are all required resources and these are the only agents available for x and z, the redundancy over y is unavoidable.

In contrast, in locally cooperative domains agents are not selected because of their potential contribution to a joint solution; instead an agent is included in an agent set because it has capabilities or output that are of singular value. In this type of problem solving, the system simply provides an environment for a set of agents with potentially interdependent goals. For example, a system might be composed of a set of robotic agents that each have tasks to accomplish in a warehouse. During the course of accomplishing their local tasks, they may find some interaction opportunities that are mutually beneficial, some that are mutually detrimental, and others that are possible, but that have a neutral payoff for one or more of the agents participating. As discussed earlier, agent interactions in this type of system are more serendipitous.

Therefore, the overlap of solution spaces need not be explicitly planned except to the extent that there must be a producer available for every resource required by agents to achieve the high-priority tasks in the domain. For example, if an important task in the robotic warehouse environment is to assign packages to a carrier, some robot(s) must be able to produce lists of packages that will need carriers.

3.2 Shared Spaces

When talking about the shared spaces of agents, we distinguish between the *local* space of an agent and the *composite* space of the system. A local space is one that is private to an agent, the composite space is one that is shared by all agents. The local *solution space* of an agent is defined by the parameters of a local solution while the local *search space* is defined by the parameters the agent uses to constrain its local search. Each agent starts out with a completely local view of search and solution spaces. However, this local view is unlikely to be effective in finding solutions that are mutually agreeable to all agents (solutions in the composite space). A primary goal of communication among agents, therefore, is for each agent to end up perceiving the closest approximation possible to the actual composite search and solution spaces. In nontrivial cases, it is unlikely that a complete and correct global view can be achieved at every agent. However, to the extent that its local view approaches the global view, an agent is likely to be more effective at proposing solutions that will be mutually acceptable (see Chapter 8).

An agent set contains a number of solution spaces: the local spaces of each of the agents, and the composite space in which problem solutions lie. Each reusable agent comes with a "built-in" local solution space that is defined by the parameters of the solutions of that agent. In other words, the local solution space is defined by the parameters that are assigned values by an agent in its local solutions. In this section, we will use examples from the **STEAM** domain to illustrate the concepts being discussed. Let $\mathcal{A}$ be the set of agents in **STEAM**:

$$\mathcal{A} = \{\texttt{pump-agent},\texttt{heat-exchanger-agent},\ldots,\texttt{platform-agent}\}$$

Figure 3.1 shows a simplified version of the solution space of **pump-agent**. This figure is simplified both in the number of parameters and the specification of the parameters' domains.

The set of parameters in the solution space of an agent α will be represented as $\mathcal{P}^{\alpha}$, the *parameter set* of α. The parameter set of **pump-agent**, as shown in Figure 3.1, is {water-flow-rate,head,run-speed,pump-cost}.

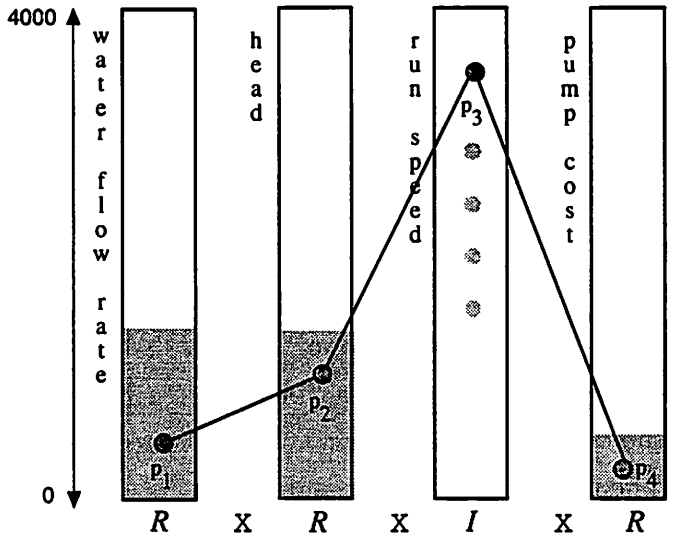


Figure 3.1. *The Local Solution Space Of Pump-Agent from the* STEAM *System*

The set of legal values for a parameter θ at agent α is its *parameter space*, $\mathcal{V}_\theta^\alpha$. For example, the parameter space of `run-speed` from Figure 3.1 is the set of integers {1200,1800,2400,3000,3600}. The legal values of parameters can be of different types such as integers, reals, numeric intervals of the form $\{(min,max),\ min,max \in \mathcal{R}\}$, or discrete labels such as $\{model\text{-}1,\ model\text{-}2,\ldots,model\text{-}n\}$. A solution in the solution space of α is a tuple $s_j^\alpha = (p_1, p_2, \ldots, p_n)$ such that $p_x \in \mathcal{V}_x^\alpha$ and such that all constraining relationships on and between the $p_x \in s_j^\alpha$ are satisfied. A parameter space may be constrained by explicit constraints on solutions such as (`run-speed` $\geq$ 1200) or through implicit requirements that are embedded in the functions an agent uses to search for solutions.[1] As a trivial example of an implicit constraint, consider the following loop in pseudo-code:

```
head :== 0;
DO  water-flow-rate = 0 to 500
  new-head :== calculate-head water-flow-rate;
  head :== select-best new-head head;
END LOOP;
```

An agent using this code implicitly constrains the parameter space of `water-flow-rate` to be the set of integers from 0 to 500, although it may not declaratively represent this anywhere. In reality, functions tend to be more complex and the implicit constraints more difficult to discern. In the above example, the value of `head` is tacitly constrained by the implicit constraint on `water-flow-rate`. However,

[1] We use the terms *solution requirement* and *constraint* interchangeably.

the effect of the implicit head constraint on the values that can be assigned to that parameter may not be determinable without in-depth expertise.

Implicit constraints cannot be expected to be shared in general, since they are an integral part of the agent's expertise and cannot be easily extricated.[2] These unshareable constraints strongly affect properties of the agent sets in which they are embedded. For example, in [Khedro and Genesereth, 1993], Khedro and Genesereth present a search model in which agents provably converge on a globally satisfactory design if one exists. However, the property of convergence can only be guaranteed if all constraining information can be explicitly exchanged. When implicit constraints are added, this desirable property no longer holds.

The existence of implicit constraints must be expected in the general case. If agents can describe all of their knowledge by listing constraints, there is some question about whether they have any real domain expertise. For example, in resource allocation problems, it is sometimes the case that all agents have the same basic architecture, problem-solving techniques, etc., and that their differences lie in owning different resources or having different priorities on the use of particular resources. In this situation, centralized constraint processing is likely to be more efficient. However, there are reasons for using distributed search even when the agents don't have heterogeneous expertise, for example, when agents are physically distributed and the volume of information precludes complete exchange or if the cost of information exchange is prohibitive. The distribution may also be inherent in domain due to a need for autonomy or privacy based on competitiveness, e.g., Sandholm describes a system of transport companies that must decide how to most profitably route their vehicles to accomplish required deliveries [Sandholm, 1993]. These companies are competitive and do not wish to expose unnecessary information, but are willing to cooperate when doing so increases their profitability. In these cases, although all constraints may be locally explicit, since not all information can be exchanged, unexchanged information must still be considered implicit in the composite solution space.

Explicit (declaratively represented) constraints are those that can be shared and, as will be discussed in Chapter 8, this sharing can greatly enhance the effectiveness and coherence of the agent set. In **STEAM**, explicit constraints are limited to simple boundary constraints of the form (`water-flow-rate` < 800) and (`water-flow-rate` ≥ 0) that specify minimum or maximum values for a parameter. In Chapter 8, we

[2]It is possible that some agents may be able to share either code or some form of abstracted explanation of implicit constraints. However, this requires specialized capabilities on the part of both the sending and receiving agents. **TEAM** is entirely neutral on the types of capabilities that can be embedded in an agent: it does not forbid this type of communication, neither does it require it.

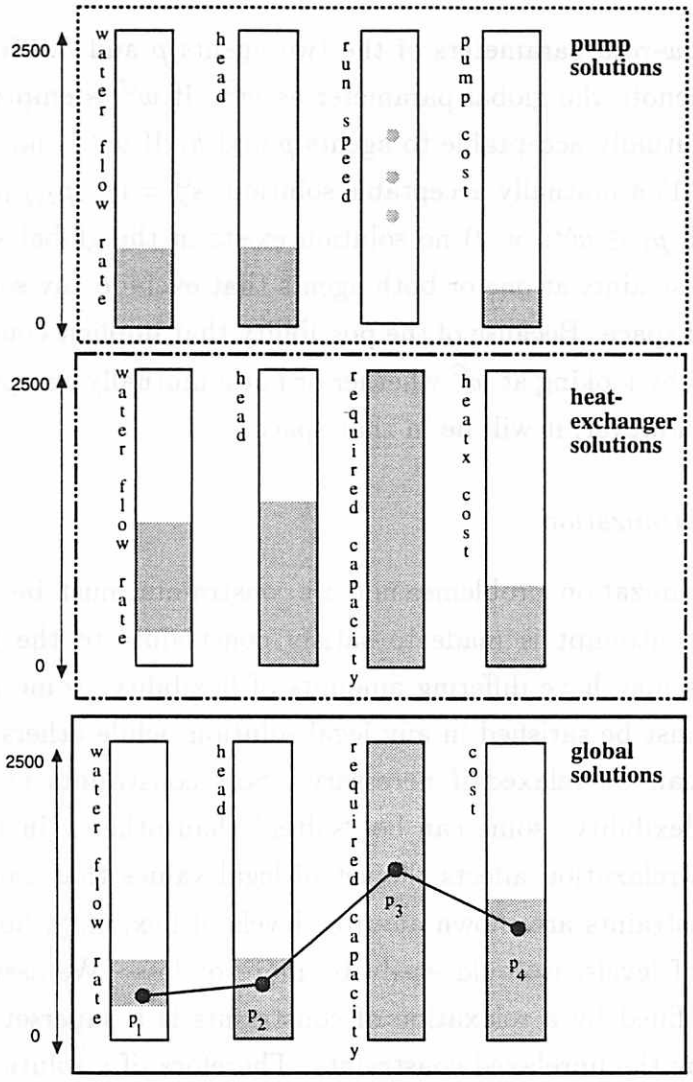
will describe experiments in which we measure the effect of exchanging constraints of this form. To explicitly include these constraints in the definition of a solution space, we use the following notation: let c_j^α be a declaratively represented local solution requirement of agent α in the set of all explicit solution requirements of α, $\mathcal{C}^\alpha$. Then, let the notation $\{c_j^\alpha : s_k^\alpha\}$ mean that c_j^α is satisfied with respect to a particular solution, s_k^α. For example if c_1^α is $(p_1 = p_2 + 10)$ and $s_1^\alpha = (15, 5, 3, 7)$, then $(15 = 5 + 10)$ is true and therefore $\{c_1^\alpha : s_1^\alpha\}$. If c_j^α is neutral with respect to s_k^α (it does not constrain any parameters in s_k^α), it is considered to be satisfied.

Using this notation, the shareable solution space of agent α can be defined by specifying the parameter set of α, $\mathcal{P}^\alpha$ and the set of explicit solution requirements over those parameters, $\mathcal{C}^\alpha$. This shareable solution space is an approximation of the actual solution space since it does not represent any implicit solution requirements that are embedded in the agent. We formally describe the shareable local solution space of agent α as follows: $\delta^\alpha = \{(p_1, p_2, \ldots, p_n) \mid \forall c_j \in \mathcal{C}^\alpha, \{c_j : (p_1, p_2, \ldots, p_n)\}$. In nontrivial cases, δ^α will be a superset of the valid solutions of agent α since it does not take implicit constraints into account.

The Composite Solution Space: Given a set of agents, $\mathcal{A}$, and a problem that they are cooperating to solve, the desired solution must derive its parameter values from the local solution spaces of the agents. However, the parameters of the global solution space are not necessarily the union of all parameters in the local solution spaces as can be seen in Figure 3.2.

In this figure, the solution space of agent p (the pump agent) contains the parameters *water-flow-rate*, *head*, *run-speed*, and *pump-cost*. The solution space of agent h (the heat-exchanger agent) contains the parameters *water-flow-rate*, *head*, *required-capacity*, and *heatx-cost*. Ignoring for the moment the problems associated with determining the semantic equivalence of local parameter spaces, we find in Figure 3.2 that the parameters *water-flow-rate* and *head* are common to both agents while *run-speed*, *pump-cost*, *required-capacity*, and *heatx-cost* represent parameters unique to individual agents. The global solution space, $\mathcal{G}$, contains the shared parameters, *water-flow-rate* and *head*, the parameter *required-capacity* from agent h, and also a unique parameter, *cost*. *Cost* is not local to either agent p or agent h, but represents a transformation on local parameters of those agents, i.e., the sum of *pump-cost* and *heatx-cost*.[3] To summarize, each parameter in $\mathcal{G}$ is local to either

[3]Simple arithmetic and set transformations on local solution parameters can be performed by the TEAM framework. More knowledge-intensive transformations must be done by an expert agent. This is addressed in our discussion of the TEAM architecture in Chapter 4.



A simplified view of the relationship of the solution spaces of the pump agent and heat-exchanger agent to the desired global solution space.

A solution in the global space, (p_1,p_2,p_3,p_4), is constrained by the local shareable solution spaces of the agents.

Figure 3.2. *Constructing a Global Solution from the Local Solutions of Agents*

agent p or agent h, local to both agent p and agent h, or is a unique parameter whose value can be derived from parameters local to agent p and/or agent h.

In Figure 3.2, notice that the constrained set of values (the shaded areas) of the shared parameters, *water-flow-rate* and *head*, are not identical for the two agents. Assuming for now that we are looking only at constraint-satisfaction problems, problems in which all constraints must be satisfied or no solution can be found, the constrained global parameter space is the intersection of the constrained local parameter spaces.[4] When the intersection is empty, no solution exists. Consider a global parameter such as *water-flow-rate* from Figure 3.2 which is the intersection

[4]We handle constraint optimization as well, in which some constraints need not be fully satisfied in a solution. We will discuss the requirements of constraint optimization in Section 3.2.1.

of the local *water-flow-rate* parameters of the two agents p and h. To differentiate the spaces, we will denote the global parameter as w^G. If w^G is empty, no solution exists that will be mutually acceptable to agents p and h. If w^G is not empty, there are two possibilities: 1) a mutually acceptable solution, $s_x^G = (p_1, p_2, p_3, p_4)$, exists in the global space with $p_1 \in w^G$; or 2) no solution exists in the global space because there are implicit constraints at one or both agents that exclude any solutions in the explicitly constrained space. Because of the possibility that implicit constraints exist, it is impossible to tell by looking at w^G whether or not a mutually acceptable solution exists. If one exists, however, it will be in that space.

3.2.1 Constraint Optimization

In constraint-optimization problems, not all constraints must be satisfied in a solution. Instead an attempt is made to satisfy constraints to the fullest extent possible. Constraints may have differing amounts of flexibility: some may be *hard*, meaning that they must be satisfied in any legal solution, while others may be *soft*, meaning that they can be relaxed if necessary. Soft constraints again can have different degrees of flexibility: some can be "softer" than others. In Figure 3.3 we show how constraint relaxation affects the set of legal values that can be assigned to a parameter. Constraints are shown at three levels of flexibility, however, this is an arbitrary choice of levels, it could easily be more or less. We assume that the set of legal values defined by a relaxation of constraints is a superset of the set of legal values defined by the unrelaxed constraints. Therefore, if a solution satisfies an unrelaxed constraint, it will satisfy the relaxed version of that constraint as well.

A simplified view of a generic parameter space. Shaded areas represent legal values for the parameter at different levels of constraint flexibility.

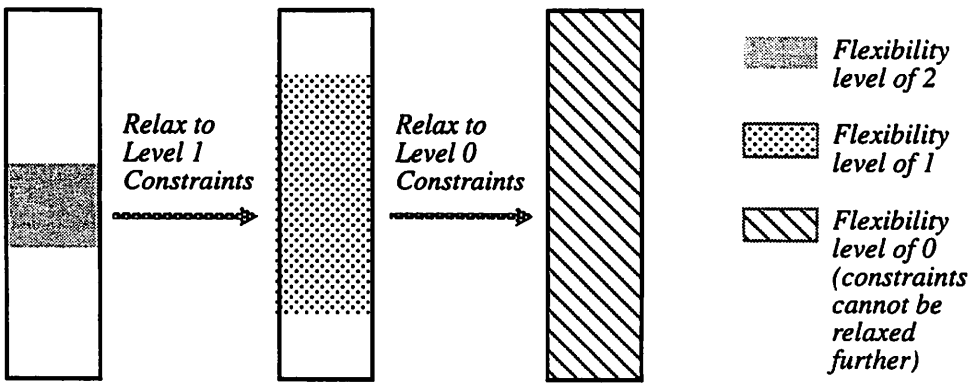


Figure 3.3. *Legal Values of a Parameter Under Flexible Constraints*

In constraint-optimization problems, both centralized and distributed, the order in which constraints are relaxed can strongly affect system performance and solution quality. Several researchers have looked at developing heuristics for constraint ordering [Fox *et al.*, 1982, Nishibe *et al.*, 1992]. In distributed search, not only is constraint ordering at a single agent important, but power and role relationships among agents must be considered. We will examine the effects of different agent/role assignments on system performance in Chapter 9. Constraint-ordering and agent-organization issues cannot be fully addressed at the level of generic problem-solving architectures or algorithms because they depend on specific application knowledge. However, we will examine a general algorithm that enables distributed search using constraint optimization in Chapter 5 and look at how this algorithm works in a specific domain in Chapter 6.

3.3 Information Flow within an Agent Set

In this section, we describe how information flows among agents, both to provide inputs required by other agents, and to communicate local constraints on parameters. Although Figure 3.2 shows a two-level solution-space structure, with the agents' solutions at level 1 supplying information to the global solution at level 2, the relationships between agents can actually be tree-structured or graphically structured as shown in Figure 3.4.

One primary goal of agent communication in reusable-agent systems is *results-sharing*: an agent produces outputs that can be used as input by other agents. The flow of information for this type of communication is dictated by the structure of the agent set as indicated by the arrows in Figure 3.4. A second goal of communication among agents is to refine the locally perceived parameter spaces that are built in to each agent so that the agent perceives the actual explicitly constrained composite parameter spaces of all of its shared parameters. Although this doesn't guarantee that the agent set will find a mutually acceptable solution, as discussed above, it does mean that an agent will not waste time doing local search in areas of the parameter space that it has been explicitly told are not acceptable to other agents.

The communication protocol used to achieve the required information flow for constraint sharing may be either proactive, reactive, or a combination of both. A proactive protocol would have an agent inquire about potential constraints on a parameter before generating a value. A reactive protocol would wait until a value has been assigned and then communicate violated constraints. The relative effectiveness of proactive versus reactive protocols is at least partially dependent on the costs of communication, value generation, and retraction of proposed solutions; however, these

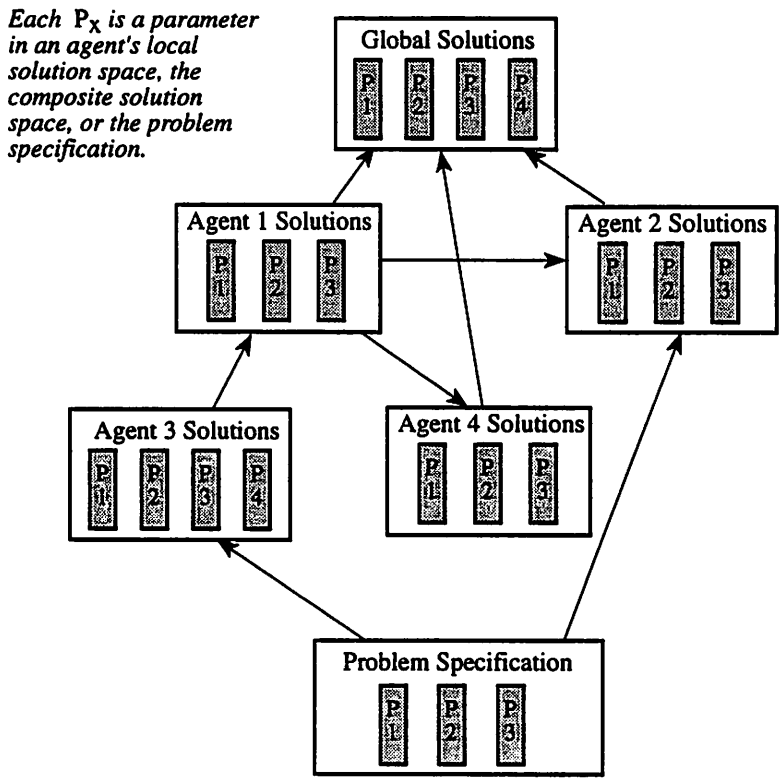


Figure 3.4. *An Example of the Information Flow through an Agent Set from Problem Specification to Global Solution*

issues are beyond the scope of this dissertation. The search algorithms developed for this dissertation use a strictly reactive approach.

3.4 The Coordination Space

The solution parameters that are important from a coordination perspective are not the same as those that are important from a domain perspective. For example, in the steam condenser design domain of **STEAM**, the **required-capacity** and **cost** parameters of a global solution are important from a domain perspective because they partially specify the problem and define the quality of the solution. From a coordination perspective however, other parameters, such as the **agents**, **acceptability**, **termination-status**, and **completeness** parameters are important because they define the state of problem-solving.

Coordination of distributed search takes place in a state space that is determined by viewing a global solution as an evolving artifact that moves from some initial state to a termination state through the application of operators. Each agent applies one or more operators, possibly performing complex local processing to achieve the functionality of the operator. However, from the viewpoint of a specific solution, the

transformations from one state to the next are effected with simple updates to solution parameters. For example, a simple two-agent coordination strategy for generating solutions and determining their mutual acceptability is shown in Figure 3.5. From

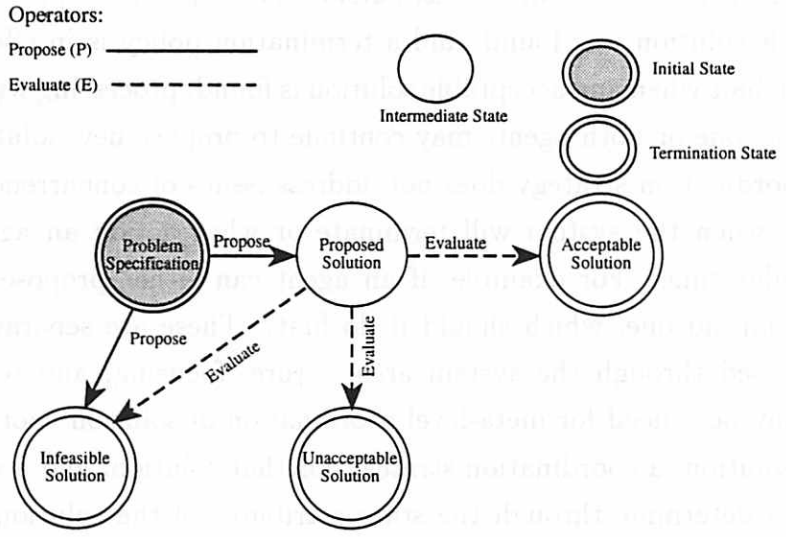


Figure 3.5. *A Simple Two-Agent Coordination Algorithm*

a coordination perspective, the important attributes of a solution in a realization of this algorithm are *solution-type* (problem-specification, proposed, acceptable, unacceptable, infeasible), and *contributing-agents* (a_1, a_2, (a_1, a_2)).

The *propose operator* is applied to a solution object with a *solution-type* of `problem-specification`. Note that this algorithm assumes that the problem will be specified by assigning values to some attributes in a solution object. This is a common method for specifying a problem, but is not meant to be taken as a required method. Another possible method would be to supply a set of constraints that define acceptable ranges on attributes rather than specific values. Returning to the example, the *propose* operator transforms the problem-specification into a solution with a *solution-type* of either `proposed` or `infeasible` and with a *contributing-agents* attribute containing the name of the agent that applies the operator. If the *solution-type* is `infeasible`, the solution has reached a termination state and no further processing will be done on that solution. This occurs when no feasible solution can be found given the problem specification. Otherwise, the *evaluate* operator can be applied to the solution. However, we define the evaluate operator to have a precondition which requires that the name of the agent applying *evaluate* must not be contained in the `contributing-agents`, so that only the agent that did not propose the solution can evaluate the solution. The *evaluate* operator transforms a proposed solution into one

that has a *solution-type* attribute of `acceptable`, `unacceptable`, or `infeasible`. The solution is now in a termination state and no more work will be done on that particular solution. However, the coordination strategy does not specify what will happen next in the system, that is, outside the scope of the proposed solution. If an acceptable solution was found, and a termination policy is in effect that directs the system to halt when any acceptable solution is found, processing will halt immediately. Otherwise, one or both agents may continue to propose new solutions.

A coordination strategy does not address issues of concurrency. It also does not mandate when the system will terminate or what action an agent should take at a particular time. For example, if an agent can either propose a new solution or evaluate an old one, which should it do first? These are separate issues that must be addressed through the system architecture, focusing, and termination policies. There may be a need for meta-level coordination of solution coordination. However, given a solution, a coordination strategy for that solution, and a set of operators, an agent can determine through the state attributes of the solution what operators it can legally apply at any given time. No matter how complex the search functionality of the operator may be, the state transformations in the coordination space can be determined separately from the domain-level knowledge.

CHAPTER 4

THE TEAM FRAMEWORK

In this chapter, we present the TEAM framework. The framework is a knowledge-based system that supports cooperative distributed search among a set of reusable agents, where the agents are themselves complete and heterogeneous systems. TEAM is application-domain independent: application knowledge is contained in the agents and in isolated structures and functions of the framework that are specified when an agent set is selected and integrated into an application system. The domain problem addressed by the framework is the management of reusable-agent interaction and integration.

4.1 Managing Information in a Reusable-Agent Application System

Information within a reusable-agent application system is partitioned according to the accessibility of that information. The framework must support agent autonomy, simultaneously enabling agent interaction and insulating agents from the need to know internal details about each other. Figure 4.1 distinguishes four classes of information based on accessibility: one that is only accessible to and used by the framework (the *framework level*), one that is shared by the framework and an individual agent (the *shared level*), one that is accessible to and used by the framework and the complete agent set (the *common level*), and one that is accessible only to an individual agent (the *agent level*).

Framework-level information is used to build representations for agents and establish communication paths, to define a framework language and common memory structures, to invoke agents and respond to agent requests, to maintain information about coordination strategies, to define a termination policy and execute that policy, and to define an acceptance policy and establish global evaluation criteria for composite solutions. *Shared-level* information specifies knowledge about particular coordination strategy and operator availability, and search- and solution-space characteristics of each of the agents in the agent set. *Common-level* information includes the common language, the problem specification, and instances of composite solutions, strategies, and other objects that are defined in the common language and reside in common memory. *Agent-level* information includes the private language of

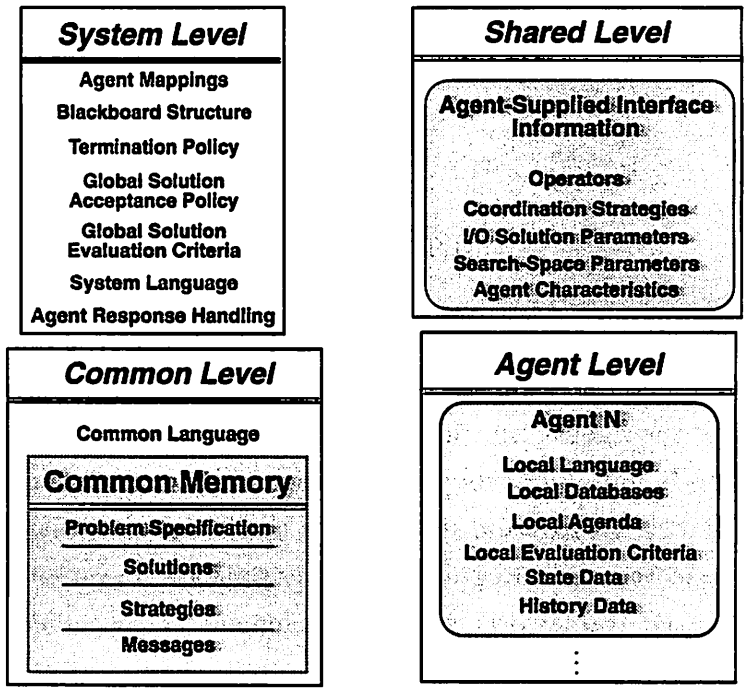


Figure 4.1. *Classes of Information*

an agent, local databases, the local agenda, evaluation criteria, and private state (e.g., the termination phase the agent is operating in) and history data (e.g., information about local solutions that have been previously tried).

These classifications help the application-system designer to provide clean interfaces between the agents and the framework. In order for agents to be added to and deleted from the system easily, all required coupling information must be available through a small, easily modifiable interface. For example, when an agent is added, the name of an invocation procedure for that agent must be supplied to the TEAM framework. TEAM then adds the name to a queue of agent invocation procedures to ensure that the agent has the opportunity to execute during each agent cycle.

4.2 *Motivating Factors for the* TEAM *Framework*

Agent heterogeneity and reusability impose very strong modularity requirements on an integration framework. Some of the general motivations for TEAM framework decisions are:

- Agents have heterogeneous domain-dependent expertise and are not expected to share this detailed knowledge except as necessary to resolve conflicts or facilitate interactions. However, agents must be able to share short-term problem-solving information, i.e., *state information*, such as emerging solutions, coordination

strategies, and feedback messages. This can be accomplished by: 1) maintaining a centralized common memory; or 2) having each agent maintain its own memory and providing a framework-level protocol for distributed memory management. In some domains, there are specific reasons for distributing memory. For example, reliability of transmission among agents, security of communicated information, reliability of the memory itself, and cost of communication to a remote common memory are issues that might affect the decision. However, it is logically simpler and less resource-intensive to build a common memory and, in the general case, a common memory is an effective communication media. We chose to use a centralized common memory for TEAM because of its logical simplicity and because it is adequate for search problems that involve logically distributed (as opposed to physically distributed) agents. However, distributing the common memory is an area that we intend to investigate in future research.

- Reusable agents should not know the specifics of accessing and managing memory resources. There are several reasons for this:

  1. The common language and object set will be different for each domain application.
  2. Physical and logical paths to objects and partitions are subject to change with every instantiation of an application system.
  3. The underlying memory architecture may change.
  4. The memory access mechanisms might change.

  In response to this requirement, the agents in a TEAM application system do not have direct access to the common memory. Instead, a framework manager, TMan, notifies agents of events of interest and updates the blackboard in response to agent requests through a message protocol.

- Many search algorithms require at least partially sequential operator application (as will be discussed in Chapter 6). Furthermore, many application domains require that knowledge be applied in a partially serial fashion. In order to exploit the potential for concurrent problem solving by multiple agents, multiple solution paths can be simultaneously developed, allowing agents to continue to work by switching from one developing solution to another when they are blocked on a particular path.

- Acceptability criteria for local solutions at the agent level may not be equivalent to acceptability criteria at the global level for composite solutions containing those local solutions. In fact, the global problem to be addressed by an agent

set is not known to the people responsible for designing and building the agents in the set (since agents are expected to be reusable). Therefore, an agent should be responsible for determining the acceptability of its own proposals since it has the highest degree of expertise available for those proposals. However, an agent should not be expected to determine the acceptability of a complete solution, unless it has been specifically designed to evaluate complete solutions objectively. We assume the general case in which no agent in an agent set has the expertise to objectively evaluate a complete solution. In this case, solution acceptability has two elements: 1) agent acceptability as determined by individual agents in the agent set; and 2) global acceptability as determined by a framework-level policy defined by the application-system builder. These two elements are described below.

Different agent-acceptability policies can be implemented depending on the degree of democracy desired for a particular problem. For example, *negotiated search* (see Chapter 5) is fully democratic: all relevant agents must judge a composite solution to be acceptable before it can be considered viable. Other acceptance policies might require some percentage of agents to judge a solution as acceptable, or they might require a specific average or minimum agent evaluation. Another approach would be to give some agents more power than others in determining acceptability by biasing the agent-acceptability policy.

Different global-acceptability policies can be implemented depending on the cooperative structure of the domain and the availability of global evaluation mechanisms. For example, in the contracting domain described in Chapter 7, global acceptability is fully defined by agent acceptability: any solution that is acceptable to all agents is considered an acceptable solution. This is true because the domain is *locally cooperative* as described in Chapter 1. No global evaluation of a buy/sell contract is required to make it an acceptable contract—if an agent makes a bad deal, it's still a deal. On the other hand, in domains that are *globally cooperative*, such as the steam condenser design domain described in Chapter 6, evaluation of complete solutions, as opposed to local evaluation of component parts, is necessary. In **STEAM**, global acceptability is determined by an policy that combines agent acceptability with an objective function on attributes of a complete design. This policy is defined in Chapter 6.

- The selection of a coordination strategy for a particular problem is based on two primary factors: 1) characteristics of the search space; and 2) capabilities of agents. In reusable-agent search, an appropriate coordination strategy cannot

be determined until the agent set is selected since it is not known in advance what search capabilities the agents will have. Furthermore, different interactions within a single problem may require different coordination strategies. For example, an agent set might use *negotiated search*, as described in Chapter 6, to be the default coordination strategy but then switch to *linear compromise* to resolve a particular conflict (see Chapter 7). The full agent set may not meet the specific inter-agent relational requirements for linear compromise whereas the subset of agents involved in the conflict do. Therefore, the framework should support the flexible application of different coordination strategies. It should be able to apply a default strategy as its overall coordination protocol but also support dynamic strategy selection in response to specific agent-interaction events.

- The organization of agents may effect performance. There should be some mechanism for an application-system developer to specify organization within an agent set. For each search strategy, there can be multiple assignments of agents to the various roles inherent in that strategy. For example, in the simple coordination strategy that was shown in Chapter 3.4, one agent takes a problem specification as input and produces an initial solution. The second agent takes the initial solution, extends it, and produces a final solution. Depending on the capabilities of the agents, it might be possible to assign either role to either agent. However, the performance of the system might be directly dependent on this role assignment.

4.3 *The* TEAM *Architecture*

The basic TEAM architecture is shown in Figure 4.2. There are three primary components: 1) a common memory; 2) a framework manager; and 3) an agent set. The functionality of each component can be summarized as:

- *Common memory* is a partitioned blackboard memory. A structured blackboard and object language are supplied by TEAM. These provide a core substructure for application-independent multiagent problem solving. When an application system is built, the existing TEAM blackboards and language are extended to describe objects and attributes that are relevant to the application domain.
- The TEAM *manager* (TMan) has the primary functions of managing:
  - Common memory (addition, deletion, retrieval, updating of objects, basic transformations on object attribute values and sets of attribute values)
  - The agent set (addition, deletion, invocation, and i/o requirements of agents)

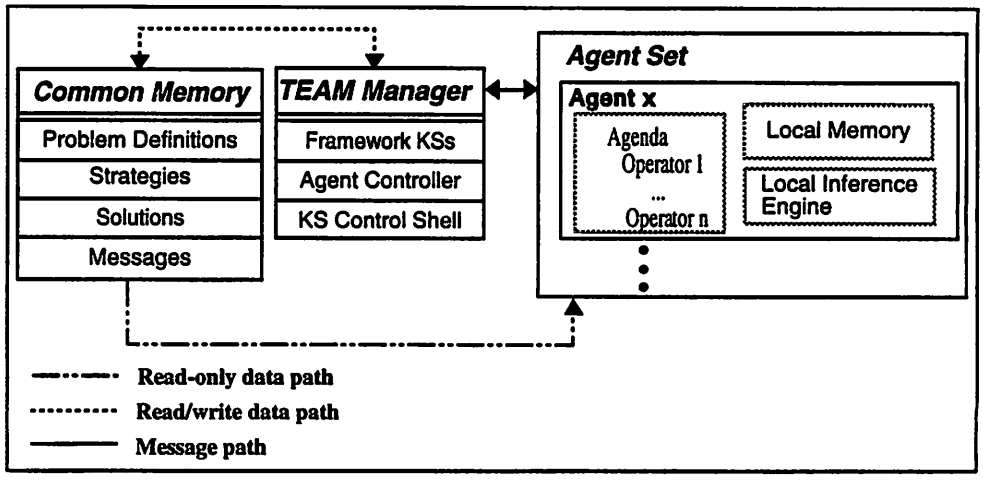


Figure 4.2. *Architecture of the* TEAM *Framework*

- Communication among agents and between TMan and agents (in the form of messages)
- Global solution evaluation and acceptability policies
- Termination of problem solving

- The *agent set* comprises individual agents that are black boxes from a framework perspective. Each agent has private search functionality, private databases, and private local control mechanisms. However, agents are connected to the framework and each other through a set of interface structures and functions and each agent must meet the interface standards.

4.3.1 Overview of the Framework Architecture

In TEAM, TMan and the common memory are implemented as a blackboard system on top of GBB, a commercially available generic blackboard shell [Blackboard Technology Group, Inc., 1991]. When we discuss the TEAM framework, we are referring to the memory, TMan, and the agent-interface structures only. Agents are implemented separately and are not restricted to blackboard architectures.

We chose to use a blackboard shell for the underlying architecture of TEAM because: 1) we saw strong similarities between the philosophy of blackboard systems and multiagent problem solving; and 2) the blackboard architecture has many intrinsic features that are well-suited to multiagent systems. Blackboard systems have three major components: a database component that is responsible for storage of and efficient access to a set of centralized partitioned data; 2) a set of expert modules (*knowledge sources*); and 3) a control component that determines the scheduling and execution of the knowledge sources (KSs).

Building a blackboard system entails defining: 1) a domain-specific language of objects and their attributes; 2) a structured blackboard memory; 3) blackboard events that trigger the application of some expertise; 4) expertise (as encapsulated in knowledge sources); and 5) control mechanisms for determining the order in which expertise will be applied. The GBB shell provides the infrastructure for defining each of these. It provides an object-oriented database language for defining blackboard structures and objects and for specifying how objects should be placed on, accessed, and removed from the blackboard. It provides a knowledge-source language that enables the encapsulation of functional expertise and the specification of events that should trigger the application of that expertise. Finally, it provides several choices for control shells that implement different policies for ordering the execution of knowledge sources.

TEAM can be thought of as a shell for multiagent problem solving, just as GBB is a shell for blackboard problem solving. As such, it extends the basic GBB infrastructure in two ways: 1) adding detailed domain knowledge (where the domain is the integration and interaction of heterogeneous reusable agents); and 2) adding capabilities required to support autonomous agents. The first type of extension is accomplished by specifying the objects, blackboard structure, knowledge sources, event definitions, and control mechanism required to address the problem of heterogeneous-agent integration. For example, objects that are relevant to the framework include strategies, solutions, messages, and agent-interfaces. The second type of extension requires the implementation of additional capabilities related to the invocation of agents and interpretation of agent messages. For example, agents are invoked through a separate control function that has been added to the basic GBB control shell.

When an application system is built, the TEAM framework is further extended through the addition of application domain knowledge. Specifically, this consists of extending the object language and blackboard structure, supplying agents, supplying information about the agent set to the framework, supplying domain-specific global evaluation functions, defining acceptability and termination policies, and installing a default coordination strategy.

4.3.2 Agent Concurrency and Synchronization

During processing, there are two distinct phases: an agent-execution phase followed by a TMan-execution phase. These two phases are cyclically invoked until problem-solving termination. During the agent-execution phase, each agent is invoked sequentially and uses the information in common memory to select and order executable operators. It then applies its operators and, possibly, returns one or more

messages to TMan. After all agents have executed, the TMan-execution phase is invoked. TMan processes any messages that have been returned by the agents during the previous phase and uses that information to update the common memory. These updates then become the stimuli for the selection of operators and scheduling decisions in the next agent-execution phase. The TMan-execution phase uses a GBB control shell (the FIFO control shell) to invoke framework knowledge sources in response to agent messages. The next agent-execution phase begins when all knowledge sources that were triggered by either the original messages or by changes in the blackboard that were effected in response to those messages have been executed (in blackboard-system terms, this state is called *queue quiescence*).

In the current implementation of TEAM, the agent-execution phase involves a simple sequential invocation of each agent defined in the agent set. This invocation style is used because TMan and the complete agent set are executed within a single process on a single machine. However, changes to the common memory that occur in response to an individual agent's execution are not made until after the complete agent-execution phase has finished. Therefore, an agent will not see the results of another agent's work until the next agent-execution phase despite its place in the invocation order, thereby simulating agent concurrency from an individual agent's perspective as shown in Figure 4.3. The agent-execution phase is synchronized in

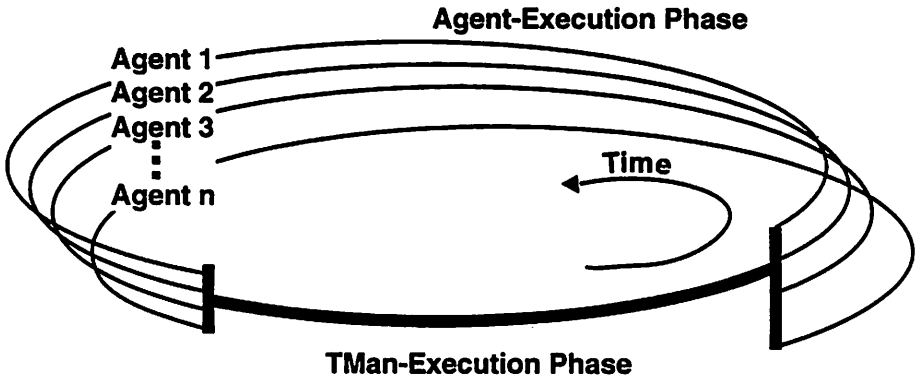


Figure 4.3. *The* TEAM *Execution Cycle*

that all agents' results are collected and processed sequentially by TMan during the TMan-execution phase. We believe that true synchronized concurrency of this type can be implemented trivially since agents already perceive this to be the case. Of course, synchronized concurrency does not take full advantage of available processing resources since all agents must wait for every other agent to finish executing during each agent-execution phase and no agent can execute during TMan-execution phases.

In future work, we will distribute agents into separate processes on different machines using the synchronization scheme above. Whether or not to implement asynchronous execution of agents is currently an open issue. In future work, we will develop an initial design for an asynchronous system and then decide whether or not there are issues that would be of general interest in extending the design to an implementation.

4.3.3 Common Memory

The common memory used by TEAM is a standard partitioned blackboard memory. The partitioning of the blackboard database and the specification of objects and attributes that can be stored in it define a common language for multiagent problem solving. Object instances on the blackboard describe something about the current state in problem solving. The TEAM blackboard partitions are:

- solutions
- problem specifications
- coordination strategies
- messages

In this section, we discuss each of these partitions.

Solutions: The solution partition contains partial and complete composite solutions that are available to all agents. These solutions can be accessed according to the values of their attributes. For example, a solution might be accessed by its current state of agent acceptability, its global acceptability, its initiating agent, its cost, its creation time, or its ID (a unique identifier). The generic TEAM *composite-solution* object is defined as shown in Figure 4.4.

The blackboard_path attribute shown in Figure 4.4 is used by TMan to store and retrieve objects. It is included in every object type. We will not include it in every object definition, however, because it is relevant only to internal framework functions.

Also, with respect to Figure 4.4, note that some attributes (e.g., cost) are not necessarily used consistently across all potential domains. Defining ontologies for representing information across domains is an area of active research [Guha and Lenat, 1990, Neches *et al.*, 1991]. We use a minimal set of attributes for generic TEAM objects that we have observed to appear in some context in many domains. These objects will be extended to include domain-dependent attributes for any application system built within the framework.

Some agents produce *proposals*, local solutions that represent either component solutions or complete solutions, depending on how solutions are decomposed and how

```
Composite_Solution
  Blackboard Path        % The blackboard path to specify how to
                         %    place these objects on the
                         %    blackboard
  Type                   % Solutions can be of different types, for
                         %    example, problem-specification or
                         %    solution
  Cost                   % The overall cost associated with a
                         %    solution
  Agents                 % The list of agents that have worked on
                         %   this solution to date
  Evaluations            % The list of agent/evaluation pairs for
                         %   this solution
  Agent Acceptability    % The solution's acceptability to all
                         %   agents that have currently evaluated
                         %   it
  Acceptability          % The global acceptability as defined by
                         %   the prevailing acceptability policy
  Component Set Completeness
                         % Does this solution have all its
                         %   components?
  Completeness           % Have all agents either proposed
                         %   components for or critiqued this
                         %   solution?
  ID                     % A unique identifier associated with this
                         %   solution
  Conflicts              % A list of conflicts associated with this
                         %    solution
  Creation Time          % The creation time of this
                         %    solution
  Initiating Agent       % The agent that developed the base
                         %    proposal for this solution
  Component Links        % Links to component proposals that
                         %   contribute to this solution
  Critique Links         % Links to critiques of
                         %    this solution
```

Figure 4.4. *A Composite-Solution Object*

the responsibility for components is assigned to agents in the agent set. These proposals are linked to composite solutions in common memory through the `component_links` attribute of a `composite_solution` object as shown in Figure 4.4. Proposals reside in an agent's private memory but are available to all agents once they become part of a composite solution through the link mechanism. Through the link, an agent can examine an explicitly shared proposal of another agent without gaining access to the full private database of the agent. A *proposal* object is shown in Figure 4.5. A

```
Proposal
  Initiating Agent        % The agent initiating this proposal
  Evaluation              % The agent's local evaluation of its
                          %   own proposal (can use any local
                          %   rating scheme)
  Acceptability           % The agent's current acceptability
                          %   rating of the composite solution:
                          %   must be acceptable, unacceptable,
                          %   or infeasible
  Cost                    % Cost for this proposal
  Conflicts               % A list of conflicts associated with
                          %    this proposal
  Parameters              % A list of parameters whose values are
                          %   specified in this proposal
  Creation Time           % The system time at which this
                          %   proposal was created
  Composite Solution      % A link to a composite solution
                          %   containing this proposal
```

Figure 4.5. *A Proposal Object*

critique is very similar to a proposal except that it does not include any attributes that are related to specifying a component. It is shown in Figure 4.6.

Another type of object that can reside on this space is a `conflict`. `Conflict` objects are used to describe detected conflicts, transmit information about violated solution requirements, and, optionally, to suggest specific resolution methods. A conflict object is shown in Figure 4.7.

Problem Specifications: The problem-specification partition contains a representation of user-defined conditions that must be satisfied by any solution or of conditions that exist in the environment that must be considered during solution generation by the agent set. The problem specification object includes a subset of the attributes of a complete solution that are constrained by the user. For example, a typical

```
Critique
  Critiquing Agent        % The critic agent
  Evaluation              % The agent's evaluation of the
                          %   solution (can use any local
                          %   rating scheme)
  Acceptability           % The agent's current acceptability
                          %   rating of the composite solution:
                          %   must be acceptable, unacceptable,
                          %   or infeasible
  Conflicts               % A list of conflicts associated with
                          %   the composite solution
  Creation Time           % The system time at which this
                          %   critique was created
  Composite Solution      % A link to the composite solution
                          %   being criticized
```

Figure 4.6. *A Critique Object*

```
Conflict Object
  Originator              % The agent that detected the conflict.
  Violated Requirements   % Requirements that are in direct conflict
                          %   with the proposal being evaluated.
  Parameter Name/Value List
                          % A list of conflicting parameters and
                          %   their assigned values (used for
                          %   determining relevance by the receiving
                          %   agent).
  Proposal                % A link to the proposal that triggered
                          %   the conflict.
  Solution                % A link to the composite solution
                          %   associated with this conflict.
```

Figure 4.7. *A Conflict Object*

problem specification in the STEAM system provides specific values for some subset of the attributes of a complete design as shown in Figure 4.8.

```
Problem_Specification
  Type:  problem_specification
  Required_capacity: 736
  Maximum_platform_deflection: 0.03
  Platform_side_length: 185
```

Figure 4.8. *A Problem-Specification Object*

Coordination Strategies: This partition contains information about suggested and instantiated strategies, either for default problem solving or for customizing a particular agent interaction. It provides a communication medium which agents use to exchange information that will allow them to converge on an appropriate strategy.

Suggested_strategies are proposed by agents given their local views of some required agent interaction (for example, local knowledge about the characteristics of their own search and solution spaces may be important in suggesting a conflict-resolution strategy for a particular conflict). The generic TEAM suggested_strategy object is shown in Figure 4.9.

```
Suggested_Strategy
  Originating_agent    % The agent that suggests the strategy
  Strategy_name        % The name of the strategy
  Roles                % The operators that the agent can apply
                       %   for this particular strategy
  Event                % The event that triggered the
                       %   strategy selection mechanism
                       %   (e.g., a specific conflict)
  Instantiation        % A link to an instantiated-strategy object
```

Figure 4.9. *A Suggested Strategy*

Multiple agents may propose strategies for achieving coordinated agent interaction in a specific context. Currently, TEAM provides a centralized selection mechanism for choosing among the set of suggested strategies and assigning the subtasks associated with each strategy to an agent in the agent set. This centralized mechanism will

be described in detail in Chapter 7. However, the task allocation functionality could itself be fashioned as a distributed-search problem, perhaps using a bidding protocol such as the contract net protocol [Davis and Smith, 1983].

An *instantiated_strategy* object represents information about the state of a strategy that has been instantiated within an application system. The generic TEAM instantiated_strategy object is shown in Figure 4.10.

```
Instantiated_Strategy
   Strategy_name       % The name of the strategy in effect
   Manager             % The name of the agent managing this
                       %   strategy (primarily used in the
                       %   strategy selection process)
   Type                % Default, custom, or conflict-resolution
   Solution            % In the case where type is
                       %   conflict-resolution, a link to
                       %   the composite solution being
                       %   developed under this strategy
   Agent_role_pairs    % A list of the agents involved in the
                       %   strategy along with the roles
                       %   they play in this instance
   Event               % A link to the specific event that
                       %   triggered the strategy selection
                       %   (unless this is a default strategy)
   Activation State    % Either active or inactive
   Search State        % Some strategies may have multiple phases,
                       %   e.g., open and closed, or rough and
                       %   optimize
   Suggested Strategy  % A link to a suggested-strategy object
                       %   unless this is a default strategy
```

Figure 4.10. *An Instantiated-Strategy Object*

Messages: GBB provides efficient access and retrieval capabilities for objects on the blackboard. However, because agents do not know the specifics of blackboard retrieval, they do not directly access objects in the common memory (with the exception of objects in the message partition). Instead, all notifications, retrievals and updates are performed by TMan under the direction of the agents. When an event occurs that an agent may be interested in, TMan notifies the agent and provides it with a pointer to any objects that are involved in the event. Therefore, the agent does not need to know the path specifications to the objects in order to view them.

When an agent wants to add or update an object in common memory, it sends a request to TMan to perform the change. All interactions take place in the form of messages that are stored in the blackboard. When an agent is added to the agent set, it is given the specific path for its own message buffer where it can retrieve incoming messages and the path for the TMan message buffer where it can send messages.

The types of messages defined in a particular application system are domain- and coordination-protocol dependent. The generic TEAM message object is defined in Figure 4.11.

```
Message
   Type                  % the type of message
   Content               % content of message (dependent
                         %   on message type)
   Sender                % the agent sending the message
   Recipient             % the agent(s) intended to receive
                         %  the message
   Time-of-transmission  % the time the message was sent
```

Figure 4.11. *A Message Object*

An example message type from the STEAM system is *initiate-solution* which is sent by an agent to TMan. An instance of this type of message contains a component solution that the agent wants to propose as the basis for a design. TMan will respond to this message by creating a design containing the proposed component and broadcasting a *solution-initiation* message to let other agents know that a new design has been created. The solution-initiation message contains a pointer to the new design so that any agent interested in extending or critiquing the design will be able to examine it directly.

4.3.4 The TEAM *Manager*

The TEAM manager (TMan) comprises a set of framework knowledge sources (KSs) that respond to agent requests and events in common memory by updating objects in the memory and sending notifications to agents. In addition to the set of knowledge sources, TMan includes a control component for scheduling and invoking knowledge sources and one for invoking agents. The control component used for KS invocation in the current TEAM framework is the FIFO control shell of the GBB system. In this simple control shell, KS instantiations are invoked in

the order in which they are triggered. Control of problem-solving in a traditional blackboard system can be very complex and the activation of KSs occurs in an opportunistic manner. This means that at each point where a control decision can be made, the system attempts to execute the KS that will advance the problem solving situation the most. This is usually a subjective decision that is based on the existence and certainty of data, the type, reliability, and relative cost of possible KSs, and potentially other factors [Corkill, 1991, Erman *et al.*, 1980, Hayes-Roth, 1985, Nii, 1986]. However, the GBB FIFO shell has the advantage of simplicity and has proven to be quite adequate for the current implementation of TEAM. One of the reasons it is adequate is that it is not necessary to choose a subset of KSs to execute in order to meet time or complexity constraints on problem solving and instead it is possible to execute all triggered KSs.

4.3.4.1 TMan Knowledge Sources

TMan has a set of basic operations that it performs in response to specific events that occur at the agent level. These operations are encapsulated in knowledge sources that specify both the conditions under which the knowledge should be applied and the knowledge itself. When an agent wants a change to be made in common memory, it notifies TMan by sending a message that details the type of change and whatever information is required to effect the change. When TMan processes the message, it will perform whatever actions are needed to make the specified changes. The changes may stimulate other TMan knowledge sources to perform some action. For example, an agent may ask TMan to extend an existing solution with a proposed component it has developed. This request triggers a knowledge source that will link the component to the solution and update any solution parameters that are affected by the extension. Adding the component then becomes the trigger for a knowledge source that checks if the solution is complete. If it is, the knowledge source will change the value of the `completeness` attribute of the solution to *complete*. This change triggers a knowledge source that checks for solution acceptability, and so on. `TEAM` KSs also map values from component-proposal attributes to solution attributes (as discussed in Section 3.2). For example, in a multi-component system, each component may have an associated cost. These component costs are often added together to determine the total cost of a solution. The `TEAM` framework can perform simple transformations on parameters such as addition, subtraction, set union, and set intersection. However, if more complex transformations are required, they must be done by an expert agent.

A small set of domain-independent abstract knowledge sources have been developed for `TEAM`:

respond-to-new-proposal A KS invoked to build a new solution or to extend an existing solution in response to the receipt of a new proposal from an agent.

calculate-global-utility A KS invoked to calculate the global utility of a solution in response to the completion of a solution.

update-acceptability A KS invoked to update the acceptability of a global solution in response to a change in some agent's local acceptability rating.

evaluate-solution-completeness A KS invoked to evaluate the completeness of a solution in response to the addition of some component or specification of an attribute.

change-coordination-strategy A KS invoked to change the coordination strategy in response to either a new conflict or to the occurrence of some event in the current strategy.

evaluate-solution-acceptability A KS invoked to determine whether a specific solution meets the solution-acceptability criteria defined by the application developer.

evaluate-termination-state A KS invoked to check whether the termination criteria for the system is satisfied in response to some specified event such as the completion of a solution.

Although these knowledge sources abstractly describe functional requirements necessary for multiagent problem solving across different application domains, the details of what events trigger a KS and exactly how the KS responds are always domain-dependent. For example, the ability to respond to an agent's proposal, *respond-to-new-proposal*, is required in any multiagent domain. However, a new proposal could be a complete solution (as in the `AGREE` domain) or it could be a component solution (as in the `STEAM` domain). The new proposal might initiate a new composite solution or it might extend an existing solution. Some application systems might break the functionality of this KS abstraction into several KSs that operate on different types of proposals. In Figure 4.12, we illustrate some of the agent-response KSs required by the *negotiated-search strategy* described in Chapter 5. The *respond-to-new-proposal* abstract KS from above is broken down into three implemented KSs: *build-new-solution*, *merge-component-proposal*, and *merge-critique*. *Build-new-solution* is invoked in response to a base proposal created by an agent, that is, a proposal intended to be used as the basis for a new solution. *Merge-component-proposal* is invoked in response to the creation of a component proposal intended to

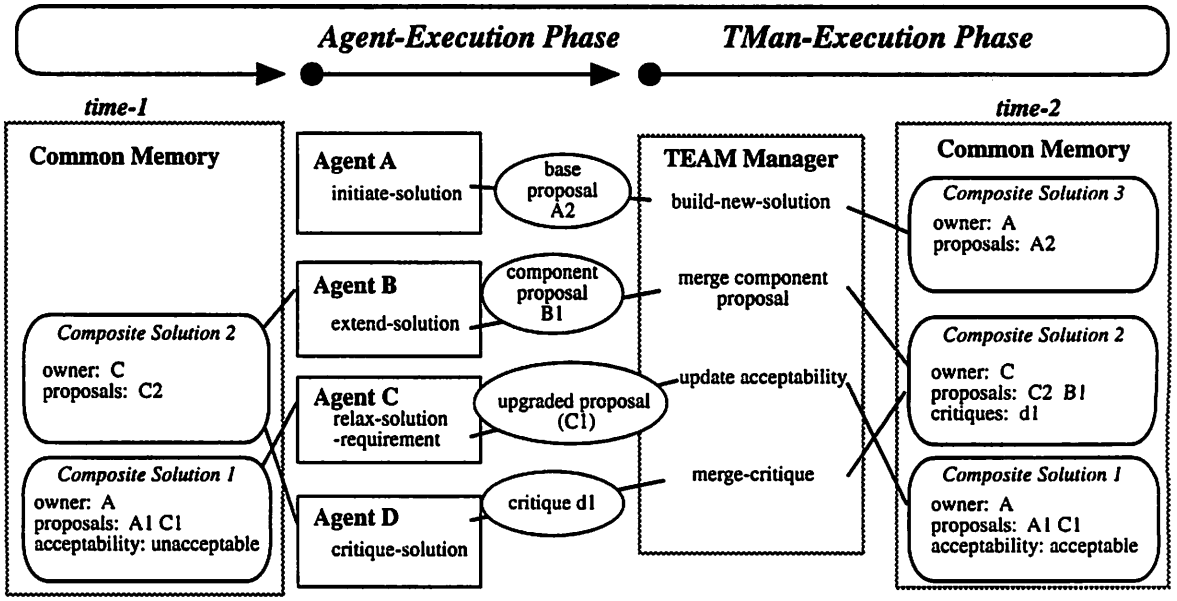


Figure 4.12. TEAM *Knowledge Sources for Negotiated Search*

be merged with an existing proposal. *Merge-critique* is invoked to join a criticism to its existing solution. Finally, the *update-acceptability* KS is invoked when an agent relaxes a local solution requirement that changes its local acceptability rating on an existing solution. This acceptability change must then be propagated to the global solution since this solution may now be acceptable.

The acceptability policy for solutions is defined in the *evaluate-solution-acceptability* KS. As described earlier, different application systems may wish to implement different acceptability policies. For example, the default acceptability policy for STEAM is that a complete solution is acceptable if every agent rates the solution as locally acceptable and the system cost is below some user-defined threshold. However, other policies are reasonable, e.g., it might be desirable to have one agent's opinion count more than another.

The termination policy for the system is defined in the *evaluate-termination-state* KS. The default termination policy for STEAM is that when three acceptable solutions have been found, the initiation of new solutions is terminated. Existing acceptable solutions are finished however, and processing ends when there are no acceptable partial solutions remaining. Another possible policy would be to simply stop processing as soon as some number of acceptable solutions are found without completing partial solutions. In Chapter 10 we will discuss a 2-stage termination policy for the STEAM domain that includes a distinct optimization phase. In that policy, first a number of draft designs are created, then those designs are optimized, and then the system terminates.

In addition to the abstract KSs described above, a particular application system may require more esoteric KSs designed to handle specific features of the search algorithm, coordination protocol, or domain of expertise.

4.3.5 Agent Sets

An agent is connected to the framework through a specification of pertinent information contained in an `agent_interface_object` as shown in Figure 4.13:

```
Agent-Interface-Object
  Name                  % A unique identifier for the agent
  Invocation Procedure  % The name of the entry procedure for
                        %    the agent
  Solution Parameters   % Names and descriptions of parameters
                        %  in the agent's solution space
                        %  (output parameters)
  Search Parameters     % Names and descriptions of parameters in
                        %   the agent's search space  (input
                        %   parameters)
  Relaxation Threshold  % The number of agent invocations before
                        %   an automatic relaxation
  Strategy/Operator Set % The set of strategies and operators
                        %   that can be applied by this agent
  Termination Phase     % The agent's view of the current state
                        %   of termination
  Agent Message Buffer  % The path specification for the agent's
                        %   message buffer
  TEAM Message Buffer   % The path specification for the TMan
                        %   message buffer
```

Figure 4.13. *An Agent-Interface Object*

Agents are distinct entities that communicate through the common memory. Each agent is invoked by TMan during each agent-execution phase and executes until it has nothing left to do or until it has some information to communicate. When it has finished executing, any required messages are sent to TMan and the agent hibernates until it is invoked again. Agents maintain persistent private databases that may change in response to environmental stimuli.

An agent's internal language may be a subset or superset of the common language, or it may be different enough from the common language that explicit interpretation

is required for communication. Agents within a set have no built-in knowledge about other agents, but they may accumulate information as processing progresses.

Each agent has control of its own local actions: there is no central scheduler to determine the ordering of operator applications within an agent. Within the context of a specific strategy that is currently in effect, each agent has an assigned role that translates into a set of potentially applicable operators. As a simple example, assume that some agent (Agent $\mathcal{A}$) is a "critic" for some set of solution parameters of a composite solution. For example, let Agent $\mathcal{A}$ criticize the solution parameter *water-flow-rate* from the **STEAM** system (as described in Chapter 6). As a critic, Agent $\mathcal{A}$ has two operators it can potentially apply: *critique-solution* and *relax-solution-requirement*. *Critique-solution* evaluates a composite solution within the agent's local context and returns a rating and acceptability value. *Relax-solution-requirement* chooses a constraint to relax, performs the relaxation, and propagates the effects of the relaxation throughout the local databases and common memory.

In order for Agent $\mathcal{A}$ to apply its operators to a particular solution, all appropriate preconditions must be satisfied. As will be discussed in Chapter 5, the search strategy itself defines a set of solution-state preconditions that hold across all agents. For this example, assume that the solution-state preconditions for applying the operator *critique-solution* are that the composite solution being criticized must have the value **acceptable** for its **acceptability** attribute, it must have the value **incomplete** for its **completeness** attribute, and Agent $\mathcal{A}$ must not be a member of the value of its **agents** attribute (see the definition of a composite solution in Section 4.3.3 earlier in this Chapter).

In addition to the solution-state preconditions above, there may be domain preconditions that are specified in the definition of the agent. In the **STEAM** example from above, Agent $\mathcal{A}$ specifies a precondition that in order to apply its *critique-solution* operator, the **water-flow-rate** attribute of the composite solution being criticized (a composite solution is a steam condenser design in the **STEAM** system) must have a value.

Although the internal scheduling mechanisms of an agent are not mandated by the framework, each agent must be able to respond to events that alter the coordination space. For example, if the value of *acceptability* attribute of a composite solution changes from *unacceptable* to *acceptable*, agents may wish to respond by extending that solution. Here, we describe one method for ensuring that agents respond in a timely manner to global changes. Notice, however, that it is the functionality that is important rather than the detailed implementation mechanisms that are used to achieve that functionality.

When an event has occurred that triggers the agent to examine a particular solution, if the solution-state preconditions of a particular operator are satisfied in that solution, the operator becomes *triggered* and is placed on the agent's local list of *pending* operators. For example, a message from TMan that a new composite solution has been added to the common memory might cause an agent to examine the new solution and trigger a *critique-solution* operator. However, before actually executing the operator, the agent must determine if any domain preconditions exist that remain to be satisfied. In our example, an `incomplete` and `acceptable` solution may have been created, but no value for the `water-flow-rate` attribute of that solution is specified as of yet. Therefore, the triggered instance of *critique-solution* will be placed on the *pending* list until that precondition is satisfied.

The satisfaction of solution-state and domain preconditions must be determined separately, although simultaneous fulfillment of both is required in order for an operator to actually be applied to a solution. Each agent locally defines its domain-dependent preconditions and these are not known to the `TEAM` manager. Therefore, TMan cannot be responsible for notifying the agent when those preconditions are fulfilled. Instead, TMan notifies agents when solution-state preconditions of generic operators are fulfilled, e.g., an `incomplete`, `acceptable` solution is placed in common memory. Once an agent is notified of an event in common memory, if it not in a position to react immediately, it must not "forget" to react at the appropriate time. Unlike centralized systems which maintain a global agenda of pending actions, in an agent-based system, each agent is locally responsible for remembering its own pending actions.

The satisfaction of solution-state preconditions can change while an operator is *pending*, awaiting satisfaction of its domain preconditions. In our example, another agent (Agent $\mathcal{B}$) may supply a value for the `water-flow-rate` attribute, thereby fulfilling the domain precondition of Agent $\mathcal{A}$. However, Agent $\mathcal{B}$ may also rate the composite solution as `unacceptable`, thereby changing the `acceptability` attribute of the solution. In this situation, Agent $\mathcal{A}$'s *critique-solution* operator cannot be executed now due to solution-state precondition violations. In this case, however, Agent $\mathcal{A}$ can remove the operator from the list of *pending* operators completely. The difference here is that when solution-state precondition fulfillment changes, TMan will send a notification. In other words, if the solution becomes `acceptable` again at some later time, TMan will notify Agent $\mathcal{A}$. However, TMan will not notify the agent if the status of some domain precondition changes.

If both the solution-state and domain preconditions of an operator are satisfied simultaneously, the operator becomes *executable*. A record of information relevant to

the specific context in which the operator is to be applied is created and placed on an *executable* agenda. We will call this record an Operator-Activation-Record (OAR), which is analogous to a KSA or KSI from the blackboard literature. Throughout the dissertation, when we refer to the "execution of an operator" or "invocation of an operator", it is more precisely the invocation of an OAR than of the operator itself. An OAR can be thought of as an operator wrapped in a specific context.

The order of invocation of executable operators is controlled locally by agents. Becoming executable does not guarantee that an operator will be invoked immediately or even that it will be invoked at all. At any given time, an agent may have multiple operators that have become executable because of recent events and it may have multiple operators that were already executable but that have not yet been invoked. The agent locally determines which operator or operators it will execute during this agent-execution cycle and in which order those operators will be executed.

The order in which an agent schedules local operators is not mandated by either TEAM or by the search algorithm in effect. However, the order in which particular operators are executed does affect problem-solving performance and, therefore, the effect of local scheduling on the overall behavior of the agent set should be considered. Some general policies for local scheduling are useful in most situations, i.e., agents should assimilate any new information received before initiating or critiquing solutions. The degree of sophistication required in local scheduling though is highly dependent on the application and the complexity of required interactions.

4.4 Building Systems of Heterogeneous and Reusable Agents

The experience of designing and implementing a multiagent architecture and application systems has led us to some general tenets that we believe will hold over a wide range of applications. In this section, we describe what we've learned about building systems.

4.4.1 System Components

If someone was going to sit down and build a system of heterogeneous and reusable agents, how would they start? Table 4.1 gives some initial definition to the problem by listing the components that comprise a system, along with some indication of their scope and interdependencies. In Table 4.1, we show some of the components that must be included in an application system, along with information about how that component fits into the overall system. The scope of a component indicates whether this component is available globally to the entire system (*system* level) or is privately

Table 4.1. *System Components in a Multiagent Architecture*

System Component	Scope	Dependencies
Strategy Specification	system	strategy search operators
Strategy Instantiation	system	strategy agent-set search operators
Termination Policy	system	domain* agent-set* strategy*
Acceptability Policy	system	domain* agent-set* strategy*
Local Search Functionality	agent	domain
Search Operators	agent	strategy domain
Local Evaluation	agent	domain strategy*
Local Control Policies	agent	domain* strategy*

maintained at an agent (*agent* level). The *Dependencies* column indicates what kind of dependencies this component has (what must be known in order to design the component). When an item in the dependencies column is followed by asterisk, the dependency is not an explicit requirement of the system, but may be incorporated as the item is tailored to the specific system.

To illustrate the table's meaning to a system developer, first examine the *strategy specification* row. A strategy specification includes a transition-network definition of a strategy and an abstract description of the operators. A strategy specification is strategy dependent of course, but it is domain independent since it is defined in the coordination space of a system rather than the domain-level solution space. The specification is also agent-set independent since it does not make explicitly reference any agent, but rather is specified in terms of search operators.

The assignment of operators to agents is accomplished through a *strategy instantiation* that establishes the agent/role assignments for a given strategy and agent set. This strategy instantiation therefore provides agents with knowledge about which

operators will be locally active. A strategy instantiation is strategy-dependent and agent-set dependent and operator-dependent.

Not all components are as cleanly delineated as the strategy specification. This can be seen in the asterisked items such as those shown in the *acceptability policy*. The domain, strategy, and agent set are all asterisked in this case, meaning those items can be either dependent or independent. For example, an acceptability policy is domain-independent and agent-set independent if it states that a solution is acceptable when all agents find it locally acceptable because no specific information about the domain or about what agents exist in the system is required to implement this policy. However, if the policy states that a solution is acceptable when all agents find it locally acceptable *and* when a global evaluation of the solution is above some threshold, the global evaluation and the threshold value are domain dependent. Similarly, if the policy states that a solution is acceptable when agents A and B find it acceptable and agent C finds it feasible, this policy is agent-set dependent by definition and is also implicitly domain dependent since the decision about which agents are important is undoubtedly based on domain-level information. A general heuristic guideline for dependencies in acceptability policies is that globally cooperative systems are likely to have domain and agent-set dependencies and locally cooperative systems are less likely to have them. This makes sense intuitively since agents in a locally cooperative system worry mainly about pleasing themselves while agents in a globally cooperative system are concerned about integrated results.

4.4.2 Using Existing Software to Build Agents

An agent can be either an existing piece of software, modified to work within an agent set, or it can be a piece of software specifically designed as a reusable agent to work within an agent set. The first case is more problematic. To incorporate pre-existing software into a reusable agent in a TEAM application system, the existing search functionality in the software must be *wrapped* in distributed-search operator shells that obtain the required inputs, execute the appropriate local search functions, and produce the defined outputs. The software must be converted to agent status by providing explicit entry and exit points, local control of operator execution, explicit declaration of interface information, memory objects that maintain ongoing context about coordination of problem-solving, and memory objects that provide context about the evolving view of the composite solution space. Existing software is unlikely to be able to assimilate and apply information from other agents, particularly if the information is potentially inconsistent with local knowledge. Therefore, the agent's local information-handling mechanisms may need to be extended or replaced. Although

these tasks are non-trivial, the STEAM system described in Chapter 6 successfully uses pre-existing code as the basis for all domain-level problem solving. Through our experience with that system, we feel confident that it is possible to incorporate software that is not explicitly written for inclusion in the TEAM framework.

4.5 Summary

In this chapter, we describe the generic framework we have created to support the development of reusable-agent search systems. The framework is flexible, application-independent, and search-strategy independent, while providing the communication and infrastructure mechanisms needed to enable appropriate agent interaction. We discuss the management and partitioning of information within the TEAM framework to ensure privacy and to permit agents to operate and integrate their results without detailed implementation knowledge about the system or agent set. We also describe motivating factors in the design of the framework including such issues as the selection of a centralized memory, maintaining a level of access indirection between agents and the memory, agent concurrency, declarative representations of evaluation, acceptability, and termination policies, and declarative representations of search strategies and agent/role assignments within a strategy.

The TEAM architecture is described in terms of its three primary components: 1) common memory; 2) the framework manager; and 3) the agent set. Both the basic architecture and the extensions that are required in order to build an application system on top of the framework are detailed. Important concepts related to the architecture include the definition of a framework language for defining objects and relationships, the specification of how objects are stored, retrieved, and extended to accommodate domain knowledge, the ability of agents to communicate with the framework manager, and the ability of agents to respond to events in the common memory. The current version of the architecture resides in a single process and, therefore, does not support true agent concurrency. The mechanisms used to support apparent concurrency and possible future enhancements to actual concurrency are presented.

In the final section of this chapter, we present some initial guidelines for building systems of heterogeneous reusable agents. We specify the components of a system and the information required for the development of each component. Finally, the conversion of existing software for use in a reusable-agent system is discussed.

A long-term goal for multiagent system-building is to have a library of strategies and agents and to be able to select, rather than create, the most effective strategy and agent set for the problem at hand. To realize this goal, every strategy must

include a specification of the problem characteristics for which it is most suitable. Likewise, every agent must explicitly declare its capabilities, knowledge, and other characteristics that may influence its effectiveness within an application system. The next step in the evolution of reusable-agent search systems, therefore, should be the development of a taxonomy of strategies and problem characteristics that would guide a system developer in selecting a strategy if a suitable one exists in the library and a taxonomy of agent characteristics that will guide the arrangement of agents into sets. Currently, we can't even enumerate what these characteristics are. Classification of problems as globally or locally cooperative is a start along with classifications of evaluation functions as linear, monotonic, etc. as described in Chapter 7. Klein [Klein, 1991] has done some initial work on identifying the characteristics of specific conflict situations and associating them with resolution procedures. The formal models of Zlotkin and Rosenschein [Zlotkin and Rosenschein, 1991] offer some preliminary insights into types of cooperation that are feasible in particular situations. There is much work to be done however both in identifying the important features of problems and agents and in mapping those features to appropriate problem-solving strategies.

CHAPTER 5

NEGOTIATED SEARCH

A distributed-search strategy is a specification of search operators and the applicability conditions of those operators that is used to define and coordinate the actions of a set of agents. *Negotiated search* is a flexible and widely applicable distributed-search strategy that specifically addresses issues that arise in multiagent systems comprising reusable and heterogeneous agents. Negotiated search acknowledges the inevitability of conflict among the agents, and exploits that conflict to drive agent interaction and guide local search. It treats conflict as an integral part of problem solving and as a source of control information for agent communication.

Negotiated search incorporates some very basic types of conflict management. Conflict avoidance is inherent in the mechanisms agents use to exchange, assimilate, and apply non-local information. Conflict resolution appears in several forms, including both extended-search and requirement-relaxation mechanisms. The iterative application of conflict-management techniques is often called negotiation: therefore, we motivate the negotiated-search strategy with an intuitive description of negotiation. This introductory-level motivation is followed by an extended example of negotiated search in a parametric-design application. The example is taken from the **STEAM** system, a prototype system that performs parametric design of steam condensers. A detailed description and evaluation of the **STEAM** system will be provided in Chapter 6, however, this example can be understood from perspective of distributed search without a full understanding of the system. The example precedes an abstract, domain-independent description of negotiated search that exemplifies the representation and theory underlying our discussion of distributed-search strategies in general, and negotiated search in particular. The abstract description is intended to give the reader a sense of how negotiated search relates to the intuitive model of negotiation and to move beyond the preconceptions of both the intuitive model and those brought by the reader to a realistic model of negotiated search among heterogeneous agents. Negotiated search is described from two viewpoints: 1) a transition-network view of how solutions progress from an initial state to a termination state; and 2) detailed descriptions of the negotiated-search operators required by the strategy. Finally, we discuss control of negotiated search from the local agent perspective.

5.1 Negotiation: An Initial Perspective

As we first began our investigation of negotiation in distributed search, our intuitive definition of negotiation went something like this:

> One agent generates a proposal and other agents review it. If some other agent doesn't like the proposal, it rejects it and provides some feedback about what it doesn't like. Some agent may generate a counter-proposal. If so, the other agents (including the agent that generated the first proposal) then review the counter-proposal and the process repeats. As information is exchanged, conflicts become apparent among the agents. Agents may respond to the conflicts by incrementally relaxing individual preferences until some mutually acceptable ground is reached.

Over the course of this research, it has become apparent that there are many behaviors that can be classified as negotiation behaviors. This particular example, however, captures the primary characteristics that one would expect to see:

- proposals are generated by one or more agents
- agents evaluate proposals based on their individual criteria for solution acceptability
- agents provide feedback about what they like or don't like about particular proposals, resulting in a progressively better understanding of the shared requirements for solutions over time
- agents can play different roles in the negotiation process, e.g., an agent can be a reviewer for another agent's proposal and then be a generator for a counter-proposal
- conflicts exist among the agents' requirements for acceptable solutions
- agents incrementally relax their solution requirements to reach agreement
- the decision to accept or not accept a proposal is a joint, democratic process

We are primarily concerned with negotiation situations where the participants are either locally cooperative or globally cooperative, non-hostile, and willing to share information. Under these assumptions, we have developed a flexible and general paradigm that will support the types of behavior needed to negotiate in many different situations. We begin by examining ways in which the intuitive definition of negotiation is imprecise.

What aspects of the definition must be made more exacting if we are to allow a broad range of activities as discussed above? First, the definition assumes that a proposal and a solution are basically the same and that a proposal becomes a

solution when the proposal is accepted by all agents. However, this assumption rules out situations in which high-level tasks are decomposed and each agent is working on some subtask. In this case, the proposal an agent makes does not represent a complete solution but rather some component of a solution; complete solutions comprise multiple interacting component solutions. Evaluating another agent's proposal is not straightforward in this situation. If an agent has specialized knowledge that allows it to only work on particular subproblems, how can it evaluate proposals that represent solutions to interacting subproblems outside of its domain of expertise? It is often the case that an interacting proposal generated by another agent will have a major impact on the quality of possible local solutions, but cannot be evaluated directly.

We will present an answer to the question and demonstrate the viability of our solution in the STEAM application, which is characterized by interacting component proposals. To briefly summarize the solution we have implemented, an agent evaluates an externally-generated interacting proposal outside of its own domain by generating a compatible local proposal and evaluating the local proposal. The evaluating agent does not attempt to say anything about the quality of the external proposal. Instead it focuses on how that proposal affects local quality, i.e., the quality of the best compatible local proposal.

Returning to our discussion of intuitive definition of negotiation, a common-sense perception of negotiation would dictate that there should be some relationship between a proposal and a counter-proposal. However, this relationship is not directly based on the proposals themselves. Assume that a proposal is either a solution or a partial solution as discussed above. From this perspective, a solution can be represented as a set of attribute/value pairs:

```
Solution A
   Attribute1:  Value1
   Attribute2:  Value2
   Attribute3:   ...
```

For example, in negotiating for a house, a proposal might have the form:

```
Purchase Agreement
   Home Model: Ranch
   Price: $200,000
   Seller: Jack
   Buyer: Jane
   Color: ...
```

Although a proposal of this type includes all the information required to implement a solution, it provides only a surface-level view of the reasoning that went into generating it. In some domains, it might be possible to make guesses about other agents' requirements that could be used in generating counter-proposals. For example, in negotiating the sale of a house, the seller can guess that the buyer will be unhappy with a higher price and can try to balance that with some other consideration. However, in the general case of heterogeneous agents, external local evaluation criteria for solutions cannot be predicted, nor can they be inferred from the "snapshot" provided by a proposal. If proposals and counter-proposals are related, this relationship must come from some deeper understanding of the shared search space of the agents. This understanding is achieved through a feedback system that can be separate from the actual proposals. Sathi and Fox recognized the underlying goal of feedback in their work on constraint-directed negotiation [Sathi and Fox, 1989]:

> Constraint-directed negotiation would simply reduce to heuristic search if constraints were used only for evaluating negotiation positions. In constraint-directed negotiation we view constraints as fundamental in the generation of new positions.

To put this in a more generic context, when information about local solution requirements is exchanged and used to guide the generation of new local proposals, the search process becomes stronger and more focused. In fact, when appropriate information is exchanged as feedback, it is often possible to avoid future conflicts. The first and best line of defense in conflict management is conflict avoidance. As will be seen in negotiated search, conflict avoidance occurs as a result of a feedback system that runs in parallel with the generation and extension of solutions.

Returning to our intuitive definition of negotiation, other issues that must be addressed include:

- Which agent(s) should generate counter-proposals?
- When feedback information or counter-proposals are communicated, which agents should receive that information and what should they do with it?
- If multiple agents send conflicting feedback information, how do those conflicts get resolved and by whom?
- How do conflicts become apparent?
- If there are multiple methods that can be used to resolve a conflict, which one should each agent use? How can it let other agents know what it's doing?
- What makes an agent decide to relax a local restriction?

- How can an agent relax a local restriction?
- How is mutual acceptability defined and recognized?
- Should the agents concentrate on developing a single solution, or should they consider multiple alternatives simultaneously?
- When multiple agents have constraining requirements on a solution, which agent should be in charge of developing the solution?

These issues are all made more difficult because agents are heterogeneous and cannot depend on receiving a particular type of information, on having similar knowledge or problem-solving styles, or on knowing the same search strategies or negotiation protocols. Specific solutions to each of the questions above have been formulated for the application systems, **STEAM** and **AGREE**, and will be discussed in relationship to the individual systems.

In negotiated search (as implemented with the **TEAM** framework), the control strategy is a multipath incremental-extension algorithm. This means that 1) each solution is initiated by a single agent during some system processing cycle and then extended and evaluated by other agents during future processing cycles; and 2) multiple solutions are being constructed simultaneously.

Incremental Solution Extension: When a solution has been initiated by some agent, Agent 1, and another agent, Agent 2, is attempting to extend it, Agent 2 may not be able to assign values to its output parameters that don't conflict with attribute values assigned previously by Agent 1. In this case, there are two possible outcomes: 1) if the conflict is caused by the violation of some hard (non-relaxable) requirement, the hard solution requirement is communicated and this solution path is pruned; or 2) if the conflict is caused by the violation of some soft (relaxable) solution requirement, the requirement is communicated and the solution is saved and viewed as a potential compromise. In the first case, no more work will be done on that solution, and, to the extent that the violated requirement can be understood and assimilated by other agents, future counter-proposals will not violate that same requirement. In the second case, the violated requirement may eventually be relaxed and, if that happens, the potential compromise will become a viable solution again. Future counter-proposals will take the violated requirement into account but are not guaranteed to avoid the same conflict, since other alternatives may be worse.

In either of the two cases above, conflict is used as the trigger for the communication of local constraining information. In multiagent systems, it is always problematic to decide what information should be exchanged and when that exchange should take place. In general, agents want to minimize the amount of information

they share since it may be expensive to communicate information and it is always expensive to assimilate information. On the other hand, sharing information that will specifically help another agent avoid future conflicts is generally cost effective since it eliminates the expense of generating unproductive solution paths (this is demonstrated in Chapter 8). In negotiated search, an agent that receives conflict information from another agent can choose whether or not to prune its own search to respect that information. When an agent detects a conflict (either when trying to extend the solution with a proposal or when criticizing the solution), it provides feedback with a *conflict* object (shown in Figure 4.7).

Multipath Processing: Because the ultimately acceptable thresholds for solution attribute values are not known to any agent at system start-up time, maintaining multiple solution paths has an important advantage. Under a single-path algorithm, solutions that cannot be extended are pruned despite the possibility that, as problem solving progresses, local relaxations may occur that will make them extendable. In fact, it is sometimes the case with backtracking systems that include relaxation that no feasible solutions are found because the required relaxations occur after all feasible solutions are already pruned. With negotiated search, not only do those solutions remain available, but by keeping solution paths with different constraint violations open, it becomes possible to explore the effects of relaxing one constraint versus relaxing some other constraint.

There are also disadvantages to concurrently exploring multiple solutions paths. Most obviously, there will be multiple partial solutions that have to be stored at all times, requiring additional memory resources. There is also an overhead cost for the processing required to focus on a promising solution path at a particular point in problem-solving, to determine when to open and close paths, and to manage the links between solution components along each path. Since the number of open solution paths is highly dependent on the domain, the number of agents, and the problem-solving strategy in effect, we will discuss these issues more in the context of each of the application systems presented in Chapters 6 and 7.

5.2 An Extended Example of Applied Negotiated Search

In this section, we trace through an execution of the **STEAM** system running the negotiated-search strategy. Negotiated search is used in **STEAM** because it is widely applicable, uses a minimal set of search operators, and takes advantage of conflict-management techniques to focus the search and provide satisficing solutions.

STEAM does parametric design of steam condensers. A complete description of the domain and of the system is provided in Chapter 6, but for the purpose of illustrating

the negotiated-search strategy, we give a brief overview here. A steam condenser comprises a set of semi-autonomous components. In the STEAM system, the condenser is decomposed into six major components, the motor, pump, heat-exchanger, platform, vbelt, and shaft, each assigned to a designer agent for that component (e.g, the motor component is designed by motor agent). A steam condenser is shown in Chapter 6, Figure 6.1. The components (and the agents designing the components) interact through a well-defined set of interface parameters, as described in Table 5.1.

The design problem is to assign a consistent set of values to the interface parameters of the components while simultaneously attempting to minimize cost over the entire steam condenser. Each agent's expertise is limited to either designing a specific component or critiquing some specific subset of solution attributes. The six designer agents were listed above. In addition a seventh agent, the frequency critic, evaluates the natural frequency of the system to ensure that excessive vibration will not occur in the overall system. No agent has global domain or control knowledge.

Each problem presented to the system is specified by assigning values to a subset of the condenser's attributes. The problem specification for the sample run is:

Problem Specification:	
required capacity	500
maximum platform deflection	.01
platform side length	120

We begin with the first agent-execution phase in the system's processing cycle (see Section 4.3.2 for an explanation of agent- and framework-execution phases). In this first cycle, the only object in shared memory is the problem specification above. Later in this chapter, we will discuss a transition-network approach to specifying the negotiated-search strategy, as shown in Figure 5.1. The figure shows that there is only one negotiated-search operator that can be applied to a problem-specification object, the *initiate-solution* operator. This operator will be described in detail later in this chapter in Section 5.4.3.2 and illustrated in Figure 5.5. More than one agent in the system can instantiate the *initiate-solution* operator. In this system run, it is instantiated by two agents, pump agent and heat-exchanger agent. Conceptually, these agents execute simultaneously as described in Chapter 4 (no changes will be made to shared memory until both agents have finished applying their operators). Therefore, no order dependencies are implied by the order in which we discuss the agents' search.

Pump agent applies its initiate-solution operator to create a pump proposal that will serve as a foundation for a new design. It generates a proposal, pump 1, and sends it to the TEAM manager (TMan) in an initiate-solution message. For an explanation

Table 5.1. *The Agent/Parameter Interface from the* STEAM *System*

Operator	Input Parameters: Parameter	Source	Units	Output Parameters
Heat Exchanger Agent				
initiate-solution	required capacity	problem specification	lb/hr	water flow rate minimum head rating acceptability
extend-solution	required capacity water flow rate available head	problem specification pump agent pump agent	lb/hr gpm ft	minimum head rating acceptability
Pump Agent				
initiate-solution				water flow rate available head run speed range required power weight rating acceptability
extend-solution	water flow rate minimum head	heat exchanger agent heat exchanger agent	gpm ft	available head run speed range required power weight rating acceptability
Motor Agent				
extend-solution	required power	pump agent	hp	power motor drive speed weight rating acceptability
V-belt Agent				
extend-solution	required power motor drive speed run speed range	pump agent motor agent pump agent	hp rpm rpm	load speed belt force load pulley weight weight rating acceptability
Shaft Agent				
extend-solution	belt force load pulley weight load speed	vbelt agent vbelt agent vbelt agent	lb lb rpm	torque weight rating acceptability
Platform Agent				
extend-solution	platform side length platform load maximum deflection	problem specification pump, motor, vbelt, shaft agents problem specification	in lb in	platform stiffness rating acceptability
Frequency Critic				
critique-solution	platform stiffness platform load motor drive speed run speed	platform agent pump, motor, vbelt, shaft agents motor agent pump agent	lb/in lb rpm rpm	rating acceptability

of system mechanics related to message passing, see Chapter 4, Section 4.3.3 and for an instance of this message type, see Chapter 6, Figure 6.7. For this first component proposal, pump 1, we show all attributes of the pump, both those that are only used locally and those that are shared with other agents. In future discussion, however, only shared attributes of proposals will be shown.

Pump 1:

model	model1-impeller11
water flow rate	.5
available head	57.02
run speed range	(1080 1320)
required power	.1
cost	103.63
weight	33.23
rating	excellent
acceptability	acceptable

During the same agent-execution phase, the heat exchanger agent also initiates a design by applying its initiate-solution operator.

Heat Exchanger 1:

water flow rate	104.12
minimum head	275.39
rating	excellent
acceptability	acceptable

Heat-exchanger agent also sends its proposal in an initiate-solution message to TMan to tell it to build a new design based on the proposal. As discussed in Chapter 6, none of the other agents instantiates the initiate-solution operator and, consequently, no other agent can contribute anything during this agent-execution phase. The framework-execution phase begins and TMan processes the two messages it received. It builds new designs for each of the two proposed components. Currently, each initiate-solution message triggers the development of a separate design even if the base proposals contained in the messages are compatible. A potential future extension to the system is to explicitly compare proposals to determine if it is possible to merge base proposals into a single design. In this particular case, however, the two initial designs are incompatible and could not be combined.

Design 1	
water flow rate	.5
available head	57.02
run speed range	(1080 1320)
required power	.1
pump	<pump 1>
initiating agent	pump agent
acceptability	acceptable
completion	partial

Design 2	
water flow rate	104.12
minimum head	275.39
heat exchanger	<heat exchanger 1>
initiating agent	heat-exchanger agent
acceptability	acceptable
completion	partial

Solution-initiation messages are sent from TMan to the agents indicating that two designs have been initiated and a new agent-execution phase begins. In Section 5.4, Figure 5.1, it is seen that there are two negotiated-search operators that can be applied to partial, acceptable designs: *extend-solution*, and *critique-solution*. These operators are described in detail in Section 5.4.3.2. Pump agent, heat-exchanger agent, motor agent, vbelt agent, shaft agent and platform agent each instantiate the extend-solution operator and frequency critic instantiates the critique-solution operator. However, each agent can optionally have domain-level preconditions on the operators in addition to those imposed by the search strategy. For example, referring to Table 5.1, in order to apply its extend-solution operator to a particular design, vbelt agent requires that there be assigned values in the design for its input parameters required power, motor drive speed, and run speed range. Since these domain-level preconditions are not satisfied in either Design 1 or Design 2, vbelt agent will not execute its extend-solution operator in this agent-execution phase.

The only agents that have their preconditions fully satisfied by any design at this point in processing are the pump, motor, and heat-exchanger agents, and these are the only agents that will extend the designs during this system cycle. If an agent instantiates more than one of the operators *initiate-solution*, *extend-solution*, and *critique-solution*, it can execute only one during a single agent-execution phase. In the control scheme used for this run, extend-solution always takes precedence over

the initiate-solution operator. Therefore, given the choice between initiating a new solution and extending an existing solution, the pump and heat-exchanger agents always extend.

Motor agent instantiates only extend-solution. It applies this operator to Design 1, generates a motor proposal, and sends a message to TMan containing the proposal.

Motor 1	
horsepower	1.0
motor drive speed	2400
weight	33
rating	excellent
acceptability	acceptable
design	<design 1>

Heat-exchanger agent also applies extend-solution to Design 1, but is unable to produce a feasible component proposal because the water flow rate specified in the design is too low and violates a local constraint.

Heat Exchanger 2	
water flow rate	.5
minimum head	nil
rating	infeasible
acceptability	infeasible
design	< design 1>

Heat-exchanger agent sends a conflict-detected message that contains the violated constraint, (water-flow-rate > .5, 0). The constraint representation presented here is of the form:

(parameter-name predicate value, flexibility)

Parameter-name is the name of the parameter that is being constrained. Predicate is one of $\{>, <, \geq, \text{or} \leq\}$. Flexibility is an integer, $0 \leq \textit{flexibility} \leq 4$. A constraint with a flexibility value of 0 is a hard constraint, meaning it cannot be relaxed. Conversely a flexibility value of 4 indicates that the constraint can be readily relaxed.

As heat-exchanger agent and motor agent extend Design 1, pump agent simultaneously attempts to extend Design 2. Notice in Table 5.1 that minimum head is an input parameter of the *extend-solution* operator of pump-agent. Pump-agent has an internal constraint, (minimum-head < 145.79, 4), that is violated by the previously assigned value for minimum head in Design 2, 275.39 ft. This constraint has a flexibility of 4, however, meaning that it is readily relaxable, so it does not preclude the development of a compatible proposal. Pump agent produces the following component proposal:

Pump 2

water flow rate	104.12
available head	310.02
required power	18.14
run speed range	(2160 2640)
weight	75.52
rating	good
acceptability	unacceptable
design	<design 2>

Although this proposal is consistent with the previously assigned attributes of the design, it does violate an internal constraint of pump agent and, therefore, is considered unacceptable at this point in processing. If the constraint is relaxed later in processing, the proposal will become acceptable. Pump agent sends an extend-solution message to TMan with the unacceptable proposal and sends conflict-detected messages to the other agents containing the violated constraint.

As discussed above, although vbelt agent, shaft agent, and platform agent instantiate the extend-solution operator and frequency critic instantiates critique-solution, they are unable to apply their operators now due to domain preconditions on input designs. Therefore, the agent-execution phase ends here, the framework-execution phase begins, and TMan processes the messages it has received.

TMan adds the new component proposals to the existing designs for which they were proposed and updates all shared parameters to reflect those in the returned proposals as shown in the extended designs below. In these designs, the conflicts are shown in the form:

{conflict-name violated-constraint$^+$ detecting-agent}

Design 1

water flow rate	.5
available head	57.02
minimum head	nil
run speed range	(1080 1320)
required power	.1
horsepower	1
motor drive speed	2400
pump	<pump 1>
motor	<motor 1>
heat exchanger	<heat exchanger 2>
initiating agent	pump agent
acceptability	infeasible
conflicts	{conflict 1 (water flow rate > .5, 0) heat-exchanger-agent}

Design 2

water flow rate	104.12
available head	310.02
minimum head	275.39
run speed range	(2160 2640)
required power	18.14
heat exchanger	<heat exchanger 1>
pump	<pump 2>
initiating agent	heat-exchanger agent
acceptability	unacceptable
conflicts	{conflict 2 (minimum head < 145.79, 4) pump agent}

Design 1 is infeasible, meaning that it violates a hard feasibility constraint, and consequently, it will never be extended further. Design 2 is unacceptable because it violates a soft constraint. Therefore, it will not be extended further at this time. However, if the violated constraint is relaxed at some later point, the design will become acceptable again and will be further extended.

Once the designs have been extended to include the new component proposals, TMan has processed all its messages and has no more work to do, so a new agent-execution phase begins. Pump agent has received the conflict message from heat-exchanger agent containing the constraint (water flow rate > .5, 0). It assimilates

that information into its own database. Since that constraint has a flexibility of 0, meaning that any design that violates that constraint will be infeasible, it will not be violated in any new designs. Pump agent has no new designs to extend, so it applies its initiate-solution operator to create a pump proposal. It sends the new proposal to TMan in an initiate-solution message.

Pump 3	
water flow rate	8.0
available head	48.54
required power	.33
run speed range	(1080 1320)
weight	33.23
rating	excellent
acceptability	acceptable

Similarly, heat-exchanger agent applies its initiate-solution operator to create a new proposal. An initiate-solution message is sent to TMan containing the proposed component.

Heat Exchanger 3	
water flow rate	110.5
minimum head	306.68
rating	excellent
acceptability	acceptable

Looking at Figure 5.1, all designs that were previously initiated (Designs 1 and 2) are currently either in the *infeasible-solution* or *partial unacceptable* states and there are no *partial acceptable* designs that can be extended or critiqued. Because there are no partial acceptable designs, none of the other agents have any work to do during this system cycle, and control returns to TMan. TMan processes the two initiate-solution messages from the pump and heat-exchanger agents by building new designs incorporating their component proposals.

Design 3	
water flow rate	8.0
available head	48.54
run speed range	(1080 1320)
required power	.33
pump	<pump 3>
initiating agent	pump agent
acceptability	acceptable

Design 4

water flow rate	110.5
minimum head	306.68
heat exchanger	<heat exchanger 3>
initiating agent	heat-exchanger agent
acceptability	acceptable

Notice that in creating the pump proposal, pump agent respected the constraint it received from heat-exchanger-agent (water flow rate > .5, 0), but heat-exchanger agent violated the constraint it received from pump agent (minimum head < 145.79, 4). The policy that an agent uses locally to determine which constraints to respect is not mandated by the **STEAM** system, the **TEAM** framework, or the negotiated-search strategy. If there is a conflict between external and internal feasibility constraints, the system fails. In this instance, however, the constraint received from pump agent was at the 4 flexibility level and was in conflict with an internal constraint of heat-exchanger agent with the same flexibility: (minimum head > 258.27, 4). In any situation where there is a direct conflict between an internal and external solution requirement, an agent must relax one or the other if it is to continue processing. The policy used in **STEAM** is that the agent locally relaxes the external constraint if they are at the same level of flexibility, basically choosing to violate the external rather than internal constraint. If the situation above occurred but the received constraint had a flexibility of 3 rather than 4, (minimum head < 145.79, 3), we would see an an example of *responsive relaxation* (Section 5.4.3.2 in this chapter). Heat-exchanger agent would relax its own internal constraint, (minimum head > 258.27, 4), in an attempt to satisfy the stronger external constraint.

Returning to the example, TMan sends messages to the agents indicating that two new designs have been installed and relinquishes control to the agent-execution phase. Pump agent begins by applying its extend-solution operator to Design 4 to generate a compatible pump component.

Pump 4

water flow rate	110.5
available head	308.04
required power	18.69
run speed range	(2160 2640)
weight	75.52
rating	good
acceptability	unacceptable
design	<design 4>

Pump agent sends an initiate-solution message to TMan containing the proposal. It also sends a conflict-detected message to the other agents, again indicating the violated constraint, (minimum head < 145.79, 4).

Heat-exchanger agent attempts to apply its extend-solution operator to Design 3 to create a compatible heat-exchanger component. However, it is unable to find one and returns an infeasible proposal along with a conflict-detected message containing a set of violated constraints.

Heat Exchanger 4	
water flow rate	8.0
minimum head	nil
rating	infeasible
acceptability	infeasible
design	<design 3>

Motor agent is also able to respond to Design 3 at this point. Since motor agent and heat-exchanger agent run simultaneously from a conceptual perspective, motor agent does not know that the design has become infeasible and produces a component proposal which is sent to TMan.

Motor 2	
horsepower	1
motor drive speed	2400
weight	33
rating	excellent
acceptability	acceptable
design	<design 3>

No other agents are able to respond at this point, and control returns to TMan. TMan has three extend-solution messages to process and integrates the component proposals into their respective designs.

Design 3

water flow rate	8.0
available head	48.54
minimum head	nil
run speed range	(1080 1320)
required power	.33
horsepower	1
motor drive speed	2400
pump	<pump 3>
motor	<motor 2>
heat exchanger	<heat exchanger 4>
initiating agent	pump agent
acceptability	infeasible
conflicts	{conflict 3 (available head > 50.1, 2) (available head > 50.1, 3) (available head > 300, 4,) (water flow rate > 50.5, 2) (water flow rate > 50.5, 3) (water flow rate > 100.5, 4) heat-exchanger agent}

Design 4

water flow rate	110.5
available head	308.04
minimum head	306.68
run speed range	(2160 2640)
required power	18.69
heat exchanger	<heat exchanger 3>
pump	<pump 4>
initiating agent	heat-exchanger agent
acceptability	unacceptable
conflicts	{conflict 4 (minimum head < 145.79, 4) pump agent}

As can be seen above, a number of heat-exchanger agent's internal constraints were violated when it tried to extend Design 3. However, none of these constraints are feasibility constraints (hard constraints with 0 flexibility), so the infeasible rating does not stem directly from the constraints. Instead it comes about because heat-exchanger agent was unable to generate any proposal that is consistent with the assigned values

of water flow rate and available head. Those values are not infeasible separately, but in combination, no proposal can be found. However, heat-exchanger agent can not express the relationship between the two attributes in a shareable format and, therefore, is unable to communicate explicit information about why it failed to find a compatible solution.

Because Design 3 is infeasible, it will not be extended further and drops out of consideration (this was also the case with Design 1). Design 4 is currently unacceptable, as is Design 2, so there are no acceptable designs to be extended or critiqued at this point. Therefore, the only operator that can potentially be applied is initiate-solution (see Figure 5.1) and the only agents instantiating this operator are pump agent and heat-exchanger agent. Pump agent begins by assimilating the constraints in the conflict-detected message sent by heat-exchanger agent. It then applies initiate-solution to generate a new component, using the assimilated constraints provided by heat-exchanger agent to avoid new conflicts to the greatest extent possible.

Pump 5

water flow rate	108.0
available head	57.94
required power	2.58
run speed range	(1620 1980)
weight	35.29
rating	excellent
acceptability	acceptable

In this case, pump agent was able to respect all water flow rate constraints and the available head constraints with flexibility of 3 or less (shown above in Design 3). One constraint, (available head > 300, 4), was violated in the proposal because no pump could be found that met both the water flow rate and available head requirements at the 4 flexibility level.

Heat-exchanger agent attempts to assimilate the constraint it received from pump agent, but finds that it already knows the information (this constraint was previously sent and found to be in conflict with an internal heat-exchanger constraint). It therefore ignores the message and applies its initiate-solution operator, generating a new heat-exchanger proposal.

Heat Exchanger 5

water flow rate	120.5
minimum head	358.75
rating	excellent
acceptability	acceptable

Pump agent and heat-exchanger agent respectively send initiate-solution messages to TMan containing their new proposals. TMan responds by incorporating these proposals into new designs and sending messages to all other agents notifying them of the new designs.

Design 5

water flow rate	108.0
available head	57.94
run speed range	(1620 1980)
required power	2.58
pump	<pump 5>
initiating agent	pump agent
acceptability	acceptable

Design 6

water flow rate	120.5
minimum head	358.75
heat exchanger	<heat exchanger 5>
initiating agent	heat-exchanger agent
acceptability	acceptable

TMan has finished processing all its messages and returns control to the agents. Figure 5.1 shows the application of the *relax-solution-requirement* operator to a solution in the 'partial unacceptable solution' state. In this example, the relaxation threshold (see Section 5.4.3.2 on automatic relaxation) is set to 5, meaning that automatic relaxation will occur after every five cycles. This is the sixth system processing cycle, and consequently, each agent will relax requirements during the agent-execution phase of this cycle.

Automatic relaxation has two major effects at an agent: it changes the agent's flexibility level and its required quality level. As discussed earlier, constraints have associated flexibility values that indicate the 'relaxability' of the constraint. The higher the flexibility value, the more flexible the constraint is. (If relaxability is thought of in terms of its relationship to solution quality, there is likely to be less penalty in quality for relaxing a more flexible constraint.) Automatic relaxation lowers the flexibility level of an agent which, in effect, relaxes all constraints with higher flexibility values.

Returning to the example above, the current required flexibility level of all agents is 4, meaning that constraints with a flexibility of 4 or below should be considered during problem solving. When automatic relaxation is applied, the required flexibility

level of an agent drops to 3 and all constraints with a flexibility of 4 are automatically relaxed. Once relaxed, these constraints are no longer considered during an agent's search, thereby expanding the search space of the agent. Furthermore, any existing solutions that were unacceptable due to violation of constraints that have now been relaxed will become acceptable. Specifically, the constraint, (minimum head < 145.79, 4), that caused conflicts in Designs 2 and 4 is relaxed, making these designs acceptable. Although not evident here, note that there are other, more directed, forms of requirement relaxation in addition to automatic relaxation as described in Section 5.4.3.2, that can also change the status of solutions from unacceptable to acceptable.

The second effect of automatic relaxation is to change the required quality level at each agent. We showed earlier that a solution may be infeasible from an agent's perspective without the agent being able to attribute the infeasibility to a specific constraint. This principle applies across the range of possible proposal ratings: an agent may rate a proposal as only *fair* without being able to specify any declarative constraints to explain the rating. Because ratings cannot be mapped directly to constraints, ratings must be relaxed independently of and in addition to constraints. In the example above, each agent begins the problem with a required quality level of *excellent*, meaning that in order to be acceptable, a proposal must be rated as excellent independently of any constraint violations or lack thereof. Automatic relaxation drops the required quality level to *good*. We will see examples later in the trace where it is the drop in required quality level that moves a solution from unacceptable to acceptable.

Returning to the example, pump agent begins a new agent-execution phase by first applying its relax-solution-requirements operator and then its extend-solution operator. It extends Design 6 under the set of constraints with flexibility of 3 or lower. However, as we saw before in the extension of Design 3, no pump can be found that will support the previously assigned values of Design 6, and the operator application results in an infeasible proposal. An internal constraint, (minimum head < 342.145, 3), is violated with respect to the minimum-head specification from the design, minimum head = 358.75. This constraint is sent to the other agents in a conflict-detected message. Again, though, the infeasibility of the proposal is not directly associated with the constraint violation.

Pump 6

water flow rate	120.5
available head	nil
weight	nil
rating	infeasible
acceptability	infeasible
design	<design 6>

Heat-exchanger agent applies its extend-solution operator to Design 5. It is able to generate a good proposal, primarily because pump agent had taken the assimilated constraints from heat-exchanger agent into account when generating the base proposal for this design.

Heat Exchanger 6

water flow rate	108.0
minimum head	18.1
rating	good
acceptability	acceptable
design	<design 5>

Notice that the heat exchanger is rated as good rather than excellent, but it is still acceptable. As part of the automatic relaxation described above, the required quality level for heat-exchanger agent was also relaxed, so that now any component with a rating of good or above is considered acceptable.

Motor agent also is able to extend Design 5 during this agent-execution phase. It generates an excellent proposal that is sent to TMan in an extend-solution message.

Motor 3

horsepower	3.0
motor drive speed	2400
weight	55
rating	excellent
acceptability	acceptable
design	<design 5>

No other agent can apply any operators at this time so control returns to TMan. TMan responds to the three extend-solution messages by integrating the component proposals with their intended designs.

Design 5

water flow rate	108.0
available head	57.94
minimum head	18.1
run speed range	(1620 1980)
required power	2.58
horsepower	3.0
motor drive speed	2400
heat exchanger	<heat exchanger 6>
pump	<pump 5>
motor	<motor 3>
initiating agent	pump agent
acceptability	acceptable

Design 6

water flow rate	120.5
available head	nil
minimum head	358.75
heat exchanger	<heat exchanger 5>
pump	<pump 6>
initiating agent	heat-exchanger agent
acceptability	infeasible
conflicts	{conflict 5 (minimum head < 342.145. 3) pump agent}

There is a global effect to both constraint relaxation and relaxation of required quality level in addition to the local effects. When any existing global designs are unacceptable due to violations of relaxed constraints, those designs are updated to reflect the relaxations. If all violated constraints contributing to the unacceptability of a design have been relaxed and if each agent's rating of the design is at or above its relaxed quality threshold, the solution becomes acceptable and available to be extended further as appropriate. Designs 2 and 4 were unacceptable due to violation of a constraint at the 4 flexibility level, but that constraint has been relaxed. After running the updates on existing solutions, those two designs became acceptable. Design 5 was trivially acceptable. Thus, control returns to the agent-execution phase with these three active designs.

During this phase, the vbelt agent is able to extend Design 5 since it now meets that agent's domain-level preconditions on the application of the extend-solution operator. The vbelt component produced is sent to TMan in an extend-solution

message.

Vbelt 1	
load speed	1841.1
belt force	79.52
load pulley weight	2.05
weight	3.5
rating	excellent
acceptability	acceptable
design	<design 5>

Pump agent has already provided components to all acceptable existing designs, so the only operator it has available is the initiate-solution operator. It has not received any new constraints, so it generates a new proposal under the current set of constraints and sends that to TMan in an initiate-solution message.

Pump 7	
water flow rate	50.5
available head	70.65
required power	2.37
run speed range	(1080 1320)
rating	excellent
acceptability	acceptable

Heat-exchanger agent also has already provided components to all acceptable existing designs, so it applies the initiate-solution operator, sending the new proposal to TMan in an initiate-solution message.

Heat Exchanger 7	
water flow rate	104.125
minimum head	131.4
rating	good
acceptability	acceptable

Motor agent is able to apply its extend-solution operator to one of the designs that became acceptable as a result of relaxation, Design 4. Design 4 has an assigned value of 18.69 horsepower for its required power attribute. This attribute is an input parameter for motor agent (see Table 5.1), which uses it to filter out motor models that do not generate enough power. Motor agent is typical of many parametric design agents in that it generates proposals based on a catalog of potential component models, in this case, motors. It searches through the catalog for the cheapest, lightest model that can satisfy the conditions imposed by the design. It assigns values to its output parameters either directly from the catalog specifications of the model or

through calculations based on input and catalog specifications. In order to understand how evaluation of a proposal is done, we will describe the motor agent's response to this design in detail. However, although this describes a common type of problem solving in parametric design, not all agents use the same evaluation scheme. The discussion is intended as an example only.

In a preprocessing task that is run once anytime the catalog is changed, motor agent evaluates the motor models it has available (in the catalog). It divides the models into quality equivalence classes by assigning each model a quality rating (one of excellent, good, fair, and poor) based on cost and weight and generates required-power and motor-drive-speed constraints at the boundary of each equivalence class. For example, to generate the required-power constraint at the boundary of the excellent class, it searches all motors with excellent quality rating, finds the maximum available power of any model, and forms the constraint (required-power < maximum power, 4). The flexibility is set to 4 in this constraint because constraints related to the *excellent* equivalence class are considered to be readily relaxable. Similarly, the *good* class has a flexibility of 3, etc.

Returning to the example, motor agent was able to find a motor model that generates the power required by the pump in this design, 18.69 horsepower. However, all models that can produce that much power are rated as *fair* or lower. The selected model is incorporated into a proposal and sent to TMan in an extend-solution message.

Motor 4

horsepower	20
motor drive speed	2400
weight	175
rating	fair
acceptability	unacceptable
design	<design 4>

The motor model proposed for Design 4 violates one of the equivalence class boundary constraints of motor agent, (required power < 4.0, 3). It also violates a boundary constraint at level 4, but this constraint has already been relaxed and is not considered at this point in processing. As noted in previous examples, the violation of a level 3 constraint does not preclude the generation of a proposal, but it does make the proposal unacceptable. The unacceptability rating from motor agent will, in turn, make Design 4 globally unacceptable for the time being. The violated constraint is sent to other agents in a conflict-detected message.

No other agents are able to apply operators due to domain-level preconditions on the design specification. Control returns to TMan to process the two extend-solution messages and the two initiate-solution messages.

Design 4

water flow rate	110.5
available head	308.04
minimum head	306.68
run speed range	(2160 2640)
required power	18.69
horsepower	20
motor drive speed	2400
heat exchanger	<heat exchanger 3>
pump	<pump 4>
motor	<motor 4>
initiating agent	heat-exchanger agent
acceptability	unacceptable
conflicts	{conflict 4 (minimum head < 145.79, 4) pump agent} {conflict 6 (required power < 4.0, 3) motor agent}

Design 5

water flow rate	108.0
available head	57.94
minimum head	18.1
run speed range	(1620 1980)
required power	2.58
horsepower	3.0
motor drive speed	2400
load speed	1841.1
belt force	79.52
load pulley weight	2.05
heat exchanger	<heat exchanger 6>
pump	<pump 5>
motor	<motor 3>
vbelt	<vbelt 1>
initiating agent	pump agent
acceptability	acceptable

Design 7

water flow rate	50.5
available head	70.65
run speed range	(1080 1320)
weight	33.33
pump	<pump 7>
initiating agent	pump agent
acceptability	excellent

Design 8

water flow rate	104.125
minimum head	131.40
heat exchanger	<heat exchanger 7>
initiating agent	heat-exchanger agent
acceptability	good

At this point, we are going to focus on the completion of a single solution rather than continuing to discuss ongoing solution initiation and extension across the entire system. We have seen examples of solution initiation, solution extension, information assimilation, conflict avoidance, conflict detection, and automatic relaxation. These themes are repeated throughout problem solving as new solutions emerge and existing solutions are extended and change state from acceptable to unacceptable and back. We will follow through the completion of one solution to look at how problem solving proceeds but following through the entire trace will not add to understanding of the system's behavior. The solution we will follow is Design 5, which is currently acceptable and has four completed components as shown above.

Control returns to the agent-execution phase at this point and shaft agent is able to apply its extend-solution operator to Design 5. The shaft design generated is rated as fair, which is currently unacceptable, but shaft agent is unable to communicate any explicitly violated constraints. It sends the proposal to TMan in an extend-solution message.

Shaft 1

shaft torque	88.15
rating	fair
weight	.75
acceptability	unacceptable
design	<design 5>

During the next framework-execution phase, Design 5 is extended by TMan:

Design 5

platform load	94.54
water flow rate	108.0
available head	57.94
minimum head	18.1
run speed range	(1620 1980)
required power	2.58
horsepower	3.0
motor drive speed	2400
load speed	1841.1
belt force	79.52
load pulley weight	2.05
shaft torque	88.15
heat exchanger	<heat exchanger 6>
pump	<pump 5>
motor	<motor 3>
vbelt	<vbelt 1>
shaft	<shaft 1>
initiating agent	pump agent
acceptability	unacceptable

Because Design 5 is now unacceptable due to shaft agent's rating, no further work is done on this solution for the next several system cycles, until automatic relaxation occurs. Although there are no explicitly violated constraints associated with the unacceptable proposal, shaft agent revises the quality level that it will consider acceptable at this time and changes the acceptability of the proposal to *acceptable.* Since shaft agent's rating was responsible for the global unacceptability of the design, Design 5 is updated and becomes acceptable.

Platform agent still has not contributed a proposal to the design. As seen in Table 5.1, it requires an assigned value for platform load in the design to be extended. Platform load is actually the combined weight of the components that will sit on the platform, in this case, the pump, motor, shaft, and vbelt. As discussed in Sections 3.2 and 4.3.4.1, the system developer can specify certain types of transformations on output variables of agents to be performed at the global level. These transformations include simple numerical operations such as addition, subtraction, etc. In this case, the STEAM system keeps a running total of the weights of relevant components and, when all components have been proposed, updates the platform-load slot of the design. Once the shaft was installed into Design 5 above, the value of platform-load could be assigned and platform agent's preconditions become satisfied. Platform agent does not extend this design right away, however, because there are several other designs to be extended, and this one is not immediately chosen. After

several cycles though, platform agent applies its extend-solution operator to Design 5 and sends a platform proposal to TMan in an extend-solution message.

Platform 2	
side length	120
thickness	1.0
stiffness	16404.2
deflection	.0057
rating	fair
acceptability	acceptable
design	<design 5>

Control returns to TMan which integrates the platform proposal into Design 5.

Design 5	
platform load	94.54
water flow rate	108.0
available head	57.94
minimum head	18.1
run speed range	(1620 1980)
required power	2.58
horsepower	3.0
motor drive speed	2400
load speed	1841.1
belt force	79.52
load pulley weight	2.05
shaft torque	88.15
side length	120
thickness	1.0
stiffness	16404.2
deflection	.0057
heat exchanger	<heat exchanger 6>
pump	<pump 5>
motor	<motor 3>
vbelt	<vbelt 1>
shaft	<shaft 1>
platform	<platform 3>
initiating agent	pump agent
acceptability	acceptable

Any time a component is added to a design, TMan checks whether the design has a complete set of components. Since this design is now complete with respect to its component set, TMan sends a component-set-completion message to the agents. However, the design is not complete with respect to the full set of agent evaluations

since the frequency critic has not yet evaluated the system natural frequency. The design will not be considered complete until all agents have provided evaluations.

During the next agent-execution phase, frequency critic applies the critique-solution operator to Design 5 and returns a critique. This critique is acceptable and results in the design being rated as both acceptable and complete, a termination state in negotiated search as seen in Figure 5.1.

Design 5

platform load	94.54
water flow rate	108.0
available head	57.94
minimum head	18.1
run speed range	(1620 1980)
required power	2.58
horsepower	3.0
motor drive speed	2400
load speed	1841.1
belt force	79.52
load pulley weight	2.05
shaft torque	88.15
side length	120
thickness	1.0
stiffness	16404.2
deflection	.0057
system frequency	11.89
heat exchanger	<heat exchanger 6>
pump	<pump 5>
motor	<motor 3>
vbelt	<vbelt 1>
shaft	<shaft 1>
platform	<platform 3>
frequency critic	<evaluation 2>
initiating agent	pump agent
acceptability	acceptable
completion	complete

As discussed earlier, the termination policy of the system is specified separately from the search strategy. In most of the experiments described in the dissertation, the termination policy is to continue processing until three complete acceptable solutions have been developed, and then to stop initiating new solutions, but finish any existing partial acceptable solutions (see Chapter 6, Section 6.2.5). Under this policy, this is the second design completed and, as such, it has no immediate effect on system termination other than to increment the number of complete acceptable solutions.

Processing continues on other solutions that are also under development. However, another solution completes on the following cycle and TMan applies its termination mechanisms, bringing the system to a halt.

In this section, we provided an annotated trace of system activity under the negotiated-search strategy in the STEAM domain. In the following section, we step back from detailing how negotiated search works in a specific domain with a specific agent set and instead discuss why it works as a domain-independent and agent-set-independent strategy. We present a general representation for distributed-search strategies and their operators, as well as the specific representation and instantiation of negotiated search and its operators.

5.3 Specifying a Distributed-Search Strategy

A *distributed-search strategy* defines a partial order on the invocation of a set of search operators that can be applied to a solution. The ordered application of the operators moves a solution from an initial state to a termination state. The operators may be, and sometimes must be, distributed across multiple agents. A specific strategy elicits a particular style of coherent inter-agent behaviors within an agent set. The definition of a strategy includes a list of the operators that are required to achieve the strategy and a transition-network description of the solution evolution. An example of a transition-network description is shown in Figure 5.1.

At the current time, we have not formalized the process of specifying a strategy. The transition-network descriptions provide clear specifications for what the coordination-space preconditions should be on an operator and a well-defined ordering on operator application with respect to a single solution. A programmer can easily map the specifications to a precondition function for an operator. However, these descriptions cannot currently be used to automatically generate the code needed by the system. Automatically moving from a "paper" design of a strategy to an executable version of that strategy is a possible direction for future research however.

5.4 The Negotiated-Search Strategy

Negotiated search (*NS*) is the default search strategy implemented for TEAM. Unless the system developer specifies otherwise, *NS* is automatically invoked at system start-up time and remains in effect except when the agent set mutually agrees to switch to a customized strategy. In this section, we will describe the search process in the context of *NS* through:

- the specific realization of an instantiated-strategy object used for *NS*;

- the incremental evolution of a solution from an initial state to a termination state through a transition network;
- the negotiated-search operators required for *NS*;
- agent control of locally instantiated operators.

5.4.1 *A Transition-Network Description of Negotiated Search*

Figure 5.1 provides a global view of the transition of a composite solution from its initial state (a problem specification) to a termination state (an infeasible solution, an unacceptable solution, or a complete acceptable solution). In this figure, states are defined in terms of three attributes of the composite-solution object (see Figure 4.4): *type*, *acceptability*, and *completeness*, and two attributes of the instantiated strategy object (see Figure 5.2): *search-state* and *number-of-acceptable-solutions*. Possible values for *type* are *problem-specification* and *solution*. The possible values for *acceptability* are *acceptable*, *unacceptable*, and *infeasible*. Possible values for *completeness* are *partial* and *complete*, where *complete* means that all agents have provided a local evaluation for the solution. A solution that has all of its component parts will still be considered a partial solution until all critic agents have responded. *Search-state* can take the values *open* or *closed*. *Number-of-acceptable-solutions* must have a non-negative integer value.

Arcs are negotiated-search operators that can be applied by some agent to a solution. Not every agent can necessarily apply every operator. The *agent-role-pairs* attribute of an instantiated-strategy object (see Figure 5.2) will be used to assign particular roles (comprising one or more operators) to agents given a specific strategy and a specific agent set.

Given the view shown in Figure 5.1, we will discuss the transitions that can occur and the conditions that enable each transition. We are describing the control flow from a global perspective: an individual agent takes note only of conditions that would cause it to execute a local negotiated-search operator. The agent-level view of problem solving will be described in Section 5.5. Also, the transition network charts the development of a single solution but multiple solution paths are simultaneously explored as discussed in Section 6.2.2 and as illustrated in the example system trace in Section 5.2.

We begin with the simplest case: the development of a solution without conflict. In this scenario, we have three agents: *A1*, *A2*, and *A3*; and the framework manager, `TMan`. Each agent instantiates one or more search operators from the negotiated search set: *initiate-solution*, *extend-solution*, *critique-solution*, and *relax-solution-requirement*. In addition, `TMan` instantiates the *terminate-search* operator. These

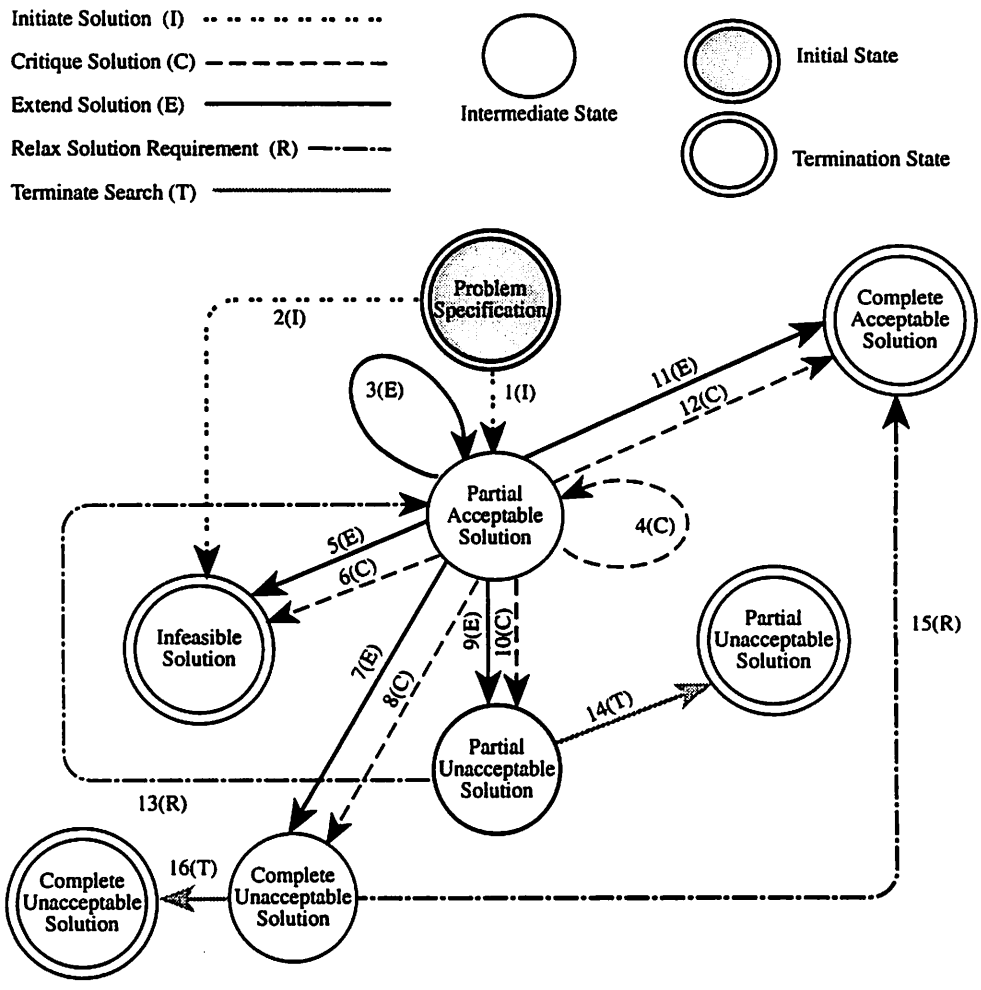


Figure 5.1. *A Transition-Network View of Negotiated-Search*

operators will be described in detail in following sections, but we provide a brief overview here to give the reader a sense of their functionality. *Initiate-solution* is applied by an agent to generate a proposal that will be used as the basis for a new composite solution. *Extend-solution* both: 1) adds a component proposal to a composite solution that comprises multiple interacting component proposals; and 2) evaluates the composite solution. *Critique-solution* evaluates a composite solution without generating a component proposal. *Relax-solution-requirement* selects a local relation to relax, updates the local database to effect the relaxation, and reevaluates existing solutions in light of the relaxation. *Terminate-search* is applied by TMan to change the state of the current strategy from *open* to *closed*, an action that affects the termination status of solutions.

Assume that *A1* initiates a solution, *A2* critiques some aspect of the solution, and *A3* extends that solution. Looking at the state diagram in Figure 5.1, first, *A1* applies the operator *initiate-solution* to a problem specification and produces a

partial acceptable solution (*arc 1*). *A2* then applies the operator *critique-solution* without detecting a conflict (*arc 4*). Next, *A3* applies the operator *extend-solution* and completes the solution without detecting a conflict (*arc 11*). The solution is now complete and, because no conflicts were detected, it is acceptable to all agents and has reached a termination state. In this case, the solution path started at the initial state, problem specification, traversed *arc 1* to a partial acceptable solution, traversed *arc 4* to remain as a partial acceptable solution, and then traversed *arc 11* to become a complete acceptable solution.

We now examine a more typical case in which some agent detects a conflict. Again, assume that *A1* initiates a solution, *A2* critiques some aspect of the solution, and *A3* extends the initial solution. *A1* first applies the operator *initiate-solution* to a problem specification and produces a partial acceptable solution (*arc 1*). Then *A2* applies the operator *critique-solution* without detecting a conflict (*arc 4*). *A3* next applies the operator *extend-solution* and detects a conflict (*arc 7*). It finds that in order to complete the solution, it must violate one of its local relations. It goes ahead anyway and completes the solution, but evaluates the completed solution as unacceptable. This solution remains as it is for some amount of time while the agents are working on other solution paths. However, at some later time, *A3* decides to relax the relation that made this solution unacceptable. The solution is now acceptable to *A3* and since it was already complete, it reaches the termination state of complete acceptable solution (*arc15*) . The solution path for this scenario started at the initial state, problem specification, traversed *arc 1* to a partial acceptable solution, traversed *arc 4* to remain as a partial acceptable solution, traversed *arc 7* to become a complete but unacceptable solution, and then traversed *arc 15* to become a complete acceptable solution.

Now assume that the same sequence of events occurs except that *A3* never relaxes the relation that makes the solution acceptable. Instead, the solution remains complete, but unacceptable, while other solution paths are being explored. Let n be an integer set by the system developer that specifies how many complete and acceptable solutions should generated by the system. For example, say that this is a design system and that the user of the system wants to be presented with three alternative designs to choose from. At some point, other solutions become complete and acceptable, the value of the *number-of-acceptable-solutions* attribute of the active instantiated-strategy object becomes $>= n$, and TMan applies the *terminate-search* operator. This solution then traverses *arc 16* to enter the termination state of complete unacceptable solution. For this situation, the solution path is: problem-specification, *arc 1* to partial acceptable solution, *arc 4* to partial acceptable solution, *arc 7* to

complete unacceptable solution, and *arc 16* to the complete unacceptable solution termination state.

5.4.2 *Instantiating the Negotiated-Search Strategy*

The transition-network description of a strategy provides an agent-independent and system-independent specification of the *negotiated-search* strategy. In order to bind the strategy to a specific application system, information about which strategy is in force and how the strategy will be distributed must be explicitly declared.

In Figure 4.10, we showed the generic representation of an instantiated-strategy object. In Figure 5.2, we show the extensions to that object used to specifically support the *NS* strategy, including the default initialization values for attributes of the strategy that can be set without reference to a particular agent set. The *agent_role_pairs* are specific to the agent set and can only be initialized once the agent set is selected.

```
Instantiated_Strategy
   Strategy_name:        negotiated-search
   Manager:              nil
   Type:                 default
   Solution:             nil
   Agent_role_pairs      % A list of the agents involved in the
                         %   strategy along with the roles
                         %   they can play in the strategy.
   Event:                nil
   Activation State:     active
   Search State:         open
   Suggested Strategy:   nil
   Number of Acceptable Solutions: 0
```

Figure 5.2. *Initial Instantiated-Strategy Object for Negotiated Search*

Agent-role pairs are established by a system developer at the time the agent set is integrated into the TEAM framework. This is done by setting parameters associated with operators instantiated at each agent to make desired operators active within a particular system. Agent-role pairs can also be established dynamically by the system if a new distributed-search strategy goes into effect during problem-solving as described in Chapter 7. The instantiated-strategy object in effect for the extended example of Section 5.2, complete with assigned agent_role_pairs, is shown in Figure 5.3.

```
Instantiated_Strategy
   Strategy_name:       negotiated-search
   Manager:             nil
   Type:                default
   Solution:            nil
   Agent_role_pairs     ((pump-agent  (initiate extend))
                         (heat-exchanger-agent (initiate extend))
                         (motor-agent (extend))
                         (vbelt-agent (extend))
                         (shaft-agent (extend))
                         (platform-agent (extend))
                         (frequency-critic (critique)))
   Event:               nil
   Activation State:    active
   Search State:        open
   Suggested Strategy: nil
   Number of Acceptable Solutions: 0
```

Figure 5.3. *Initial Strategy Object for the Extended Example*

This strategy object supplies information to the agents about how they should interact and what functionality is required of them and it supplies information to the TEAM framework about what strategy is in effect and what the current state of the strategy is. We have shown the transition-network view of the *negotiated-search* strategy and how that strategy is bound to a system. In the next section, we describe the detailed functionality required for realization of the *NS* operators within each agent.

5.4.3 Operator Descriptions

In this section, we describe in detail the operators used for *NS*. From a control perspective, there are three classes of operators:

1. operators that affect solution state
2. operators that affect strategy state
3. operators that affect agent state

Operators that affect solution state are applied by agents (including TMan) to composite-solution objects. Operators that affect strategy state are applied by TMan to instantiated-strategy objects in shared memory. Operators that affect agent state are applied by an agent to its own local information.

Every agent can examine composite solutions and can apply operators that will result in changes being made to that solution. As described in Chapter 4, agents do not make direct changes to the shared memory. Instead they send messages to the TEAM framework manager, TMan, to request the changes. TMan then applies system-level functions to effect the changes. This level of abstraction enables the system developer to maintain a separation between the implementational details of the framework and the abstract definition of operators and strategies. For example, an agent should not have to know the details of how to access a particular attribute of a specific solution instance. As long as the agent can provide the name or identification of the instance and the name of the attribute, the framework can hide the implementational access details.

5.4.3.1 Strategy-State Operators

Strategy-state operators affect the global state of the strategy currently in effect. A strategy-state operator is applied by TMan in response to some globally defined event. When a strategy-state change occurs, it affects all solutions currently under consideration.

Terminate-Search: We begin with a description of the *terminate-search* operator. Terminate-search is a strategy-state operator that is applied directly by TMan when the *number-of-acceptable-solutions* attribute of the negotiated-search strategy reaches a user-supplied value, n. The value of n indicates the number of desired alternative solutions that should be returned. The input to this operator is an instantiated-strategy object that describes the strategy currently in effect. The output is a modified strategy object with its *search-state* attribute modified to have the value *closed*. The *terminate-search* operator is shown in Figure 5.4.

Strategy-state operators are applied by TMan to instantiated-strategy objects in shared memory. An instantiated-strategy maintains information about the activation state, phase, and role assignments of a strategy. Changes to strategy objects are triggered by events related to either acceptability or termination policies.

By examining the termination policy used in the STEAM system, we can understand the purpose and functionality of *terminate-search*. Under this policy, there are two stages of search: *open* and *closed*. During the first stage, agents try to define and bound the composite search space and to ensure that the space has been reasonably searched. This involves initiating and extending solutions at various points in the space, exchanging local constraining information, and making decisions about how to relax local solution requirements. During the second stage of problem solving, agents no longer initiate new search paths or relax solution requirements. However,

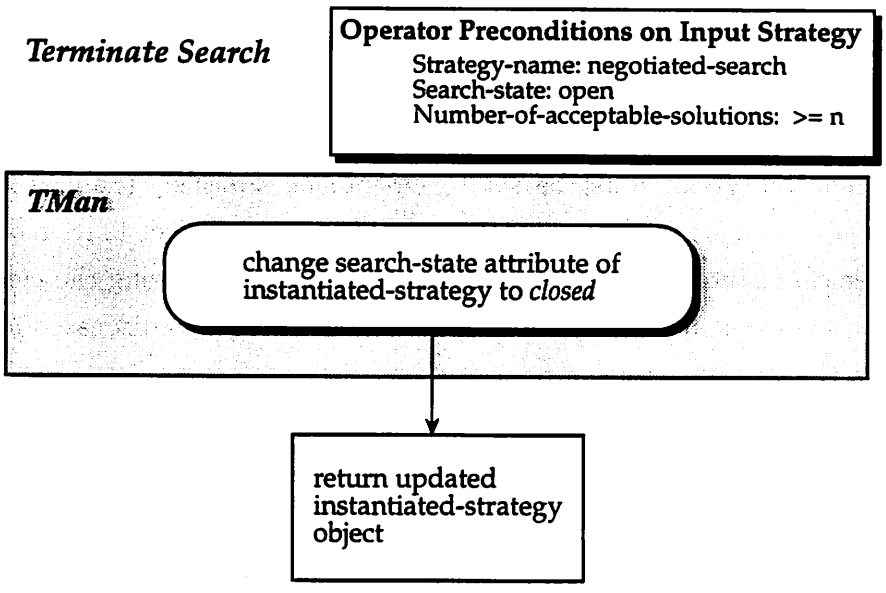


Figure 5.4. *The Terminate-Search Operator*

they complete any promising solutions that were already started to ensure that good solutions are not abandoned.

Problem solving begins with the *search-state* attribute of the active instantiated-strategy object set to *open*. It continues until n complete and acceptable solutions are generated, where n is the user-specified number of desired alternative solutions. When the appropriate number of solutions have been found, the *terminate-search* operator is applied (by TMan) to change the stage of the *NS* strategy from *open* to *closed*. As seen in Figure 5.1, when this change occurs, partial and complete unacceptable solutions move from intermediate to termination states. Partial acceptable solutions, however, remain in intermediate states. The realization of the state changes in the system are accomplished by deactivating some operators and activating others. For example, the relax-solution-requirement operator will be deactivated at all agents so that no further relaxations will occur. This effectively eliminates the possibility that solutions that are currently unacceptable will become acceptable at some future time.

5.4.3.2 Solution-State Operators

Strategy-state operators affect the state of problem solving by changing meta-level attributes of the system. Solution-state operators, on the other hand, affect a single composite solution. For negotiated search, we have defined four solution-state operators: *initiate-solution*, *critique-solution*, *extend-solution*, and *relax-solution-requirement*. These operators are described in detail below.

Initiate-Solution: This is the basic operator for generating new proposals and counter-proposals. The precondition requirements are that the type of input solution is *problem specification*, its completeness slot must have the value *incomplete*, and its acceptability slot must be set to *acceptable*. Further, the search-state attribute of the active negotiated-search instantiated-strategy object must have the value *open* because the *initiate-solution* operator is not applicable during the *closed* search stage: no new solutions are initiated during the closed stage, as discussed above. Figure 5.5 depicts the *initiate-solution* operator.

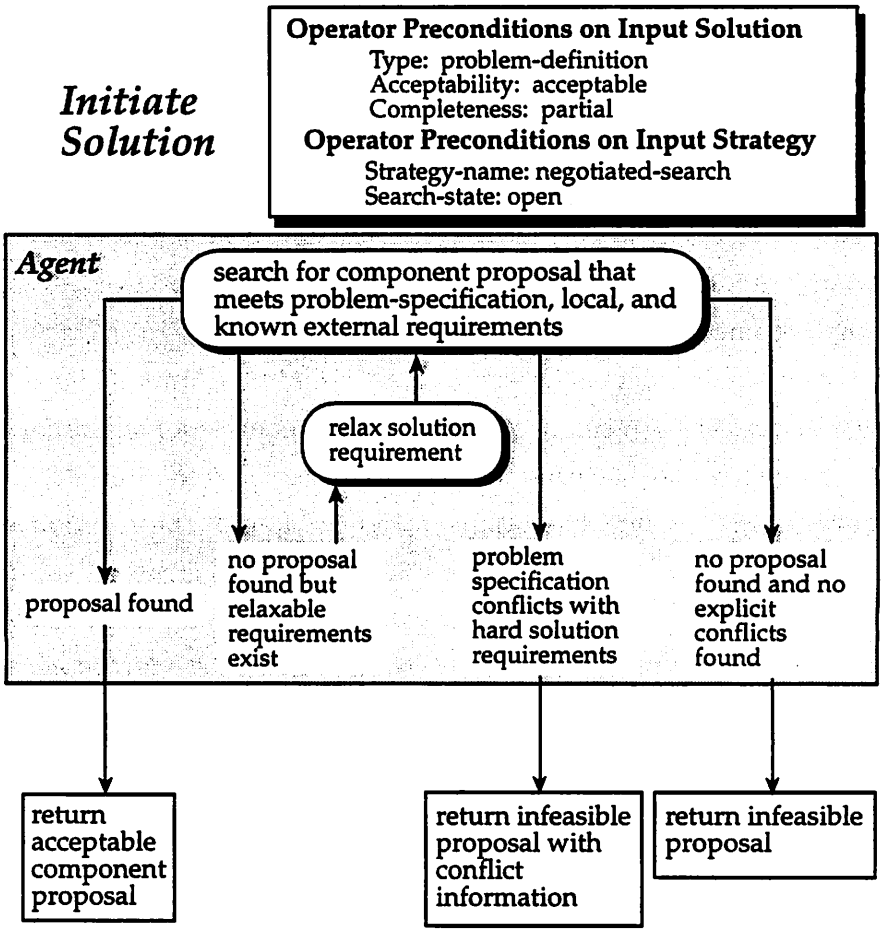


Figure 5.5. *The Initiate-Solution Operator*

Examples of the application of this operator are provided in the system trace in Section 5.2. *Initiate-solution* is applied within the agent's view of the composite search space as defined by the solution requirements imposed by the problem specification, the agent's local requirements, and any known external requirements from other agents. External requirements are those that were communicated by other agents in response to some earlier conflict. They are locally assimilated and, once assimilated,

can be used to avoid conflicts over the same requirement in future proposed solutions. Under the full set of local and external requirements, the agent creates an initial partial specification of a composite solution called the *base proposal*. The base proposal is simply a local solution that specifies some attributes of a composite solution. When a base proposal is found, the agent sends an *initiate-solution* message to TMan containing the proposal (*initiate-solution* messages are discussed in Section 4.3.3) and then ends its execution cycle.

When no base proposal is found under the requirements as described above an agent will attempt to relax a solution requirement. This expands the solution space and, hopefully, the agent will find a solution in the expanded space. The algorithm used for choosing a constraint to relax is not mandated by TEAM or by the negotiated-search strategy since different algorithms may be appropriate in different domains, with different types of requirements, and with different requirement representations. In Figure 5.6, we depict the algorithm used by STEAM for choosing a numerical boundary constraint to relax. It is based on finding the most flexible constraint possible, preferably owned by some other agent, for the least important parameter, and for the least important boundary (either minimum or maximum).

Notice that when an agent internally relaxes a constraint owned by another agent, the other agent is not forced to relax this requirement in its local space. Instead, the two agents maintain inconsistent levels of relaxation. The initiating agent was able to tell from its own view of the composite space that no solution is possible under the complete set of requirements and, therefore, some agent will have to relax a requirement. From its local perspective, it selects which agent should relax and which specific constraint should be relaxed. All else being equal, it will choose to violate another agent's constraints rather than relax its own, but if all else is not equal, it will relax its own. Because each agent will have a different bias for relaxing requirements (based on preferring to relax other agents' requirements), and because the relaxations affect the local view of the composite solution space, each agent will have a different perspective on solution initiation. The global effect of this is that agents are able to explore the consequences of different relaxations.

Once relaxation has occurred, if a base proposal is found, it will be acceptable to the initiating agent. This is because if an internal requirement is selected to relax, the agent will explicitly relax it, thereby changing its range of acceptable values. If it chooses another agent's solution requirement, the proposal generated will not violate any local solution requirements and therefore will be locally acceptable. However, since it violates some other agent's requirements, it is suggested as a possible compromise rather than a solution that is expected to be fully acceptable. The

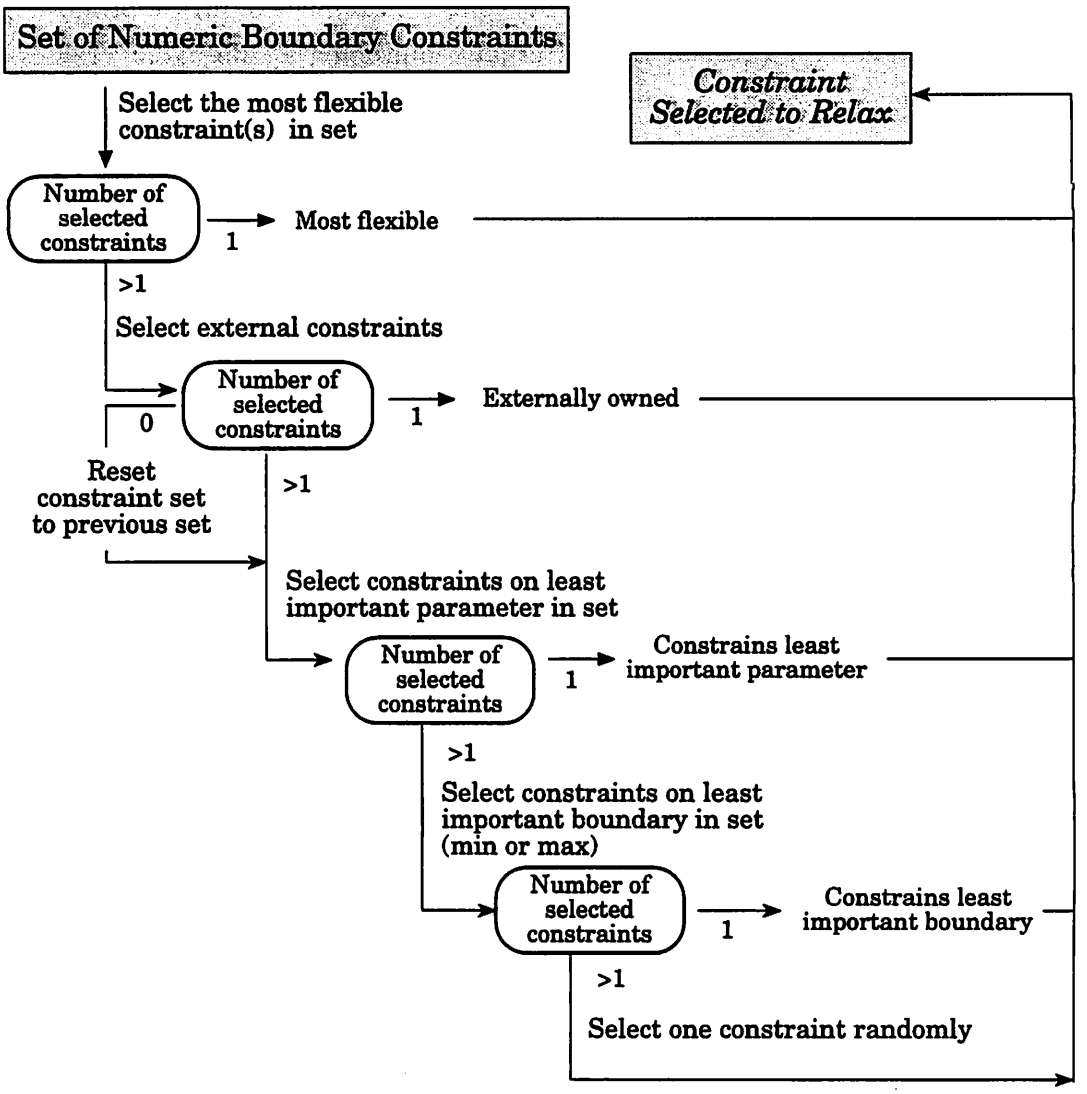


Figure 5.6. *The Algorithm used for Selecting a Numeric Boundary Constraint to Relax in* STEAM

external agent cannot be forced to accept the compromise. However, it may choose to accept it after looking at other possibilities.

If no base proposal can be found at any level of external or internal requirement relaxation, the agent fails and sends an *infeasible-proposal* message to TMan. If there are any hard solution requirements that are incompatible with the problem specification, these will be returned as well and will be reported to the user. If no hard requirements are in conflict, but no solution is found, the agent simply returns. This is a point where either human/machine or machine/machine negotiation could occur. It might be desirable to change the problem specification, either automatically or interactively with the user. In some domains, failure is not an acceptable alternative. For example, Moehlman [Moehlman and Lesser, 1990] describes a system that attempts to control

a fire given a set of resources. In this case, giving up and allowing the fire to burn is never a solution. When the system can't find a way to control the fire within a set of initial specifications, it relaxes the specifications and tries again to find a solution. Intuitively, if it can't save the house and the barn, it gives up the barn and tries to save the house.

We currently let a TEAM application system fail if an agent fails to find a solution under the initial problem specification. In some situations failure to find even an initial proposal can indicate a deficiency in the set of agents that were chosen for the problem: perhaps a more powerful agent set could solve the problem. Another possibility is that the problem specification is simply too restrictive and no solution exists. In this case, any guidance that could be provided to the user about why the system failed would be helpful in determining whether and how to change the specifications. *Initiate-solution*, therefore, will return any explicit conflicts found.

At least one agent in the agent set must instantiate *initiate-solution*, however, it is often desirable to have it instantiated at multiple agents. Instantiating solutions at multiple agents is likely to result in a more diverse set of initial solution paths and more thorough coverage of the composite solution space. However, depending on the agent set and agent characteristics, it may also have a distracting effect. These conflicting potentials are discussed in detail in Chapter 9.

It was noted in Figure 5.5 that the initiate-solution operator is applied to a problem definition within the negotiated-search strategy. However, the problem definition is a static object—it does not change over the course of problem solving. This leaves open the question of when the initiate-solution operator should be applied. At the extreme, this operator could be applied by an agent at every execution cycle since its precondition requirements will be continuously met. This issue is outside of the control of the negotiated-search strategy and instead falls in the realm of local control. Each agent must determine, given a set of operators whose preconditions are met, what it should do at a given time. For example, an agent may have the option to either initiate a new solution or extend an existing solution that was initiated by another agent.

In the current version of STEAM, agents do not have highly sophisticated local control protocols. Initiate-solution is basically applied by an agent when it has nothing else to do during an execution cycle. This works in STEAM because there are not many agents with solution-initiation capabilities. However, in the general case, agents must have some mechanism for reasonably choosing an operator to apply based on local and/or inter-agent criteria. The decision often must take into account how a particular task affects the coherence of the entire system. This falls outside of the scope of

this dissertation, but has been an active area of research [Corkill and Lesser, 1983, Decker and Lesser, 1992a, Durfee and Montgomery, 1990, von Martial, 1992, Lesser, 1991]. In future work, the effects of sophisticated local control and coordination will be examined in more detail.

Extend-Solution: The *extend-solution* operator is required for domains where solutions comprise interacting component proposals. An agent applies this operator to generate a component proposal that extends and is compatible with an existing solution. The input for this operator is a partial solution that was generated by another agent and that may or may not have been extended by additional external agents. The output is a proposal and, when a conflict exists, conflict information. Figure 5.7 details the functional definition of *extend-solution*. An example of the application

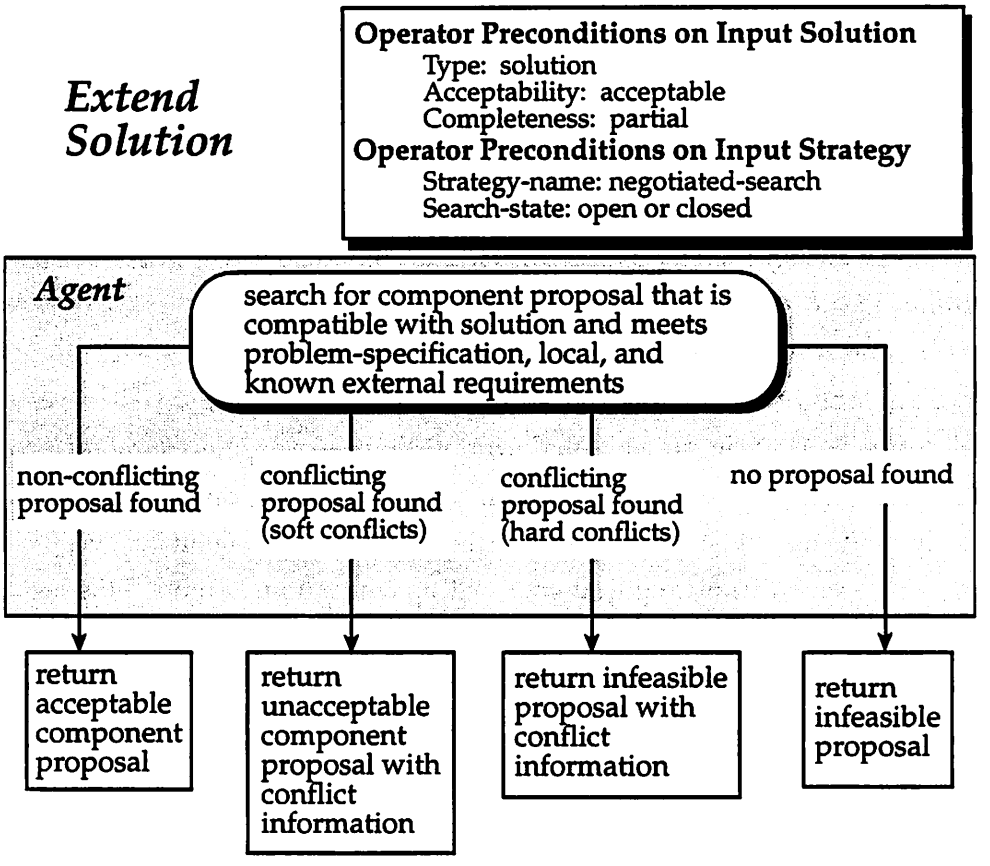


Figure 5.7. *The Extend-Solution Operator*

of this operator during execution of the STEAM system is found in Section 5.2. The agent executing the operator first searches for a compatible proposal under its known solution requirements and the requirements that are imposed by the assigned values of the input existing solution. For each parameter that is used to constrain the search of the agent, if a value for that parameter is supplied in the existing solution, the

actual value is used. For example, in **STEAM**, **pump-agent** will use the existing value for **minimum head** if it has already been assigned (see Chapter 6). When there is not an assigned value in the solution, however, the agent attempts to establish a value or bounded range of values by applying known solution requirements, both internal and external. In the example above, using only *a priori* information about the pump models available in its 'catalog', the **pump-agent** would select numeric boundary constraints on the value of **minimum head** such as (minimum-head $\leq$ 404.47) and (minimum-head $\geq$ .1), resulting in a bounded range of (.1 404.47). These constraints represent the minimum and maximum boundaries of values that can be assigned to the minimum-head attribute in any potential solution.

If the assigned values in the input solution violate hard (inflexible) solution requirements at this agent, the agent immediately returns an *infeasible* proposal along with as much information as possible about the violations. In this case, the agent does not attempt to actually generate a proposal, but instead returns a standard "dummy" or template of a proposal with an acceptability attribute set to infeasible.

If no hard solution requirements are violated, the agent searches for a compatible local extension of the solution. If a compatible extension is found that is consistent with assigned values and does not violate any solution requirements, it is returned as an *acceptable* proposal. Otherwise, if the best compatible extension violates some local solution requirements, the proposal is returned as an *unacceptable* proposal along with any information that can be sent to describe the conflict. In this case, it is not impossible to extend the existing solution, but all consistent extensions violate some local solution requirements. However, the work has already been done to find this solution, and the solution may turn out to be the best that can be found. Therefore, the solution is saved as a potential compromise.

Notice that two types of information are returned, a proposal and conflict information. These types represent the two different functions described in our discussion of negotiation: generating proposals and generating feedback. The proposal designates a set of solution attribute values that partially specify a solution. Conflict information, on the other hand, is an abstraction of the local search space of the agent, specifying violated requirements on that space. A proposal, therefore, provides a specific solution to a specific problem while conflict information provides guidance about the requirements for any solution to that problem. When another agent receives conflict information, it may affect all future proposals generated by that agent.

Returning to our discussion of the *extend-solution* operator, a fourth possibility is that no compatible solution can be found by the agent even though no specific requirement violations are detected. In this case, the agent returns an *infeasible* proposal without any conflict information. This is actually a common situation. It occurs

because there are implicit constraints on solutions that exist in the implementation of an agent. The agent is unable to articulate the problem and therefore cannot provide any guidance about how to avoid it. In a particular application system, if many solution paths are ending in infeasible proposals at a particular agent with no conflict information being generated, an examination of the implementation of that agent may turn up implicit constraints that are affecting the search. Making these constraints explicit could then improve the global effectiveness of the search by providing feedback to other agents.

Critique-Solution: This operator is similar to *extend-solution* except that it returns a critique rather than a proposal (see Figure 4.6). *Critique-solution* is used by an agent that wishes to critique an existing solution without adding a component proposal to the solution. It can be used in applications systems where the overall problem is not decomposed into components, but rather solutions are generated in entirety and evaluated as a whole as is the case with many locally cooperative systems or multi-perspective reasoning systems (the **AGREE** system has this characteristic for example). It may also be used in systems that do decompose the problem but where some set of solution elements is evaluated separately from the proposal generation process. For example, in the **STEAM** application system, the natural frequency of the system is dependent on the stiffness of the platform and the weight of all components that will sit on the platform. Before calculating natural frequency, all components that affect stiffness and weight must be specified. Once the attributes are specified, system natural frequency is calculated and must be checked for compatibility with frequencies associated with pump and motor components. These tasks are handled by an agent that critiques a finished design. This critique is inherently associated with a set of components rather than a single component solution–otherwise, it would be included as part of the domain expertise of one of the component designers.

Critique-solution takes as input an existing partial acceptable solution. It can be applied whether the *search-state* of the *NS* strategy is open or closed. It returns a critique that specifies an acceptability value. When that value is not *acceptable*, it also returns conflict information that specifies why the solution is not acceptable if possible. Figure 5.8 details the functional definition of *critique-solution*.

Relax-solution-requirement: Relaxation of solution requirements is a necessary part of negotiated search. Agents relax solution requirements for different reasons. There are three primary forms of relaxation, *unilateral relaxation*, *responsive relaxation*, and *automatic relaxation*. Responsive and automatic relaxation were described in the context of a detailed example from the execution of the **STEAM** system in Section 5.2. Unilateral relaxation occurs when an agent decides to relax a requirement

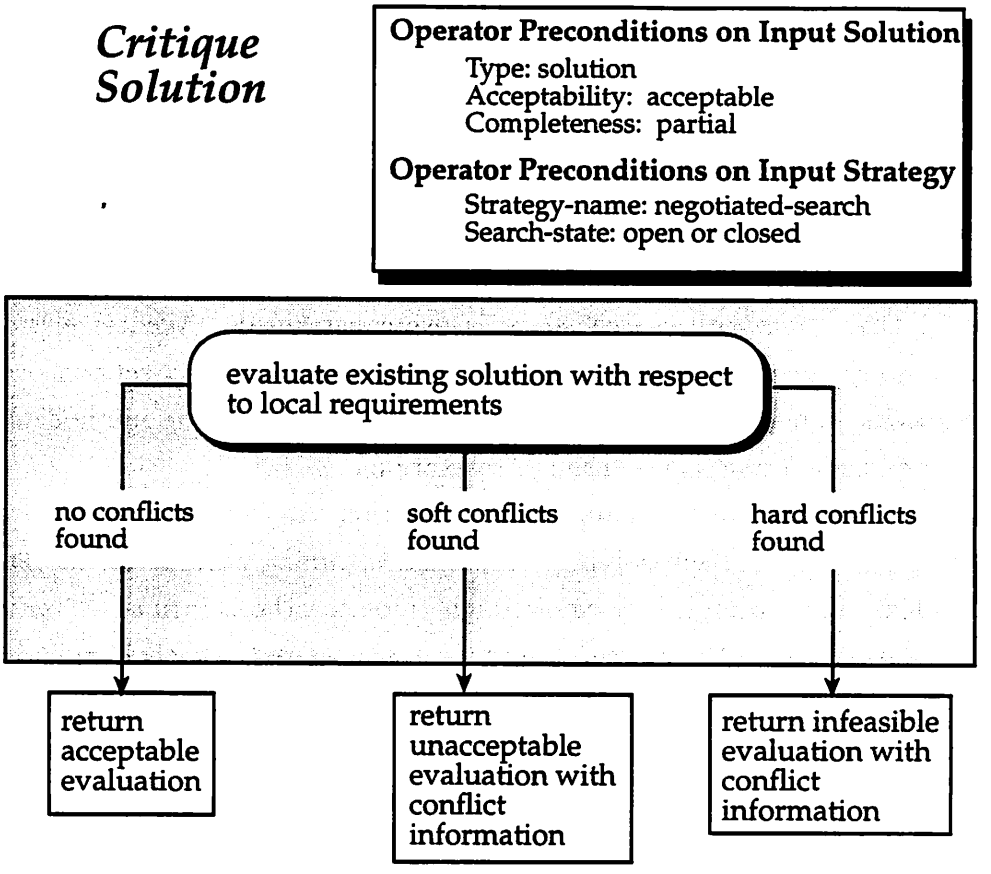


Figure 5.8. *The Critique-Solution Operator*

due to its inability to find a solution under the problem specification, i.e., the agent knows that, given the problem specification and its set of solution requirements, it will not find a locally acceptable solution. This situation occurs in the *initiate-solution* operator as shown in Figure 5.5.

Responsive relaxation occurs when an agent relaxes a solution requirement because of some explicit information about the requirements of some other agent(s), i.e, a conflict is found between flexible local solution requirements and less flexible external solution requirements. This occurs when external information has been received and stored by an agent and the agent attempts to retrieve a consistent set of solution requirements, as described in Section 5.4.3.3.

Automatic relaxation is a reaction to a lack of overall problem-solving progress. In the current TEAM framework, automatic relaxation occurs at specific processing-cycle intervals, for example, all agents may relax a solution requirement after 10 processing cycles. Alternatively, the user can specify the relaxation parameter separately for each agent, so that one agent may relax after 10 processing cycles while another will not relax until after 20 processing cycles. The concept behind automatic relaxation is that

the problem may be overconstrained by the full agent set so that no solution exists under the relations of the composite search space. If all relations were guaranteed to be sharable and if communication were perfect, agents could exchange all information required to determine whether or not this is the case. Either unilateral or responsive relaxation would be guaranteed to occur and automatic relaxation would not be necessary. However, the reality is that the agents can't always determine whether or not the composite search space is overconstrained. Agents relax requirements only under specific conditions that can't be guaranteed to occur in overconstrained situations. Therefore, it is necessary to have some heuristic method for deciding to expand the solution space through relaxation.

The original concept of automatic relaxation was to use explicit information about the progress of problem solving rather than a timing parameter, to determine when to relax. For example, an agent might monitor the solutions it initiates and use information about their acceptability to other agents and their global acceptability to decide if it is being too tight in its own requirements. Alternatively, an agent might relax a requirement if its last n proposals were rejected by other agents. These mechanisms have not yet been implemented; however, this an area that is open to future exploration.

Because of automatic relaxation, we can guarantee that if any initiated solution can be feasibly extended by all agents, some solution will eventually become acceptable to all agents. In other words, if any initial proposal is generated that can result in a feasible solution, either the solution based on that proposal will eventually become acceptable to all agents, or some other solution will become acceptable to all agents and deadlock will not occur.

5.4.3.3 Agent-State Operators

Agent-state operators are applied to private agent information, rather than to global objects as is the case with solution-state and strategy-state operators. Since these operators do not affect composite solutions, they do not appear in the transition-network description of a strategy (such as that in Figure 5.1). However, they play an important role in facilitating the transfer of knowledge and the coordination of internal actions with the agent-set environment. An agent-state operator updates an agent's local state in response to changes in the global environment and information provided by other agents.

In this section, we describe two operators that are used to update an agent's view of the composite search space by assimilating conflict information provided by other agents, *store-information* and *retrieve-information*. By assimilating external

conflict information, it is sometimes possible to avoid conflicts. When avoidance is not possible, the agent can make intelligent conflict resolution decisions based on an understanding of how a decision will affect itself and other participating agents.

Store-information: This operator takes conflict information from other agents, translates the information into a locally usable form if necessary, syntactically checks to see if the information already exists in the local knowledge base and, if not, stores it so that it can be retrieved. A received requirement may be indexed by various attributes including the name of the sending agent, the flexibility of the requirement, the names and acceptable values of constrained solution attributes, and, in the case of ordered solution attributes, whether the requirement defines a minimum or maximum boundary on potential values, e.g., $x > 5$.

Retrieve-information: This operator extends or replaces an agent's default capability to retrieve relevant constraining information from its knowledge base. Historically, agents are designed with the expectation that their knowledge will be locally consistent and, therefore, their retrieval mechanisms generally do not handle cases where conflicts may exist in the retrieved requirements. Requirement retrieval occurs during solution initiation, extension, and criticism. The goal of the retrieval process is to find the most restrictive, but non-conflicting, set of solution requirements that constrain a solution for the current local search problem. Different types of requirements require different treatment, but to provide a concrete example of retrieval, we present the algorithm used for selecting boundary constraints on numerical solution attributes in our application systems. As previously stated, boundary constraints are very simple constraints that specify either a maximum or minimum value for an attribute such as $x > 5$.

In this algorithm, potentially relevant constraints are first retrieved and sorted into maximum and minimum boundary groups. The most restrictive maximum constraint (MAX) and the most restrictive minimum constraint (MIN) from each group are selected (where most restrictive means the highest value from the MIN group and the lowest value from the MAX group). Then the algorithm loops through the following sequence until a non-conflicting set of minimum and maximum values is found or until it is determined that no non-conflicting set exists.

LOOP: If the value of MAX is greater than or equal to the value of MIN, return MAX and MIN since a non-conflicting set has been found. Otherwise, if the flexibility of MAX is greater than the flexibility of MIN select the next most restrictive maximum constraint (MAX) and go to LOOP. Otherwise, if the flexibility of MAX is less than the flexibility of MIN, select the next most restrictive minimum constraint (MIN) and go to LOOP. Otherwise, the flexibility of MAX is equal to the flexibility of MIN.

Then: if MAX is locally owned, select the next most restrictive minimum constraint (MIN) and go to LOOP. If MAX is not locally owned and MIN is locally owned, select the next most restrictive maximum constraint (MAX) and go to LOOP. If neither MAX nor MIN is locally owned, select the next most restrictive minimum constraint (MIN) and go to LOOP.

In reusable agent sets, operator diversity is expected—not every agent will instantiate every operator including the *store-received-information* and *retrieve-information* operators. Because of this, when an agent formulates and sends conflict information to another agent, there is no guarantee that the receiving agent will use that information appropriately. Therefore, although conflict information is shared willingly and cooperatively in negotiated search, agents do not depend on other agents to react in a fixed way to that information.

5.5 Agent-Level Control of Operator Application

In some sense, agent-level control is irrelevant to the discussion of negotiated search. Every agent is expected to participate coherently in whatever strategy is in effect and every strategy requires that specific tasks are to be performed by specific agents when particular conditions exist. From the TEAM framework perspective, it doesn't matter how an agent schedules its tasks as long as required tasks are accomplished at reasonable times. An agent's control structure should be appropriate to its architecture and problem-solving methodology—a rule-based lisp agent will have a different form of control than a branch-and-bound fortran agent. However, because agents may have multiple tasks that are ready to be performed at the same time, it is sometimes necessary to make decisions about how these tasks should be ordered to result in the most effective and efficient overall problem-solving performance possible. In this section, the requirements for local control are discussed.

Figure 5.1 describes coordination-state preconditions that must be satisfied before an agent can apply one of its operators to a particular solution. However, because there are multiple solution paths, and because some operators are not directly involved in solution generation (e.g., *store-received-information*), an agent may have multiple operators ready to execute at any given time. The order in which an agent schedules local operators is not mandated by either TEAM or by the negotiated-search algorithm. However, because that order affects global objects and, therefore, the environment in which all agents are embedded, the order in which particular operators are executed does affect system performance. This suggests that the effect of local scheduling on the overall behavior of the system must be considered. Some general policies for local

scheduling are useful in most situations, i.e., agents should assimilate any new information received before initiating or critiquing solutions. The degree of sophistication required in local scheduling though is highly dependent on the application and the complexity of required interactions.

The search functionality of each agent is represented modularly as negotiated-search operators. The agent designer must determine which operators can reasonably be instantiated given the characteristics of the agent. The system developer determines which agents are to be active for a particular system based on agent-set characteristics. Then, given a set of active and instantiated operators, the control problem for an agent is to decide which operator to execute at each possible time.

The decision to execute a particular operator is based on the occurrence of some event. For example, when an agent, A1, initiates a solution, it sends a message to `TMan` to add that solution to shared memory. Once it has updated the memory, `TMan` sends solution-initiation messages to other interested agent(s) in the system that a new solution was initiated in shared memory. Some agent, say agent A2, may receive several messages that it can respond to by executing local operators. For example, multiple agents may have initiated designs during the last cycle, so there may be several solution-initiation messages. As described in Section 4.3.5 during the discussion of operator-activation records, in response to the solution-initiation messages, agent A2 creates instances of *extend-solution* activation records, one for each message. Assuming that all of the operators triggered by the messages are immediately executable, A2 has to decide which operators to execute, and when to execute them. In the `TEAM` framework, each agent can execute only one memory-update operator in an agent cycle, where a memory-update operator is one that outputs a message to `TMan` that will result in changes to the shared memory.

In deciding which operators to execute at a given time, there may be several levels of readiness. The *extend-solution* operator is triggered by the addition of a new composite solution to shared memory. However, the executing agent may have additional domain-dependent constraints on its ability to respond reasonably to this event. For example, `vbelt-agent` in the `STEAM` application program instantiates the *extend-solution* operator. However, when this operator is triggered by a new composite solution, it will not actually execute until the solution has values specified for the design attributes `required-power`, `motor-drive-speed`, and `run-speed-range` as seen in Table 5.1. If the new solution does not have a value for one of these attributes, `vbelt-agent` does not put *extend-solution* on the executable agenda. Instead, it places it on a waiting queue and checks the composite solution for the required values again during the next agent cycle. Mechanical design problems that are decomposed

by components have inherent interactions among agents: some attributes of designs are constrained by the values of other attributes. However, this is a domain issue rather than a framework issue. We point it out primarily to make clear that local control decisions may be more complex than dictated by the general requirements of the framework.

Returning to our discussion of the generic local control requirements of agents, there are several activities that must occur. First, an agent must identify which of its active operators are triggered by messages it has received since its last execution cycle. Newly-triggered operators must be prioritized along with any operators that are waiting to execute from previous cycles. Prioritization can be done in many different ways, ranging from *a priori* ratings on operator types, to FIFO orderings based on the time each triggering event occurs, to sophisticated knowledge-based search for orderings that will result in optimal schedules. Newly-triggered operators can be executed before waiting operators, waiting operators can be executed first, or all operators can be (re-)rated relative to the new information. Again, TEAM does not dictate how this should be done as there is no single answer that is best in all situations. In some application systems, the order of operator execution is not very important, in others, it strongly affects system performance and/or solution optimality. Once the operators are prioritized, they execute sequentially. Multiple agent-state operators can execute, but, in *negotiated search*, an agent's execution cycle ends when it executes a memory-update operator. The effect of this is that an agent can execute multiple operators such as retrieving or storing information during a cycle, but will only generate one component proposal or critique during a cycle.

Figure 5.9 illustrates a simplified processing cycle from STEAM, showing how the TEAM mechanisms described in Chapter 4 are utilized in the execution of negotiated search. In this example, only three agents are active: pump, motor, and heat-exchanger designers.

A more complex example is illustrated in the system traces of a seven-agent problem in Section 5.2 and in Appendix A.

5.6 *Future Extensions to Negotiated Search*

In this section, we briefly discuss several extensions to the basic *negotiated-search* strategy. These extensions would enhance the functionality of the strategy by addressing issues that are problematic in specific domains.

Incorporation of Specific Conflict-Resolution Operators: Although negotiated-search captures negotiation and conflict detection, resolution, and avoidance in a

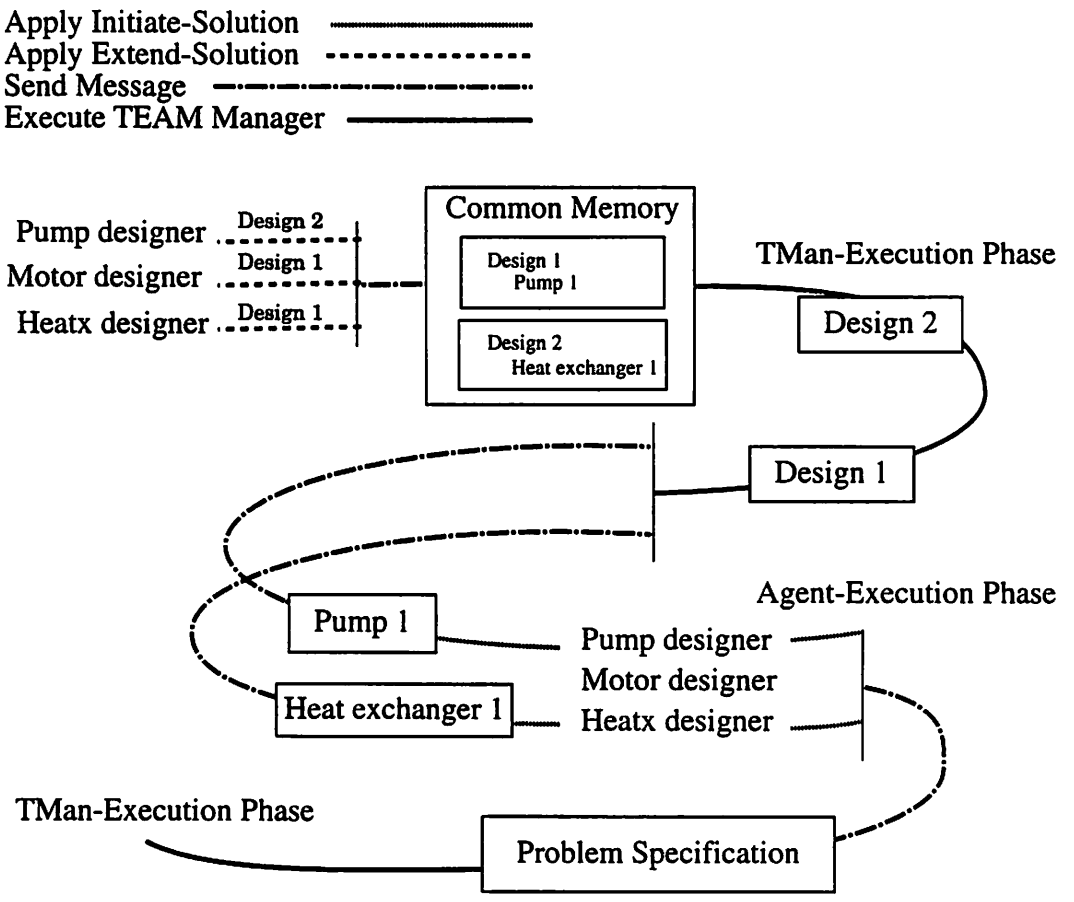


Figure 5.9. *The Execution Pattern of Negotiated Search*

flexible and generic way, it does not permit highly specific conflict-resolution techniques to be applied even when the situation warrants it. For example, say that two agents are involved in a buying/selling situation and each has a target price in mind. If both agents are fairly flexible and they are not too far apart, it might be possible to simply take the average of the target prices. If we assume that the negotiation of this price is only one component of a larger search process, then this technique is not appropriate for the entire search but only for this one component. In the current version of *negotiated-search*, it is not possible to apply a specific technique to a part of the overall problem. However, the strategy could be extended to allow agents to 'step outside' of the general protocol to deal with particular issues. An initial representation of this is shown in Figure 5.10.

Extending Negotiated Search for Greater Concurrency: In Chapter 2, we discussed the need for agents to be able to interact throughout the product-development cycle in concurrent engineering approaches to manufacturing. In that domain, it is important for diverse agents to be involved in the process as early as possible in order to avoid errors that would otherwise not become apparent until much later in the process.

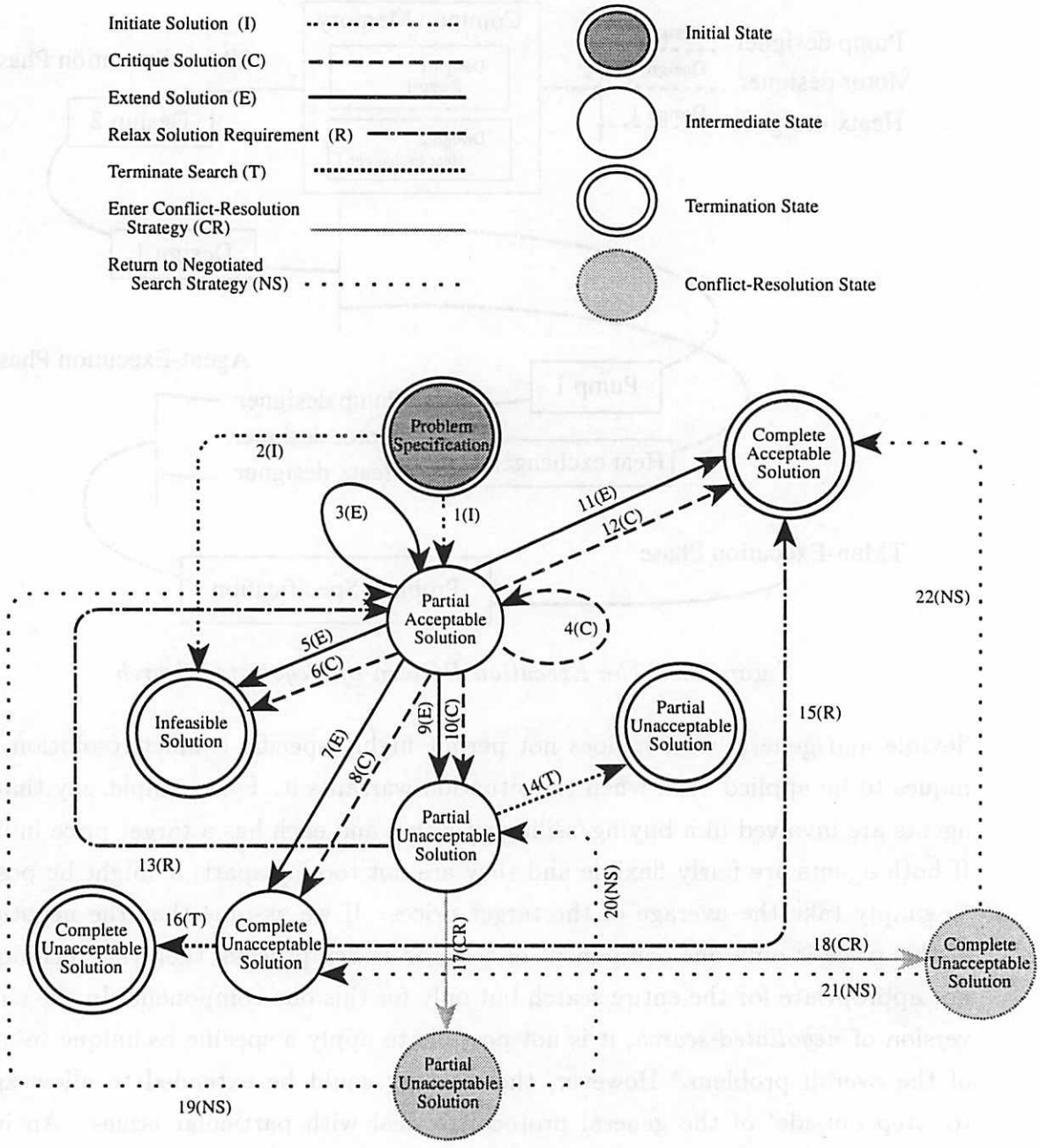


Figure 5.10. *Negotiated Search Extended to Include the Application of Specific Conflict-Resolution Techniques to Limited Problem Areas*

These errors can result in costly revisions and delays. However, there is often a highly sequential domain-dependent ordering of information that precludes an agent from being able to apply its expertise until the appropriate information becomes available. This imposes a sequential or depth-first ordering on agent involvement in the product-development process.

To address the issue of sequentiality of agent invocation, we have developed a strategy that allows agents to make *default assumptions* about information that they need to begin processing. In this strategy, it is possible for agents to begin working without complete specification of their input parameters, instead giving them the capability to make assumptions about likely values. If the assumptions are mistaken, some work may become invalid. However, they have the opportunity to speak up early in the solution-development process and communicate important constraints. This strategy both enables more concurrency in agent processing and provides explicit mechanisms for agent notification when assumptions are violated.

The changes that were made to the generic *negotiated search* strategy to accommodate default reasoning involved revising the *extend-solution* operator and adding a new operator, *check-for-assumption-error*. The generic *extend-solution* operator is not applied until key values (domain-dependent preconditions, specified by the agent instantiating the operator) have been assigned. In this version of the strategy, the *extend-solution* operator is augmented to allow the use of default-assumption reasoning by allowing an agent to proceed without having all the values for its input attributes explicitly specified. When a required attribute does not have a value specified, the agent makes its best guess about what that value will be and goes ahead with processing. A new operator, *check-for-assumption-error*, is added to ensure that assumptions that were made are checked against the actual values when those values are assigned. If an assumption turns out to be incorrect, the agent that made the assumption is notified of the error and must decide whether the violation invalidates the solution. The revised strategy has been implemented in the TEAM framework, although we have not yet done any studies on how it affects system performance. It is shown in Figure 5.11.

5.7 Summary

In this chapter, we introduced *negotiated search*, a widely applicable distributed-search strategy. This strategy specifically incorporates conflict avoidance and resolution techniques motivated by an intuitive analysis of the process of negotiation. The strategy is first described from a transition-network perspective, followed by a discussion of how that perspective can be captured in a set of implemented operators

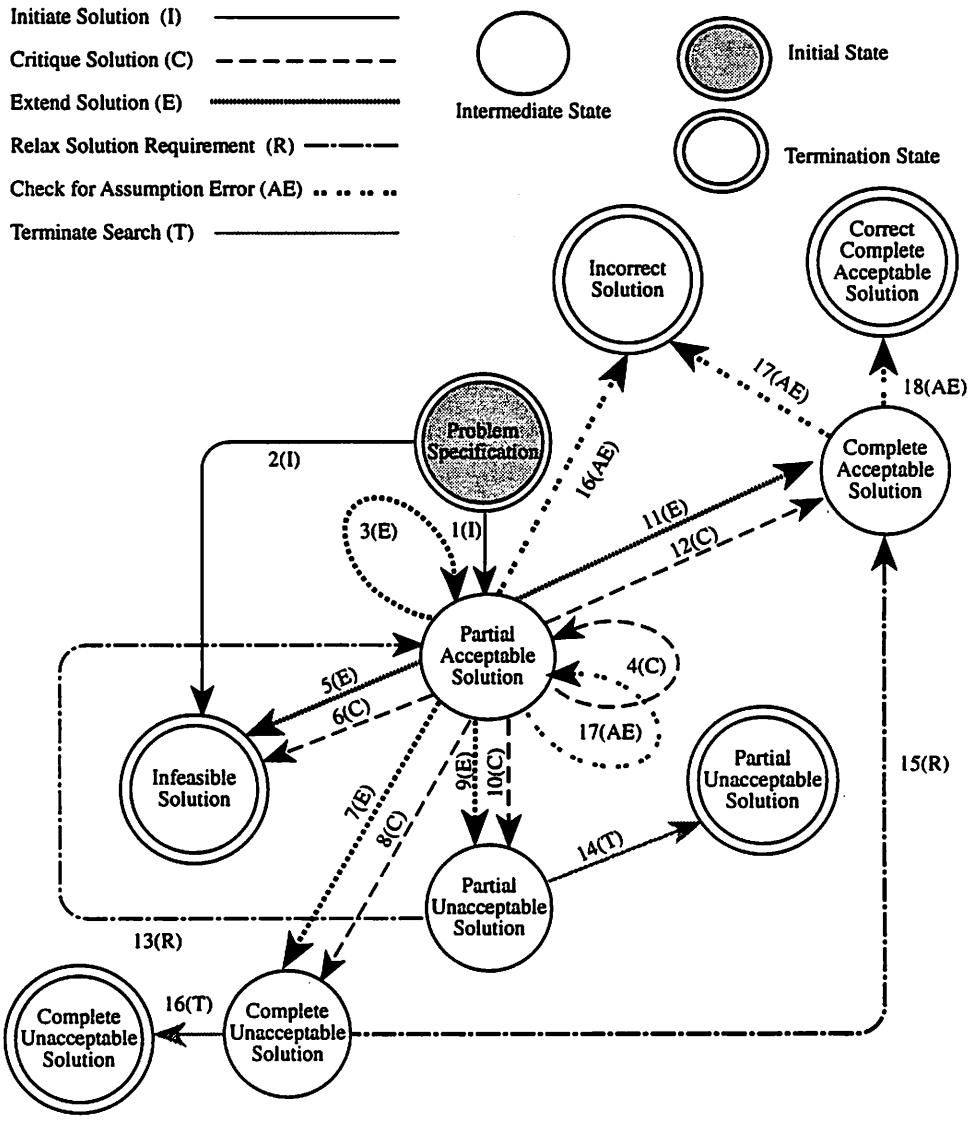


Figure 5.11. *Negotiated Search Extended to Handle Default Reasoning*

within agents. The required operators for *negotiated search* are as generic and widely available as possible: *initiate-solution*, *extend-solution*, and *critique-solution* can be implemented using whatever form of local search is resident in an agent. The *relax-solution-requirement* operator must be tailored to the type of solution requirements of the agent, but has a standard conceptual foundation. The information-assimilation operators, *store-information* and *retrieve-information* require that an agent be specifically built or augmented to allow inconsistent information to reside in its knowledge base and that the agent is able to select and apply information despite potential inconsistency. However, these operators are optional—when the operators are not implemented within an agent, search within the composite space reduces to blind search for that agent.

Because *negotiated search* is highly generic and uses information in a general way, it is a weak form of search. In this chapter, we have described extensions to that strategy that make it more suitable in particular situations and domains. In Chapter 7, we will present several different search strategies that are less generic but more powerful under appropriate conditions. We want to emphasize that negotiated search is not meant to be an appropriate search for all occasions. Rather, it is a form of search that is particularly suitable for heterogeneous reusable agents because it does not require specialized interactions from participants in the global search process. The more diverse the agents, the less likely that they can provide each other with information that very precisely delimits the search. This strategy is aimed at using whatever information is available in a constructive way.

CHAPTER 6

STEAM: PARAMETRIC DESIGN OF STEAM CONDENSERS

A primary concern in this dissertation is to investigate the structure and process of systems composed of heterogeneous and reusable agents. To clearly demonstrate the viability and effectiveness of this class of systems, we have built two prototypes in different domains. The first prototype, **STEAM**, is an application system for the parametric design of steam condensers. **STEAM** is described in this chapter. The second prototype application, AGREE, is a contract negotiation system that will be described in Chapter 7.

STEAM is built on top of the **TEAM** framework described in Chapter 4, extending its generic structures, language, and problem-solving mechanisms to address domain-level knowledge and control. We begin this chapter with a discussion of the advantages and disadvantages of multiagent cooperative design and of why the **TEAM** style of problem solving is appropriate for design. We continue with a detailed description of the **STEAM** system and with a comparison of **STEAM** to a system with identical domain knowledge but with a different architecture and control.

6.1 Multiagent Cooperative Design

Multiagent cooperative design is the process of designing an artifact using a set of specialized cooperating expert agents. Each agent has some relevant knowledge and/or skill that can be applied to the design process, but no agent has knowledge that will allow it to create a complete and acceptable design. In order to integrate their knowledge and skills, the agents must have mechanisms for communicating information about their shared resources and solutions, coordinating their actions, resolving conflicts that occur as a result of interacting or redundant problem solving, composing partial designs into a complete design, and evaluating both partial and complete designs.

There are three types of arguments that can be made for multiagent cooperative designed based on issues of 1) knowledge heterogeneity in design; 2) software engineering principles; and 3) system performance criteria. With respect to knowledge heterogeneity in design, we present the following characteristics which make the design process particularly amenable to a multiagent approach:

- Design of complex artifacts is inherently a collaborative process: the complexity of many design problems is beyond the scope of any single technology, where each technology represents an area of expertise.
- Each technology has its own specialized language, much of which is not used or understood outside of the technology. For example, the language of coordinates used in determining the spatial layout of a room is not relevant to describing the function of the room.
- Different models of the problem or of subproblems are appropriate for different technologies. For example, different technologies may represent the same object at different levels of abstraction, or as having different attributes based on the importance of various attributes within the technology.
- Different technologies use specialized generative and analysis methodologies which may depend strongly on a specialized representation of concepts, for example, finite element analysis or numerical analysis.
- Each technology has performance or evaluation criteria that are distinct from those of other technologies, e.g., the evaluation of an artifact from a functional perspective may be quite different than the evaluation of an artifact from an affordability perspective.
- The processes used within a particular technology may require specialized hardware and software resources, such as floating-point accelerators or graphical-representation tools.

From a software-engineering perspective, a multiagent cooperative design system provides advantages over a large monolithic system designed to accomplish the same task(s). A compelling argument for choosing a modular expert-agent approach is the ease of building, maintaining, testing, and debugging relatively small agents. In practice, the design of complex systems is usually done by decomposing the design problem into smaller, more manageable subproblems that are inherently suitable to a multiagent architecture.

In a multiagent system, the expertise pertaining to a particular subproblem can be captured in a single agent without the need for maintaining consistency with all other knowledge that will eventually be embodied in the system. In nontrivial design domains there are inherent conflicts among different technologies. These conflicts stem from the lack of consistent theories across disciplines, the diversity in performance measurements imposed by each discipline, and the different models and methods used. By building agents that are internally consistent and using a framework such as TEAM to handle conflict among agents, the disparity among technologies can be dealt with more flexibly, smoothly, and naturally.

Multiagent systems may improve solution quality relative to monolithic systems because they bring different perspectives to bear on the problem. Assuming an equal degree of expertise and search sophistication between a single-agent and multiagent system, the multiagent system is likely to explore more of the space of potential solutions since it will be evaluating solutions from different perspectives. In a monolithic system, there is a single bias that will guide the generation and evaluation of potential solutions; in multiagent systems, solutions may be generated and evaluated by multiple agents with different biases. When solutions are generated by different agents, there is a better chance of achieving globally optimal rather than locally optimal solutions.

Finally, from a system performance perspective, cooperative multiagent design systems can take advantage of inherent parallel computing potential. Much of the processing that occurs in design is local to a specific agent and can be performed without tight synchronization. Therefore, by enabling concurrent execution of multiple agents, the efficiency of the system can be improved.

Although there are clearly advantages to multiagent cooperative design, there are inherent disadvantages as well. The ultimate effectiveness of a system depends on whether the costs of overcoming the disadvantages are outweighed by the benefits that result. The biggest disadvantage to using multiagent systems is the need for sophisticated integration machinery (conflict resolution, coordination) to achieve coherent cooperative behavior. However, we believe that the more complex the domain, the more benefit will be derived from using a multiagent system. For simple problems, the software development costs of agent integration and coordination outweigh potential improvements in the costs of design, testing, and maintainability. For more difficult problems, the development costs incurred due to the increased complexity of the integration software become relatively small, especially with respect to the long-term costs of enhancing and correcting the overall system. Sommerville suggests that implementation effort represents the smallest part of software development with more effort going into both design and testing. Furthermore, for large, long-lived, software systems, he estimates that maintenance costs normally exceed development costs by factors which range from two to four [Sommerville, 1989].

6.2 The STEAM *System*

In the following sections, we describe the application domain of the STEAM system and the general organization of the system including task decomposition and agent concurrency issues. We continue with a detailed description of STEAM in terms of its agents, extensions to the TEAM language, coordination strategy, solution-acceptability

policy, and termination policy. By examining specific instances of general TEAM attributes, we illustrate how the separate pieces of a multiagent architecture are integrated into a coherent system that meets the requirements of the domain and the performance criteria of the system designer.

6.2.1 *The* STEAM *Domain*

Much of the application knowledge in the steam condenser domain was originally developed by Meunier and Dixon for use in an *iterative respecification* system, IRSys [Meunier, 1988].[1] We finish the chapter by comparing STEAM to IRSys. STEAM and IRSys share a common domain, the design of steam condensers. They both decompose a steam condenser into a set of components, namely, a pump, heat exchanger, motor, platform, shaft and v-belt. Figure 6.1, taken from [Meunier, 1988], shows the representation of a condenser that is used in both systems. Each component in the

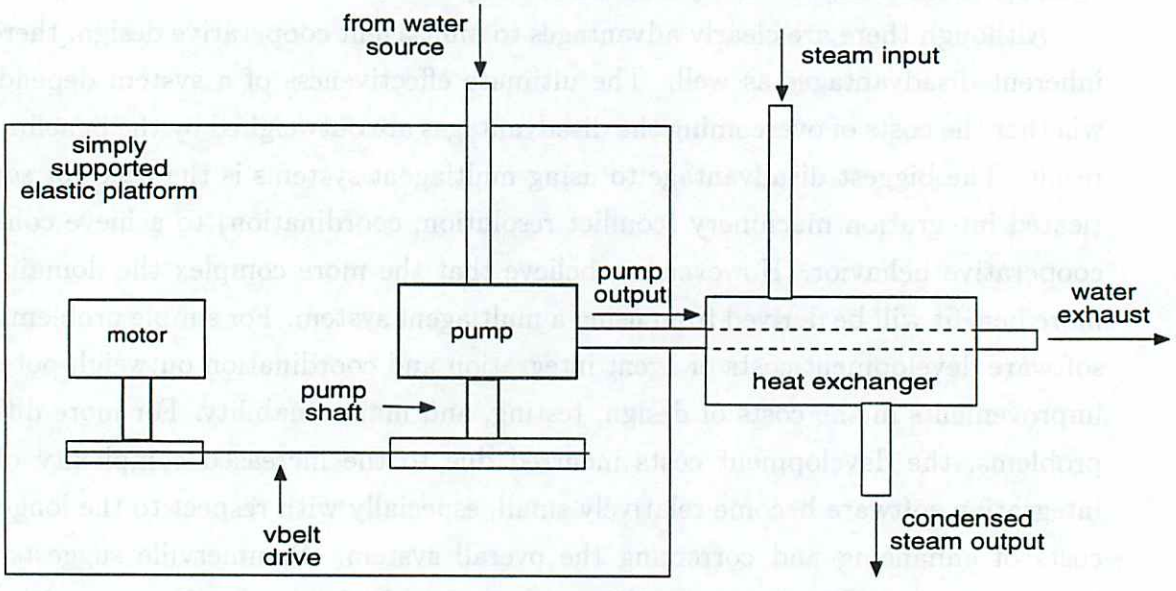


Figure 6.1. *A Steam Condenser*

condenser is designed by a separate agent. For example, pump-agent produces pump designs. The components are independent except for shared interface parameters, e.g., pump-agent and motor-agent share a *required-power* parameter because each pump model requires a minimum level of power to be output from the motor. The number of shared interface parameters between agents is small (no more than 3 parameters

[1]Meunier and Dixon did not provide an acronym for his system and since it is cumbersome to frequently repeat an entire descriptive phrase, in this dissertation, we will use the acronym IRSys to refer to that system.

are shared between any two agents) and no parameter is directly shared by more than two agents, although the value of a parameter shared between two agents, say A1 and A2, may constrain the value of a parameter shared by one of those agents with another agent, e.g., A1 and A3. We believe that the techniques used in this domain will scale up to situations where the interactions between agents are more intensive, but we haven't yet looked at a domain where this is true.

STEAM performs parametric design, a type of design in which the general form of the artifact being designed is known, but the designer must find values for a set of variable parameters. It should be noted that our interest in steam-condenser design is not in generating a production-level design system, but rather in studying the heterogeneous and reusable-agent model we have proposed in a suitable domain. We are not concerned with the precision of the domain knowledge or efficiency of the local search methods used by the agents. We believe that it would be possible to build more "intelligent" agents that could handle subtleties of the design process that are not currently managed in an effective way in either IRSys or STEAM. However, although good domain modelling, search, analysis, and evaluation techniques are important to building production-level systems, here we are looking at the more general issues of integrating multiple intelligent agents into effectively functioning systems.

6.2.2 Task Decomposition

The task decomposition of heterogeneous multiagent systems is based on the expertise of the systems in the agent set. There are seven agents in STEAM. Six of the agents design individual components of the steam condenser: these are `pump-agent`, `heat-exchanger-agent`, `motor-agent`, `vbelt-agent`, `shaft-agent`, and `platform-agent` (see Figure 6.1). A critic agent, the `frequency-critic`, is used to check a system characteristic (natural frequency) that requires expert analysis of system-level attributes, requiring results from several agents. Notice that there are no agents in this system that are redundantly solving the same problem. We don't have two pump designers for example. In globally cooperative systems, redundancy is usually not a desirable attribute, but there are situations where it is useful such as in multiperspective reasoning [Werkman, 1992]. We will see an example of redundant problem solving in Chapter 7 where two agents are negotiating a buy/sell contract.

By default, the STEAM agents are organized in a flat structure with no agent either having control of another agent's problem solving or having more power in determining solution acceptability. In Chapter 9, we will describe experiments in which the organizational structuring of the agent set is manipulated; however, in the discussions in this chapter, we assume an anarchical organization.

Although the organizational structure in STEAM is flat and, therefore, does not impose an execution order on the agent set, there are domain-level input/output parameter dependencies that force a partially sequential ordering of agent execution. The default distributed search strategy used by STEAM is negotiated search which is illustrated in Figure 5.1. In this strategy, designs are initiated by applying the *initiate-solution* operator to a problem specification. Two agents are capable of initiating solutions, `pump-agent` and `heat-exchanger-agent`, and the two possible agent-execution orders that result are shown in Figure 6.2.

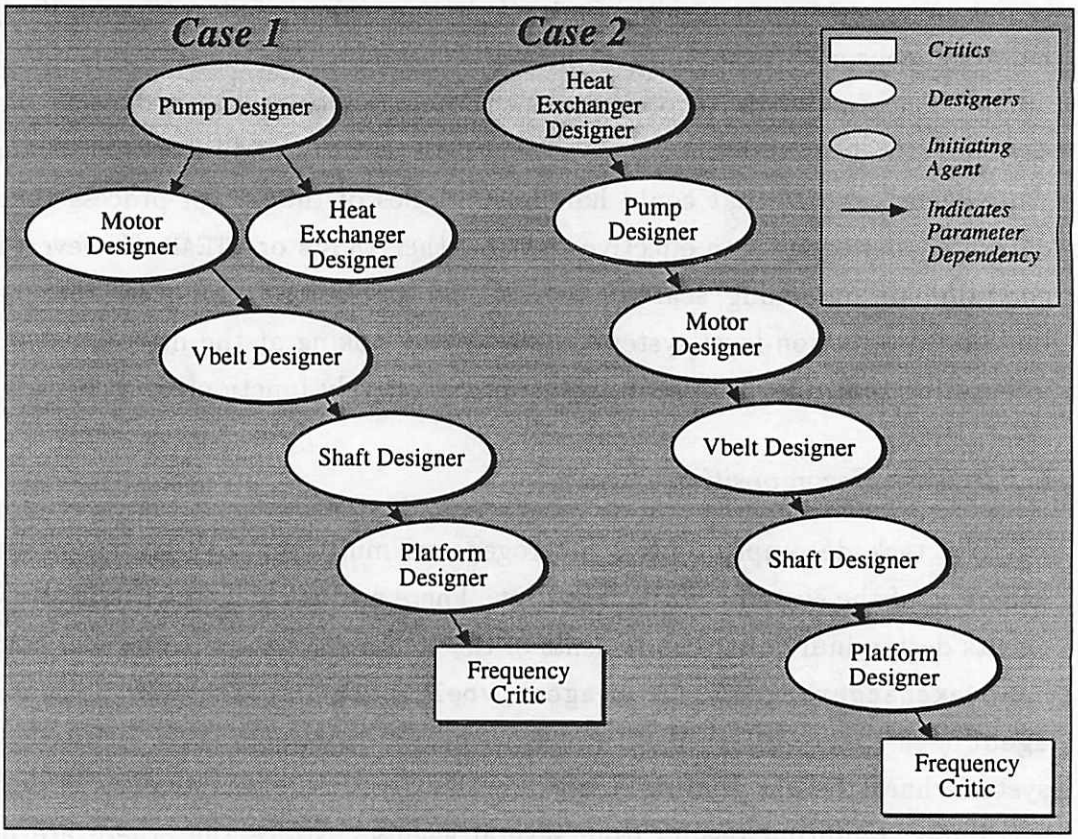


Figure 6.2. *Agent Execution Order in* STEAM

The decision about which agents should be able to initiate solutions is based on several factors: 1) the search capabilities of the agents; and 2) the domain-level parameter dependencies that exist in the problem. An example input/output parameter dependency exists between `pump-agent` and `motor-agent`. `Pump-agent` must run before `motor-agent` because it generates a value for the parameter *required-power* which is used by `motor-agent` to select a motor model and, for this reason, `motor-agent` is unable to initiate solutions independently. In Chapter 5, we discussed the use of default assumptions as a mechanism for allowing agents to execute in a less

sequential manner. Using default assumptions, it would have been possible to build solution-initiation capabilities into other agents, such as `motor-agent`, and this is in fact being done in ongoing research. However, the functionality required to initiate a solution is somewhat different from the functionality required to extend an existing solution. Due to time constraints, we felt that the extra work required to supply that functionality was not warranted for the dissertation research.

Returning to the description of the `STEAM` system, as noted above, there are domain-level parameter dependencies among agents that enforce a mainly sequential ordering on agent execution along a particular solution path. However, this does not mean that only one solution at a time can be in progress. To exploit the efficiency benefits of parallel agent execution, it seems quite natural to explore multiple solution paths simultaneously. In Figure 6.3, we illustrate the simultaneous development of multiple solutions over eight time units with four active agents. Two of the agents initiate solutions: `pump-agent` and `heat-exchanger-agent`. All agents, include `pump-agent` and `heat-exchanger-agent`, can extend solutions initiated by other agents following the agent-execution orderings shown in Figure 6.2. This is a simplistic

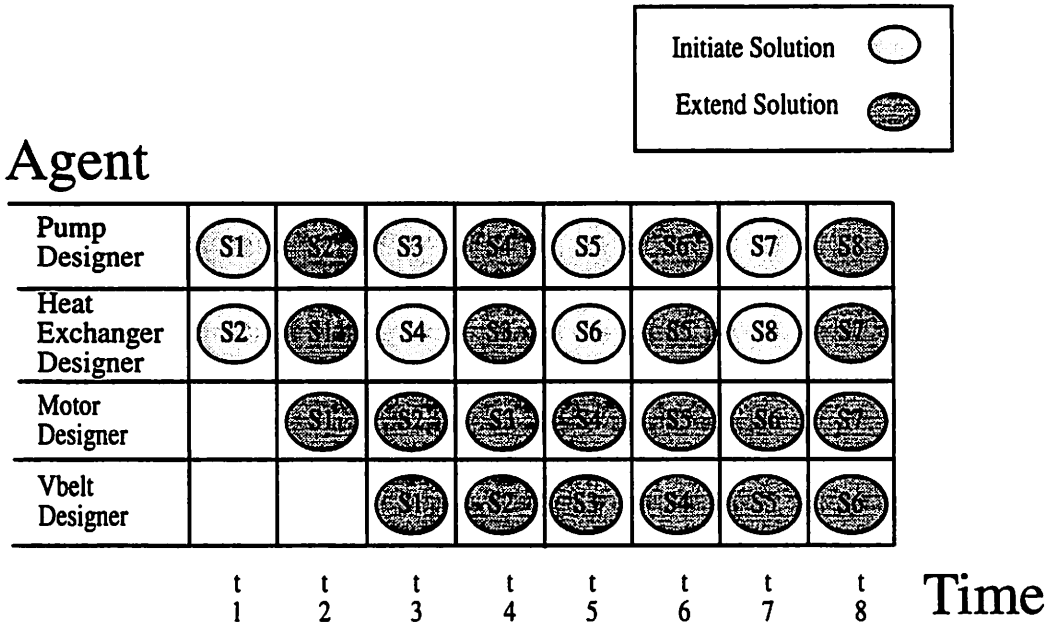


Figure 6.3. *Simultaneous Solution Development in* `STEAM`

view of simultaneous solution development because it does not address the issues of incremental solution evaluation and acceptability. In the figure, every solution remains acceptable throughout problem solving. In reality, some of these solutions might be dropped because they are found to be infeasible, or extension of some

solutions might be suspended because they become unacceptable. These issues were discussed in detail in Chapter 5.

Besides efficiency, simultaneous exploration of multiple solution paths has other benefits: increasing solution space coverage and allowing direct comparison of different potential compromise solutions. Increased solution space coverage is seen as particularly useful when multiple agents are able to initiate solutions based on their local views of the shared parameter spaces. They are likely to have different initial viewpoints about what points in the parameter spaces will contribute to a good design. This diversity in viewpoints ensures broad coverage of potential starting points for a design, leading to a thorough exploration of possibilities.

Expanding multiple solution paths simultaneously also supports direct comparison of potential solutions. The steam-condenser design domain is overconstrained by the full agent set in all but the most trivial cases (as are almost all design domains). Therefore, basically all solutions are compromises: some agent(s) has had to relax solution requirements. It is often difficult to tell what the effect of a particular relaxation will be on a complete solution since the mapping between parameter values assigned at the local level and global performance evaluation is not direct. In other words, it is not usually possible to predict the effect of changing the set of legal values for a parameter at a particular agent on the overall evaluation of the complete solution. For example, if `pump-agent` relaxes a constraint on `water-flow-rate`, it may lead to the selection of a pump model that requires less power than was possible in the unrelaxed state. This leads to the selection of a less expensive motor, which in turn leads to a less expensive steam-condenser design. So, although `pump-agent` is less satisfied with the pump it has generated, the overall design is considered better if cost is used as the performance criteria for judging designs. Although not all compromises end up so happily, in domains where compromise is inevitable, it is certainly useful to be able to generate and compare different compromises in a controlled way.

6.2.3 Extending the TEAM *Language*

In this section, we show several examples of how the TEAM language described in Chapter 4 is extended for STEAM. We will describe several domain-specific objects as specializations of their generic counterparts, e.g., designs as specializations of composite solutions.

A solution in STEAM is a design, which is represented as a specialization of the `composite-solution` object in Figure 4.4. In order to make it domain-specific, we added slots to represent attributes of a steam condenser, such as system frequency and shared parameters that agents need to communicate, such as the horsepower of

the motor. WE also added slots used to link the design to its specific components, such as pumps, motors, etc. The design object used in **STEAM** is shown in Figure 6.4.

In addition to the representation of composite solutions as designs in **STEAM**, each domain-specific agent proposal extends the generic proposal object shown in Figure 4.5 to incorporate domain- and agent-specific characteristics. These extensions take the form of additional slots, such as water-flow-rate, or initialization of existing slots, such as the assigned value of the parameters slot. The parameters slot is a static slot in that its value does not change over the course of problem-solving. Each agent, however, has a unique value for the slot. Figure 6.5 illustrates an instance of a completed pump proposal created by the pump agent during a run of the system.

In Figure 4.7, we showed a generic conflict object for the **TEAM** language. In Figure 6.6, we show an instance of a conflict that occurred during a run of the system. This conflict was detected by **pump-agent** when its internal constraint, (minimum-head < 342.14), was violated in a design it was attempting to extend. This constraint is listed in the figure for clarity, but in fact, the slot **violated-constraints** actually contains a *pointer* to the constraint object which contains other information as well such as the flexibility of the constraint. The parameter name/value list contains the names of parameters with values that violate known constraints along with the assigned value of each parameter. For example, in the figure, the assigned value of the minimum head parameter is 358.75 while the violated constraint has a maximum boundary of 342.14. The design slot contains a pointer to the shared design being extended while the proposal slot contains a pointer to the proposal that was created to extend the design. Note that this proposal will have a value of minimum head of 358.75 and will be initially unacceptable since it violates a local constraint.

In Figure 4.11, the form of a generic message object in **TEAM** is shown. The default search strategy for **STEAM** is the negotiated-search strategy shown in Figure 5.1. Figure 6.7 depicts an instance of an initiate-solution message, a message type used specifically by the negotiated-search strategy. This type of message is sent by an agent that has executed an *initiate-solution* operator and produced a proposal that is to be used as the base proposal for a new design. The initiating agent sends the message, containing the new proposal, to **TMan**. **TMan** then responds to the message by creating a new design incorporating the proposal specifications. Other message types used in the **STEAM** system include extend-solution, system-completion, component-required, conflict-detected, and wait.

The **instantiated-strategy** object used to install negotiated search as the default search strategy in the **STEAM** system is shown in Figure 6.8. This is a

```
Design
  Blackboard Path       % The blackboard path to specify how to
                        %    the design on the blackboard
  Type                  % Steam condenser design
  Cost                  % The cost of the complete steam
                        %   condenser
  Agents                % The list of agents that have worked on
                        %   this design to date
  Evaluations           % The list of agent/evaluation pairs for
                        %   this design
  Agent Acceptability   % The design's acceptability to all
                        %   agents that have currently evaluated
                        %   it
  Acceptability         % The global acceptability as defined by
                        %   the prevailing acceptability policy
  Component Set Completeness
                        % Does this solution have all its
                        %   components?
  Completeness          % Have all agents either proposed
                        %   components for or critiqued this
                        %   design?
  ID                    % A unique identifier associated with this
                        %   design
  Conflicts             % A list of conflicts associated with this
                        %    design
  Creation Time         % The creation time of this
                        %    design
  Initiating Agent      % The agent that developed the base
                        %    proposal for this design
  Platform Load         % The total weight of the pump, motor,
                        %    shaft, and vbelt (used by frequency
                        %    critic)
  System Frequency      % The system natural frequency
  Available Head        % The available head from pump-agent
  Minimum Head          % The minimum head required for heatx-agent
  Water Flow Rate       % The water flow rate between pump- and
                        %    heatx-agent
  Motor Drive Speed     % The drive speed from motor-agent
  Load Speed            % The run speed from pump-agent
  Required Power        % The power required by pump-agent from
                        %    motor-agent
  Horsepower            % The horsepower provided by the motor
  Platform Thickness    % Platform thickness
  Stiffness             % Platform stiffness
  Required Capacity     % Required capacity of the condenser
  Maximum Platform Deflection
                        % The maximum allowable platform deflection
  Platform Side Length  % The required platform side length
  Heat Exchanger        % A link to a heat exchanger design
  Pump                  % A link to a pump design
  Motor                 % A link to a motor design
  Platform              % A link to a platform design
  Shaft                 % A link to a shaft design
  Vbelt                 % A link to a vbelt design
  Frequency Critique    % A link to the frequency critique
                        %    of this design
```

Figure 6.4. *A Design Object*

```
Pump Proposal
  Initiating Agent: pump-agent
  Evaluation: good                 % The agent's local evaluation of
                                   %   its own proposal
  Acceptability: acceptable        % Will this agent accept a design
                                   %   incorporating this proposal?
  Cost:  227.00                    % Cost for this proposal
  Weight: 50.4375                  % The weight of the proposed pump
  Conflicts: nil                   % A list of conflicts associated
                                   %   with this proposal (none
                                   %   found)
  Parameters:                      % A list of output parameters
     (water-flow-rate              %   whose values should be copied
      available-head               %   into designs (some attributes
      required-power               %   of pump proposals are strictly
      load-speed)                  %   local and should not be
                                   %   copied)
  Creation Time: 10
  Design: design_12                % A link to the design this
                                   %   proposal extends
  ID: heatx_id_59                  % An ID number associated with the
                                   %   original design and all
                                   %   related proposals
  Model:  model4-impeller42        % Domain-specific information
  Water-flow-rate:  130.5          %   about the selected pump
  Maximum-head: 224.635
  Available-head: 179.135
  Required-power: 8.89
  Load-speed: nil
  Run-speed-range: (2700 3300)
  Pump-run-speed: 3000
```

Figure 6.5. *An Instance of a Pump Proposal from* STEAM

```
Conflict
  Originator: pump-agent      % The agent that detected the
                              %   conflict.
  ID: heatx_id_32             % The identifier associated with
                              %   the design and its associated
                              %   components.
  Violated Constraints: (minimum-head < 342.14)
                              % A pointer to a violated constraint
  Parameter Name/Value List:  (minimum-head 358.75)
                              % A list of conflicting parameters and
                              %   their actual assigned values (used
                              %   for determining relevance by the
                              %   receiving agent).
  Proposal: pump_2            % A link to the proposal that
                              %   triggered the conflict.
  Design: design_14           % A link to the composite solution
                              %   associated with this conflict.
```

Figure 6.6. *An Instance of a Conflict from* STEAM

```
Message
   Type:  initiate-solution
   Content: pump-proposal1 % A pointer to a base proposal
   Sender: pump-agent      % The agent sending the message
   Recipient: tman         % This type of message is intended for
                           %   the framework manager
   Time-of-transmission: 1 % The time the message was sent
```

Figure 6.7. *An Instance of an Initiate-Solution Message*

specialization of the generic instantiated-strategy object shown in Figure 5.2 and reflects the actual operator/agent assignments of the STEAM system.

```
Instantiated_Strategy
   Strategy_name:       negotiated-search
   Manager:             nil
   Type:                default
   Contract:            nil
   Agent_role_pairs     ((pump-agent
                            (solution-initiator
                             solution-extender))
                          (heat-exchanger-agent
                            (solution-initiator
                             solution-extender))
                          (motor-agent
                            (solution-extender))
                          (vbelt-agent
                            (solution-extender))
                          (shaft-agent
                            (solution-extender))
                          (platform-agent
                            (solution-extender))
                          (frequency-agent
                            (critic)))
   Event:               nil
   Activation State:    active
   Search State:        open
   Suggested Strategy:  nil
   Number of Acceptable Solutions: 0
```

Figure 6.8. *The Initial Instantiated-Strategy Object for* STEAM

6.2.4 Solution Acceptability in STEAM

Global solution acceptability in STEAM is a combination of local and global acceptability and/or evaluation on design cost. The policy used in STEAM is that in order for a design to be acceptable, it must be acceptable to all agents and its global evaluation must be acceptable. Agent acceptability is determined locally by the evaluating agent. We have used the convention that an agent maintains a variable called its *flexibility-level*. When an agent evaluates a solution, if it has any

violated constraints that are less flexible than the current value of the flexibility-level variable, the solution is considered *unacceptable*, otherwise it is *acceptable*. The value of the flexibility-level variable changes as requirements are relaxed, and as this value changes, the local acceptability of a solution may change as well. Notice that this local-acceptability policy maps solution evaluation into a trinary function with a range of (*acceptable*,*unacceptable*, and *infeasible*). Acceptability is not dependent on a specific numeric rating on the proposal—in other words, an agent may rate a proposal poorly and still find it acceptable or vice versa. Just as in many day-to-day situations that people face, these agents must sometimes accept proposals that aren't locally appealing in order to progress globally.

In order to derive the global evaluation of a solution, the TEAM framework applies a user-defined function to completed solutions that results in a single numeric rating. In STEAM, this function simply returns the cost of the design to be used as a rating. The user may also supply a threshold value for global solution evaluations. If a threshold value is supplied, the rating must be above it, for maximization problems, or below it, for minimization problems, in order for the solution to be acceptable. The code used to implement this acceptability policy is presented in Appendix C.

It should be noted that this is only one policy for determining acceptability, and that other policies might be more suitable in other situations. For example, in locally cooperative problems, global acceptability is not necessary and, therefore, the acceptability policy might simply be that no agent finds a solution unacceptable. Other policies might not use a binary system for local evaluations and instead require that all local evaluations be numeric and be above some threshold or that the average of local evaluations be above some threshold. Policies do not have to be democratic: a policy could give the power to determine or influence solution acceptability to a subset of agents in the agent set. For example, a policy might state that agents a_1 and a_2 must rate a solution as acceptable while other ratings from the agent set are ignored or downgraded. The selection of an acceptability policy for a particular application system is domain dependent and not well understood. However, TEAM provides the flexibility to implement whatever policy is chosen.

6.2.5 The System Termination Policy in STEAM

The default termination policy for STEAM is that when at least three complete acceptable designs have been generated, TMan applies the *terminate-search* operator. No new solutions are initiated at this point and no further requirements are relaxed. However, the system continues processing existing partial acceptable solutions. The

system finally terminates when there are no further partial acceptable solutions to work on. The code used to implement this policy is listed in Appendix D.

Termination policies can be customized to the environment just as acceptability policies can be. For example, the default policy for STEAM returns at least three alternative designs to the user. This policy allows the user to make final decisions on which design should be finally selected. However, it is trivial to change the policy to switch phases after only one design is found. Likewise, it is trivial to halt the system immediately when some user-specified number of designs are found, rather than allowing it to finish partial acceptable designs. The TEAM framework is flexible with respect to termination policies and the implementation of a policy is modularized in such a way that a policy can be easily modified to fit the domain.

6.3 A Comparative Description of IRSys

In this section, we briefly describe IRSys and compare methodologies and results from the two systems. IRSys takes a hierarchical approach to task decomposition, unlike STEAM which uses a flat decomposition. Individual *designers* solve terminal subproblems which are then integrated into subsystems (controlled by *managers*) which are then integrated into a complete system. A manager integrates subproblem solutions, evaluates the resulting system (or subsystem), and, if the resulting system is not acceptable, respecifies the problem for each of its subordinates to solve again. A manager's responsibility is to write problem specifications for, and resolve conflicts among, subordinate managers or component designers. A component designer's responsibility is to produce a component that meets the specifications it is given. This is in contrast to STEAM where a component designer is responsible not only for producing components, but also for rewriting its own specifications in response to information from other agents, and resolving conflicts as necessary.

In IRSys, the steam condenser is divided into three subsystems: the pump system, the pump/platform system, and the heat-exchanger system as shown in Figure 6.9. The agent-execution order in IRSys is shown from left to right in Figure 6.9 within each level. When a manager is invoked, it invokes each of its subordinates in turn. Control passes from the manager to a subordinate, back to the manager, to the next subordinate, etc., until the complete subtree beneath the manager has executed. Control then passes from the manager, up the hierarchy to its manager, and then down to the next manager or designer on the right if one exists.

IRSys develops a single design: first attempting to create a "rough draft" and, if successful, refining that design to optimize global performance criteria. STEAM does not explicitly separate draft and optimization stages. As information about design

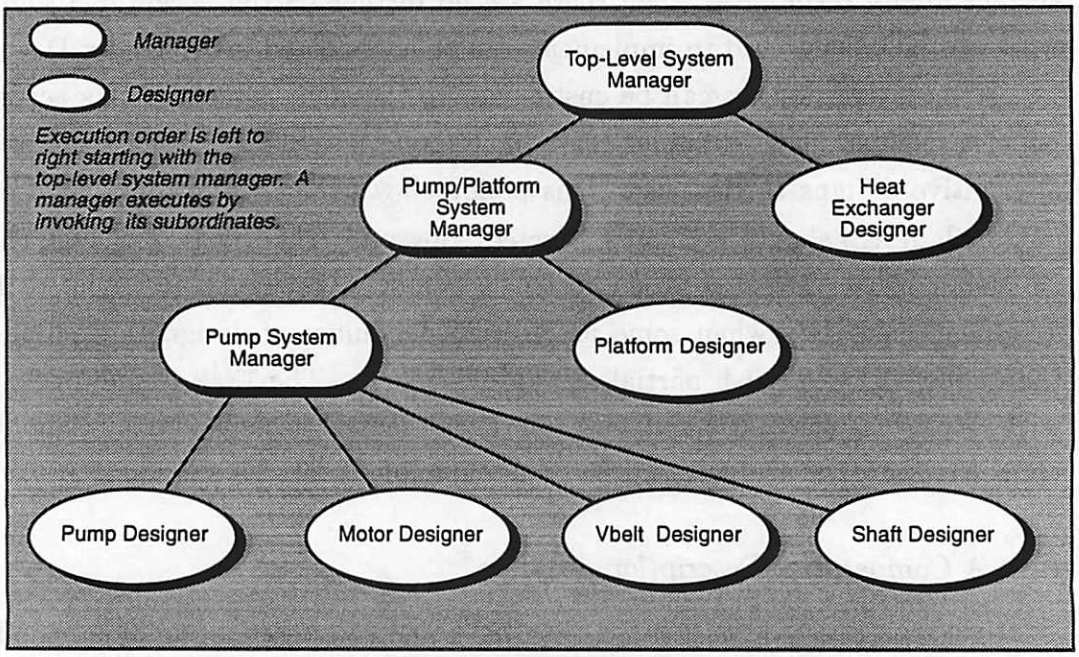


Figure 6.9. *The Hierarchical Task Decomposition of* IRSys

requirements propagates throughout the agent set, it is expected that new initial designs will improve.

6.4 *Comparison of Results from* IRSys *and* STEAM

We ran both systems on the same data sets. The first four data sets were taken from [Meunier, 1988], and the rest were generated by selecting values for each of the problem specification parameters, *required capacity*, *platform side length*, and *maximum platform deflection*. The values were randomly generated from within the feasible value range for each parameter. We show results from 101 data sets, one of which resulted in a system failure. These experiments were run on a Texas Instruments Explorer 2. The code for both systems was written in Common Lisp. The complete results from these experiments are contained in Appendix E. Figure 6.10 summarizes the comparative solution quality between the two systems of the 100 data sets that resulted in solutions, measured in terms of the lowest-cost design found by each system. In this figure, the cost of the minimum-cost design generated for each problem specification is shown for IRSys and STEAM. The average cost for IRSys was $9605.95 while it was $8375.77 in STEAM. On average, therefore, the STEAM designs were 12.81 percent less costly than the IRSys designs. It is not known how well either system performs relative to true optimality as measured by cost but we suspect that there is a ceiling effect in the data. In other words, we believe that the designs

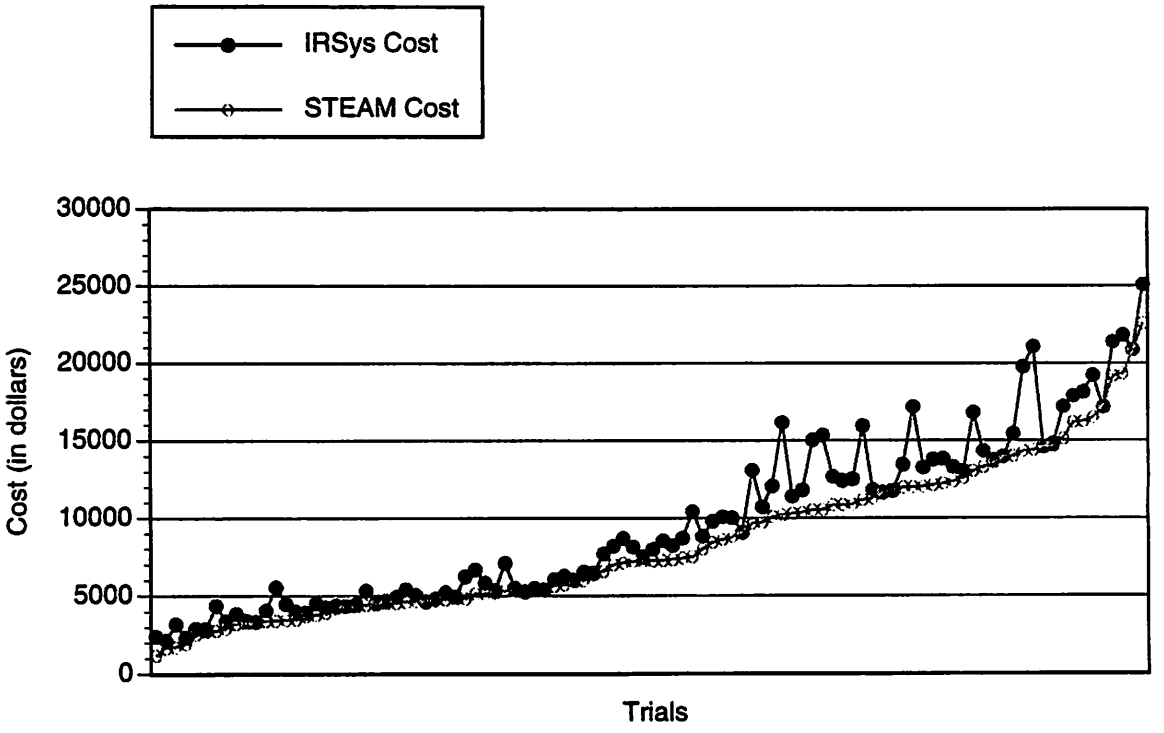


Figure 6.10. *Comparison of Solution Quality in* IRSys *and* STEAM

produced by both IRSys and STEAM are relatively close to optimal and, therefore, there is a fairly tight limit to the possible gains that can be achieved.

In addition to comparing design quality as represented by design cost, we looked at two measures of system performance. First, we compared the total number of component proposals that were generated during a run of the system. This is a crude measure of comparison, since both systems spend some portion of their time in management and control which will not be reflected in this measure. STEAM spends time in framework control and in other agent capabilities (assimilating information, for example) and IRSys spends time in its manager nodes. However, this measure provides at least some gross comparison of the efficiency of the system in terms of the number of component designs produced by each agent. This measure is particularly important in domains where the cost of generating proposals is very high. As can be seen in Figure 6.11, STEAM agents design far fewer components overall. The average of total component designs attempted per problem specification for IRSys is approximately 204 and the average number of component designs attempted per problem specification for STEAM is approximately 84, a reduction of 58.82 percent.

The second measure of system performance for these experiments was the real time, in seconds, of each system run. Figure 6.12 shows the comparison of run times for the two systems. IRSys averaged 240.21 seconds per run while STEAM averaged

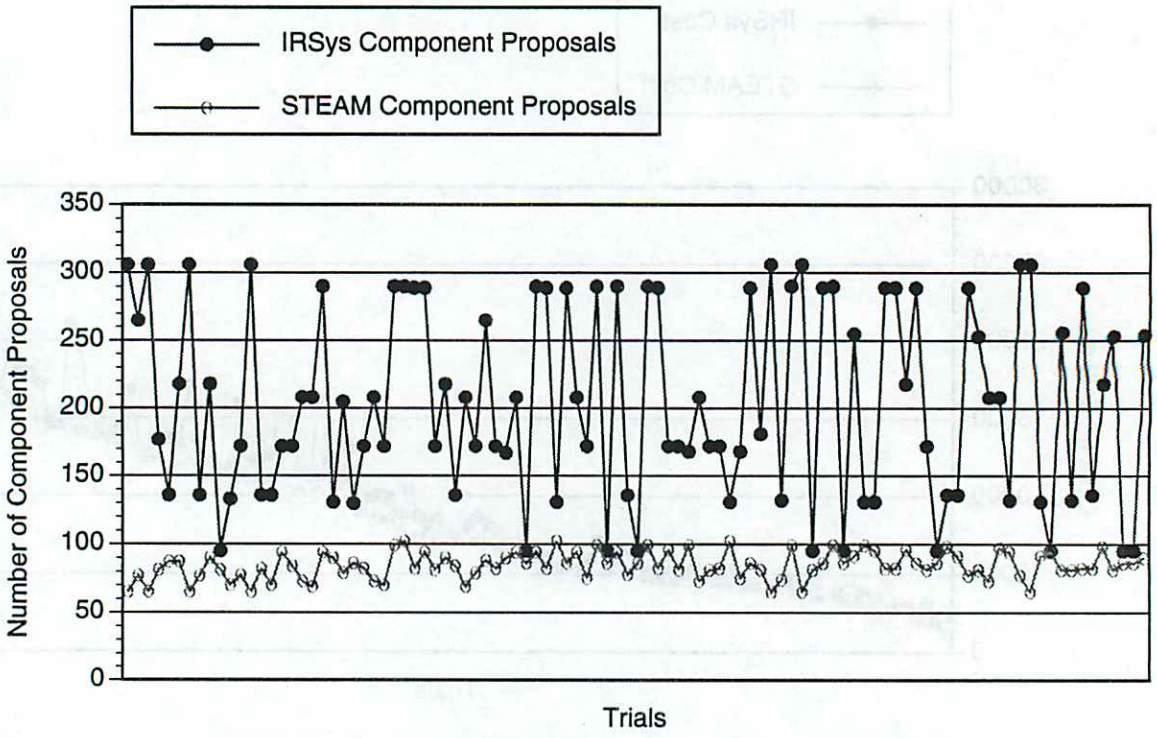


Figure 6.11. *Comparison of the Number of Component Proposals Generated in* IRSys *and* STEAM

114.53 seconds for an average 52.32 percent reduction in the amount of time spent to run the system.

In the one case in which an infeasible problem specification was provided, STEAM produced fewer component proposals (2 proposals for STEAM, 12 for IRSys) but took longer to recognize the failure (43.48 seconds for STEAM, 12.27 seconds for IRSys). First, we examine why fewer proposals were produced in the STEAM system. in STEAM, pump-agent and heat-exchanger-agent both attempted to initiate solutions directly from the problem specification. Pump-agent generated one component proposal during the first agent-execution cycle. Heat-exchanger-agent determined during the first agent-execution cycle that the *required-capacity* value was infeasible. It returned an infeasible component proposal and system processing terminated immediately. In IRSys, however, the entire pump and floor system was designed before heat-exchanger-agent was executed (see Figure 6.9). Because the constraining information was confined to the last agent to execute, it was impossible to tell that the system would fail until all other agents had executed. In a small design system like IRSys this is not a major problem. However in a larger context, such as product development, the invocation of each agent may be very costly and it is important to avoid wasted effort. It is definitely to the system's advantage to have any agent that

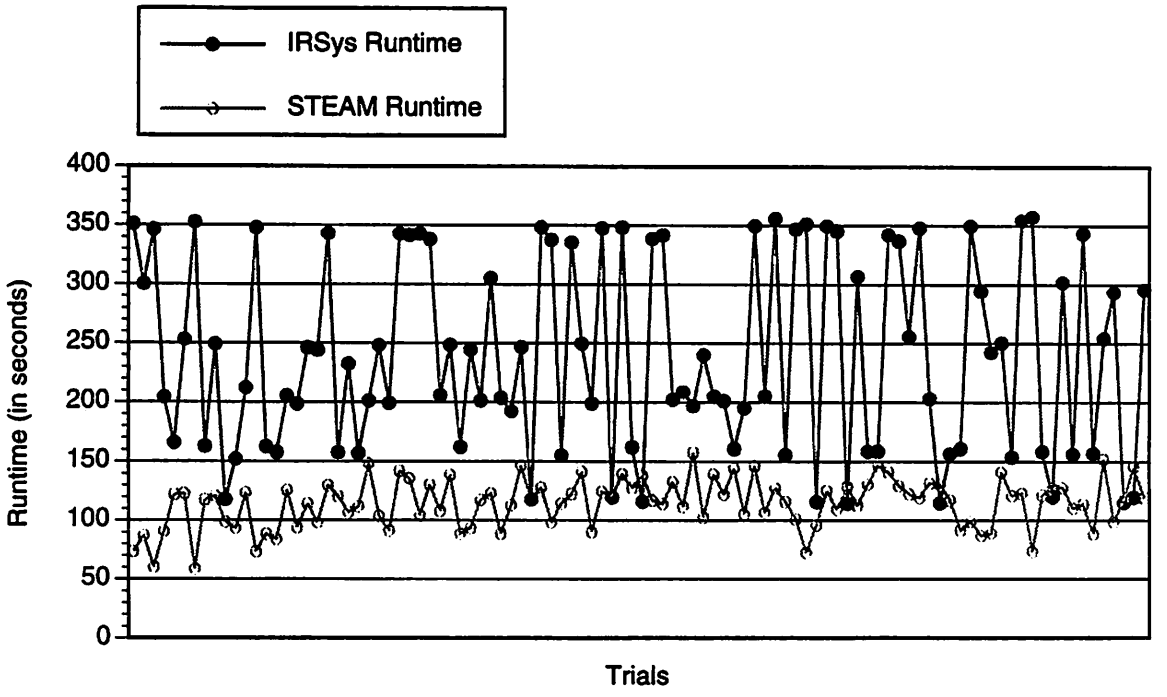


Figure 6.12. *Comparison of Runtime Required by* IRSys *and* STEAM

directly relies on constraining information in the problem specification examine that information immediately which is not done in IRSys.

However, although STEAM produced fewer component proposals than IRSys, it did take longer in real time to recognize the failure. This happens because STEAM agents spend some time at the beginning of each run setting up dynamic constraints. In other words, at the beginning of each run, STEAM agents preprocess the problem specification to determine if there are boundary constraints that can be determined from the information in the problem specification. If so, these constraints are made explicit and stored in the agent's database. As will be seen in Chapter 8, the preprocessing done by STEAM agents actually reduces the amount of time it takes to find a solution in most cases. However, it also introduces extra overhead at the beginning of each run, which is reflected in the amount of time required to recognize infeasible problem specifications.

There are many factors contributing to the performance differences in the two systems. It is difficult to compare results directly because the systems have such different approaches to controlling the search process. However, some of the major STEAM features that we believe lead to the improvement in design cost are:

- The initiation of solutions by multiple agents. This allows the system to look at a wider variety of starting points for potential solutions, rather than focusing on points that are promising for a single subsystem.

- The direct assimilation of conflict information from other agents. An agent is able to direct its local search to areas of the solution space that are likely to lead to mutually acceptable solutions based on a more global perspective.
- The search strategy. STEAM examines more solutions at a shallow level and only goes deeply down paths that seem relatively promising.

Basically, the extra work that has been done to build a flexible framework for supporting agent coordination and interaction in STEAM pays off in better designs. STEAM does not have an advantage in domain expertise since the two systems share exactly the same domain knowledge. However, it uses that domain knowledge in a more opportunistic and flexible way to respond to specific situations.

6.5 Summary

In this chapter, we examined the first of the two application systems developed in the TEAM framework: STEAM, a seven-agent system for parametric design of steam condensers. We began the chapter with a discussion of why the design domain *is* appropriate for TEAM problem solving. We described three attributes of design systems that are advantageous for our model: knowledge heterogeneity, knowledge modularity, and the potential for concurrent agent execution. We also discussed disadvantages of the model, especially the need for sophisticated integration machinery that adds to the development burden for agents and to the overhead costs of solving problems. These disadvantages appear to be more serious for simple applications where the extra overhead and development costs outweigh the benefits. However, as problem domains become more complex, the flexibility, modularity, and concurrency of our approach become cost-effective and performance-effective.

The second section in this chapter describes specifics of the steam condenser design domain including the task decomposition and organizational structure used for STEAM. Next, we discuss the mapping from TEAM to STEAM describing how features of the TEAM framework are manifested in STEAM, specifically how the generic TEAM language is extended in STEAM, how the negotiated-search strategy is embedded in the system, and the specific acceptability and termination policies used by the system.

STEAM domain knowledge was extracted from an iterative respecification system which we call IRSys. The two systems have identical domain knowledge but different architectures and search strategies. We compared the results of running the two systems on thirty identical problem specifications to evaluate their relative performance. We found an average 12.81 percent improvement in solution quality (as measured in terms of minimum design cost) in the STEAM system. We also found an average

58.82 percent reduction in the number of component proposals generated by STEAM as compared to IRSys and a corresponding average 52.32 percent reduction in runtime.

Although we have not done a focused investigation of how STEAM achieves its improvements in solution quality and performance, we hypothesize some system characteristics that may contribute to the performance differential. In STEAM multiple agents initiate solutions leading to broad and thorough coverage of the composite solution space whereas in IRSys solutions are initiated by a single agent. STEAM agents directly assimilate and utilize conflict information from other agents, whereas IRSys agents interact only through managers that use conflict information to write problem specifications for their subcontractors. In Chapter 8, experiments are done in which the STEAM system is run first with assimilation enabled and then with it disabled and the results are compared. As will be seen in Chapter 8, assimilation improves solution quality by approximately 5.7 percent. Therefore, we can attribute that percentage of the quality improvement directly to assimilation. Finally, IRSys uses a sequential backtracking search strategy, while STEAM uses the negotiated-search strategy which appears to be very well-suited to this type of search.

In summary, in this chapter, we motivated and described the STEAM system. We discussed how the the system incorporates the TEAM framework and implements negotiated search. STEAM was directly compared to IRSys and its relative performance was found to be quite impressive. We discussed factors that may be contributing to this performance. The implementation of STEAM proves that the TEAM model is a viable one. It further illustrates that the TEAM framework is workable and lends credence to the claim that this style of problem solving is both feasible and practical when applied to design problems.

STEAM is a globally cooperative system: the quality of a solution is evaluated globally. In the following chapter, we introduce AGREE, a locally cooperative system for price negotiation. We also look at several more restricted but powerful distributed-search strategies (in contrast to negotiated search) and examine the conditions under which these strategies are opportune.

CHAPTER 7

CUSTOMIZING DISTRIBUTED-SEARCH STRATEGIES

In this chapter, we will describe several *distributed-search strategies* that are very effective in certain limited situations. The more general negotiated-search algorithm described in Chapter 5 can be thought of as a default strategy that does not rely on specific agent characteristics or specific inter-agent relationships. Although it is very flexible and can be applied to a large number of problem-solving situations, there are many cases where better, more efficient, methods exist for deriving solutions.

An effective strategy can reduce the computational cost of finding a solution by reducing the number of local searches, the complexity of local search, or the number of required interactions among agents, or by increasing the precision of the search. Distributed-search strategies specify search operators and a protocol for applying the operators to solutions as in the negotiated-search strategy described earlier. A customized strategy also specifies a set of conditions that must exist at the local agent level or across the set of agents. In the following sections we describe:

- the **AGREE** system
- an implemented distributed-search strategy, *linear compromise*
- **TEAM** mechanisms which enable strategy selection and role assignment
- an empirical comparison of *negotiated search* and *linear compromise*
- two unimplemented distributed-search strategies, *bin1* and *bin2*

7.1 The AGREE *System*

The **AGREE** system supports simple two-agent price negotiations. Agents try to reach consensus on the price of some unspecified artifact with one agent acting as a buyer and the other as a seller. The agents are locally cooperative, meaning that there is no global evaluation of solutions. A contract is acceptable as soon as both agents agree to the specified price. It is always possible that agreement will not be reached.

The value of the artifact is simulated by the system as follows: a base fair market value for the artifact is randomly generated. Then, each agent generates its own

perceived market value by randomly choosing a value within a specified percentage of the base, modeling a system in which market value is determined subjectively and the agents involved in a transaction may not agree. Each agent then generates either the maximum or minimum price it would accept, in effect, it's 'bottom-line' price. The buyer determines the maximum price it would pay and the seller determines the minimum price it would accept. These values are selected by randomly generating a value within a specified percentage of the locally perceived market value. The local utility of an agent for a particular contract price is based on a normalization of the difference between the contract price and the bottom-line price for that agent. The system halts when both agents agree to accept a contract or when they determine that no agreement will be reached.

7.2 Linear Compromise

The *linear-compromise* strategy can be applied when two agents have intersecting linear functions over some variable that describe the utilities of their solutions (an example is shown in Figure 7.1). We demonstrate the effectiveness of linear compromise in the AGREE system. In this implementation, the local utility of a solution is calculated by applying a local linear objective function to a price specified in a contract between the buyer and seller. In Figure 7.1, we illustrate intersecting linear utility functions over the value of contract price. Note that the intersection point of the two

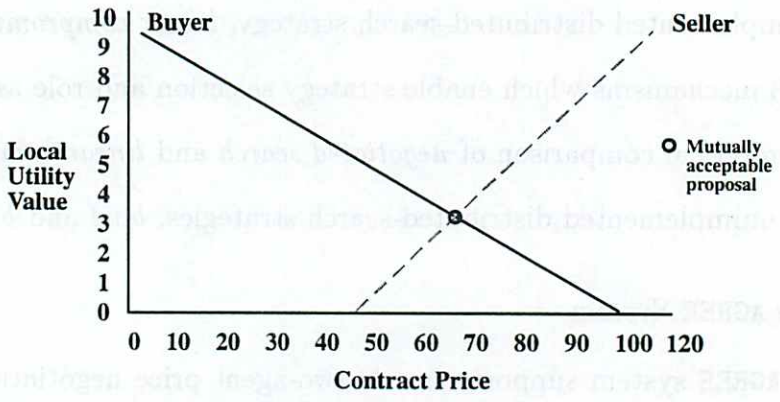


Figure 7.1. *Intersecting Linear Functions*

functions can be algebraically calculated. The intersection point is the "fairest" value that can be assigned for price since the utility of that solution is equal for each of the agents. The intersection point can be calculated by an agent in one of two ways: 1) the other agent can communicate information about its utility function; or 2) the required information can be extrapolated from points proposed by the other agent,

if it is known that those points represent a linear function. Either or both of these methods could be implemented if desired. In implementing the linear-compromise strategy in **AGREE**, we chose to use the second method.

7.2.1 Implementing Linear Compromise

In the linear-compromise strategy, as implemented in **AGREE**, there are two roles: *point-generator* and *solution-calculator*. Each of these roles must be assigned to one of the agents. The point-generator must provide two points along the path defined by its utility function and the solution-calculator uses those points to extrapolate the function and calculate the intersection point. It may be that either agent could play either role or one of the agents may only be able to play one of the roles. The requirement for successful application of this strategy is that at least one agent must be capable of taking responsibility for each of the roles and that a different agent takes each role. Each role has only one required operator as shown in the transition-network view of linear compromise in Figure 7.2.

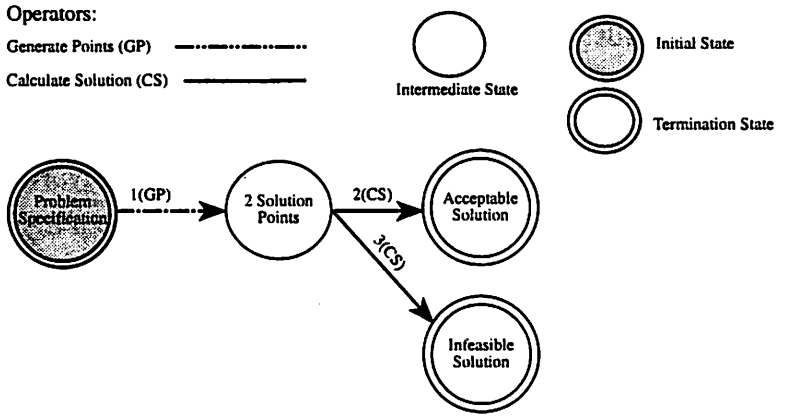


Figure 7.2. *The Transition Network for Linear Compromise*

The *generate-points* operator implements the following functionality: generate two points in the local solution space and place those points in in shared memory. No further restriction is placed on the operator. An agent may provide previously existing points or it can generate new points depending on how and when the strategy is invoked. Different agents may implement the operator with different search mechanisms for generating points. Similarly, the *calculate-solution* operator is defined simply to retrieve two points from the other agent in shared memory, and to algebraically determine the intersection of the two linear functions. This intersection point is accepted by both agents because it represents the fairest compromise possible. Before

agreeing to apply this strategy, an agent must be willing to abide by the resulting contract.

7.3 Choosing a Strategy to Apply

When the linear-compromise strategy is selected for application and roles have been assigned to agents, the strategy is instantiated and executed, and the resulting compromise is installed as the mutually acceptable solution. This strategy is extremely effective when it can be applied because it uses an algebraic method for computing the solution, the solution is optimal in the sense of fairness, and the strategy does not require any iterative exchange of information or progressive relaxation. However, the restrictions on the use of the strategy are quite severe and, therefore, it can only be applied in limited situations. In this section, we describe how a set of agents can evaluate the problem-solving situation and determine if any known customized distributed-search strategies are applicable. In order to keep the presentation focused and concrete, we discuss this in the context of **AGREE**. However, the strategy selection mechanisms are domain-independent and can also be applied to other systems built within the **TEAM** framework.

Because the agents are heterogeneous and cannot make assumptions about global solution-space characteristics, agent capabilities, local-search algorithms, or the evaluation criteria of other agents, there are two distinct phases involved in recognizing that an appropriate distributed-search strategy exists. First each agent must individually determine if it knows of any strategies that would fit its local situation, and second, some agent (or subset of the agent set) must determine if any of the locally available strategies fit the inter-agent situation. Consider the linear-compromise strategy described above: first, each agent must recognize that it has a linear function that describes utility value for a solution over some variable. Then an agent (or agents) must recognize that all relevant agents have indicated that this strategy is available and that every required role for the strategy can be assigned to some agent. Once this is established, each agent must be informed that a strategy has been chosen and must be told what its responsibilities will be.

Strategy selection can occur at different times during problem solving. First, it may be possible that a strategy can be selected at the beginning of a problem that is appropriate for solving the whole problem. This is the case that will be described in the discussion of linear compromise in the **AGREE** system. The system has a simple evaluation policy for solutions, and therefore, can use a simple strategy for solving the complete problem. An initial round of strategy selection therefore occurs at the beginning of a problem-solving task.

A second case, that is not currently implemented in **TEAM**, is that specific conflict-resolution strategies can be selected to address conflicts that occur dynamically during problem solving. This dynamic selection capability is described in Chapter 5 and a transition-network view of an extended version of negotiated search that incorporates it is shown in Figure 5.10. In this extended strategy, the detection of a conflict by an agent may lead to selection of a specific type of conflict-resolution strategy that is intended to resolve only that conflict rather than to solve the complete problem. Conflict detection occurs when an agent is evaluating a solution (in **AGREE**, a contract) initiated by another agent. The evaluating agent finds some aspect of the solution (e.g., contract price) to be unacceptable and sends a conflict message to the initiating agent. It also attaches a `conflict` object to the contract being evaluated. A generic domain-independent version of conflict object was shown in Figure 4.7. For the extended strategy, the conflict object is also extended as shown in Figure 7.3.

```
Conflict
  Originator              % The agent that detected the conflict
  Violated Requirements   % Requirements that are in direct conflict
                          %   with the proposal being evaluated.
  Parameter Name/Value List
                          % A list of parameters and their
                          %   assigned values that are in conflict
                          %   with the original composite solution
  Proposal                % A link to the proposal that triggered
                          %   the conflict
  Solution                % A link to the composite solution
                          %   associated with this conflict
  Suggested Resolution    % A compromise that is acceptable to this
                          %   agent (optional).
  Instantiated Strategies
                          % A list of instantiated-strategy objects
                          %   that is initially empty
  Suggested Strategies    % A list of locally known resolution
                          %   strategies that appear to be
                          %   applicable from the detecting agent's
                          %   viewpoint
```

Figure 7.3. *A Conflict Object*

The attributes of `instantiated-strategies` and `suggested-strategies` are added to the generic conflict object described in Chapter 4 specifically to support

embedded conflict-resolution strategy selection. For example, if an agent has a linear utility function over price, locally represents the linear-compromise strategy, and can play at least one of the roles required for linear compromise, it will suggest the linear-compromise strategy by placing a **suggested-strategy** object in the **suggested-strategies** slot of the conflict object. If more than one strategy is suggested in this way, the list of strategies is ordered by local preference. A generic suggested-strategy object is shown in Figure 4.9. This generic object is customized for embedded conflict-resolution strategies as shown in Figure 7.4. For linear compromise,

```
Suggested Strategy Object
 Suggested_Strategy
   Originating_agent    % The agent suggesting the strategy
   Strategy_name        % The name of the strategy
                        %   (e.g., linear compromise)
   Roles                % The roles that the agent can play
                        %   for this particular strategy
                        %   (e.g., point-generator)
   Conflict             % A link to the conflict object that
                        %   triggers this suggestion
   Instantiation        % A link to an instantiated-strategy object
                        %   (initially empty)
```

Figure 7.4. *A Suggested-Strategy Object for Embedded Conflict Resolution Strategies*

the possible roles are {*point-generator,solution-calculator*} and, if the agent can play more than one role, the list is ordered by local preference.

The agent that initiated the contract becomes the conflict manager for this conflict when it is notified of the conflict detection by another agent. It must choose a strategy to resolve the conflict and assign the operators associated with that strategy to some subset of participating agents. In the AGREE system, the conflict manager does not need any information other than a conflict object to select a strategy and assign roles because there are only two agents involved. If there are more than two agents, the conflict manager must collect information from other agents about what strategies and roles they have available. This can be accomplished by sending the conflict object to agents participating in the contract and requesting that they return suggested strategies for conflict resolution.

When the contract manager has collected the required suggested-strategy information from all participating agents, it must choose a strategy to instantiate. It

takes the intersection of all the suggested strategy lists as an initial set of potential strategies and then checks whether or not any of these is complete with respect to role assignments. Potential strategies are tried in the order preferred by the contract manager. To instantiate a strategy, the contract manager must find a set of role assignments in which every role can be assigned to an appropriate agent. If the contract manager does find a complete set of role assignments, it generates an `instantiated-strategy` object and sends a message to each agent that specifies the selected strategy and the role that should be active at that agent. (A generic instantiated-strategy object is shown in Figure 4.10). If no strategy or no consistent set of role assignments is found, the agents fall back to the default distributed-search strategy.

When an agent receives a strategy assignment message, it attempts to execute its assignment by activating the appropriate operator or operators required for the role it is to play. In the case of a strategy that is invoked specifically to resolve a particular conflict, the operators are active only with respect to that particular solution. Currently, an agent must accept a suggested resolution strategy if it is capable of doing so whether or not that strategy is one that it prefers. In the current TEAM implementation, the contract manager has control of strategy selection and role assignment and will always give top priority to its own preferences. Note, however, that negotiating over possible strategies and role assignments can itself be treated as a conflict situation with the same type of iterative search/relaxation methodology that applies to domain problem-solving. We have adopted a fixed protocol for resolving these conflicts rather than allowing the full range of conflict-resolution activities. However, this has been and remains an open research issue [Davis and Smith, 1983, Laasri *et al.*, 1992, Sen and Durfee, 1992]. There is always a tradeoff between the generality of an approach and the cost of that approach. We have chosen to limit generality and flexibility in exchange for limiting the amount of search required to find a suitable strategy.

There are three possible outcomes to the execution of a strategy: 1) a solution is successfully generated; 2) some agent participating in the strategy recognizes that no solution exists within the current problem specification; or 3) some agent participating in the strategy discovers some exceptional condition that cannot be handled. When an agent recognizes either Case 1 or Case 2 above, it notifies the TEAM framework manager that termination conditions have been met, and that the system should halt. However, Cases 2 and 3 require some further explanation. Consider the following scenario: two agents have linear utility functions over a variable as described in the discussion of linear compromise. After following the selection protocol described above, the *linear*

compromise strategy is instantiated. Figure 7.5 illustrates the potential cases that are

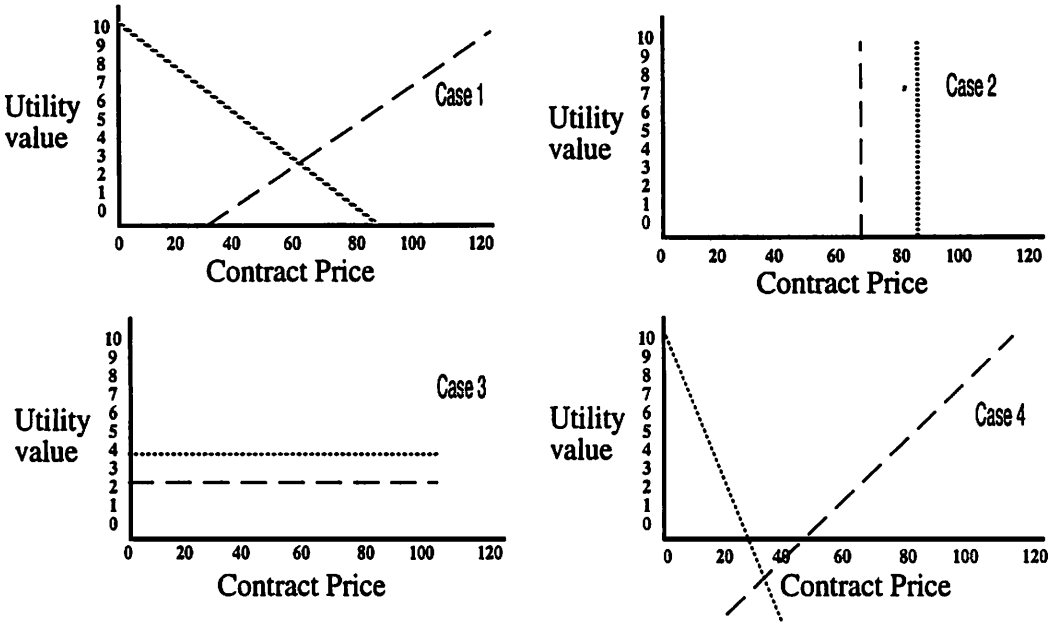


Figure 7.5. *Composite Spaces for Linear Compromise*

encountered when the strategy is executed. In Case 1, the intersection point of the two lines is returned as a solution. In Cases 2 and 3, the graphs of contract-price/utility value show parallel non-intersecting lines. In Case 4, the lines intersect outside of the defined range of one or more of the functions, i.e., the intersection occurs at a utility value that is not within the bounded set of values for which an acceptable solution is defined. The composite space of the functions, as shown in the figure, cannot be determined until the strategy is executed and no agent can predict which case will occur. This is a simple example of a complex phenomenon: salient characteristics of the composite solution space often cannot be determined locally. Therefore, it is possible that a strategy that seems feasible from the local perspectives of all involved agents is actually infeasible over the composite space.

When a strategy fails, the strategy selection process can begin anew by notifying all agents and returning control to the conflict manager. The conflict manager can remove the strategy from the list of potentially applicable strategies and reselect. However, it must be determined whether the strategy failed because of some deficiency in the capabilities of the strategy or because there is no solution to the problem. To illustrate, suppose that Case 2 or Case 4 in Figure 7.5 has occurred. In these cases, no solution exists within the definition of the problem. In Case 3, however, every contract price is a potential solution but, because the lines are parallel, no intersection point

can be calculated and the linear compromise strategy would fail. Another strategy could be activated however that would not fail. For example, a strategy could be executed to randomly select a contract price, resulting in both agents having positive, although unequal, utility values. Alternatively, the general negotiated-search strategy could be applied and, although it would take a little longer to find a solution, it would eventually find one. The main point is that if it is possible for a strategy to determine that no solution to the problem exists, it is appropriate to return a failure and halt the system, while in other situations, it is appropriate to try another strategy.

7.4 Comparison of Linear Compromise and Negotiated Search

In this section, we compare the effectiveness of two strategies: a modified version of the negotiated-search strategy described in Chapter 5 and the linear-compromise strategy described in Section 7.2. The **AGREE** system was run on 10 problem specifications (as shown in Table 7.1). For each problem specification the system was run once under negotiated search and once under linear compromise. The results of these trials are compared below in terms of solution quality and processing time. Although the number of trials is limited, the general conclusions can be derived analytically and therefore the experiments serve primarily to illustrate the implementational viability of the model.

In the extended version of the negotiated-search strategy used in these experiments, each agent generates proposals at the currently acceptable level of utility value for some number of cycles. If no agreement is reached, the acceptable level of utility value is dropped and the agents generate proposals at the new utility value for some number of cycles. This algorithm repeats until either a solution is found or until the acceptable level of utility value drops below some threshold for each of the agents, at which point negotiated search fails, and no contract is made.

The difference between this version of negotiated search and that presented earlier is that, in this version, new proposed solutions must be *at* the current acceptability level rather than *at or above* the current acceptability level as in the earlier version. For example, if the current acceptability level is *good*, solutions that are rated as *excellent* will not be proposed in this extended strategy. This is appropriate for a locally cooperative situation, where it is known that agents have an inverse relationship between their local evaluations, i.e., where agents are directly competing. In order to move toward resolution, agents must progressively move down their local utility gradient. We therefore adapted the negotiated-search strategy presented in Chapter 5 as applicable in globally cooperative situations, in order to make it effective in locally cooperative situations.

7.4.1 *Solution Quality under Different Strategies*

The problem specifications and resulting contract prices derived by the modified negotiated-search experiments are shown in Table 7.1. The problem specifications were generated randomly as described in Section 7.1, but were sorted into ascending order based on contract price in the tables and figures below. The agents' bottom-

Table 7.1. *Problem Specifications and Resulting Contract Prices Under Negotiated Search in the* AGREE *System*

Trial Number	Buyer's Initial Offer	Buyer's Maximum Offer	Seller's Minimum Price	Seller's Initial Price	Contract Price
1	185696	230783	141469	210964	196178
2	426658	495098	322499	493148	443309
3	405510	545477	456101	493336	474928
4	432030	536765	398897	577888	488217
5	452347	572992	426629	513957	488249
6	519855	629300	484763	666709	568264
7	458202	686769	477945	762279	588534
8	722955	755135	609051	866125	738834
9	835865	968982	752096	1097243	909152
10	284448	408309	432317	487862	no solution

line prices and the final contract price for each trial are graphically summarized in Figure 7.6. In this figure, we show the results of 9 of the 10 trials since 1 trial did not result in a mutually acceptable solution.

Although the contract price always falls somewhere between the two agents' bottom-line prices, it is not always a completely "fair" compromise because one agent's utility value for the price may not be equal to the other agent's utility value. In Table 7.2, we compare the agents' utilities for each contract found under negotiated search to demonstrate that, with that strategy, agents will not be equally happy with a solution. In contrast, *linear compromise* can be considered an optimal strategy in terms of agent fairness. We define a fairness measure that is the absolute value of the difference in the utility values calculated for the buyer and seller. The lower the value of the fairness measure, the fairer the solution: an optimal solution has a fairness measure of 0. If we evaluate solutions based on this measure, we define the ultimate in cooperation: the global evaluation of a solution is based on minimizing the difference between the agents' local evaluations of that solution.

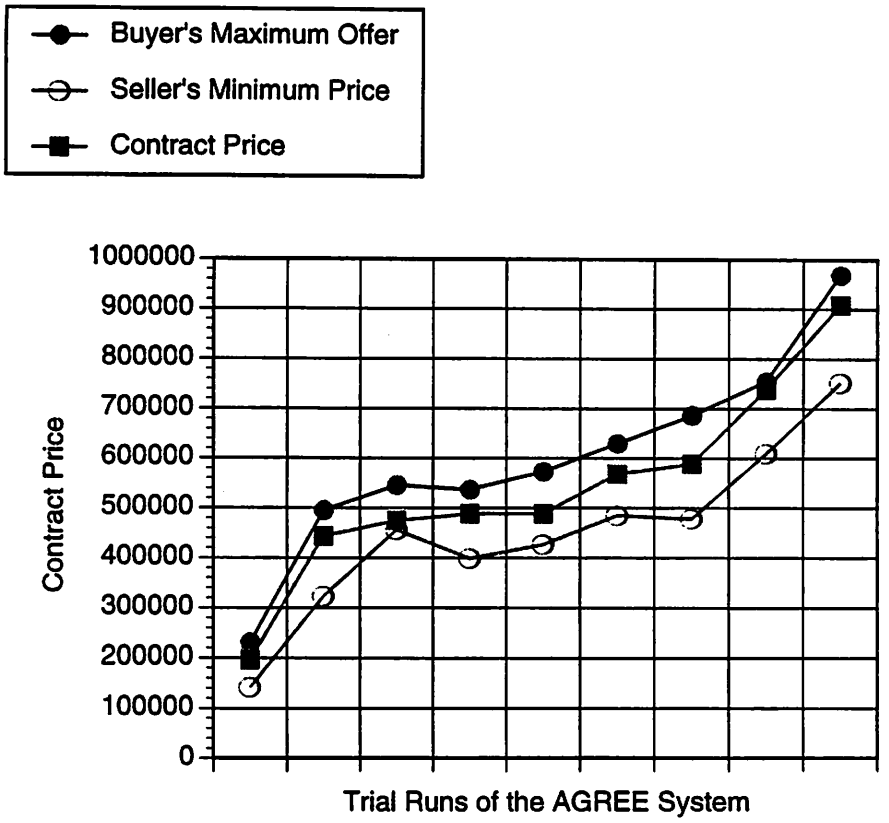


Figure 7.6. *Bottom-Line Prices of Agents Contrasted with the Agreed-Upon Contract Price in Trials of the* AGREE *System under Negotiated Search*

Table 7.3 compares the contract prices and agent utility values achieved by each strategy over the set of problem specifications. The average agent utility is shown for negotiated search since the buyer and seller have different utilities for the contract. However, because linear compromise is optimal with respect to agent fairness, the buyer and seller utilities are equal for that strategy.

The contract prices found by linear compromise are optimal if fairness is used as a measure of optimality. As can be seen in the table, the solutions found by negotiated search are suboptimal. However, in many domains, this level of suboptimality is acceptable since it is proportionally small. Ultimately, of course, it is up to the system designer to decide whether optimality is achievable and/or required and how much human and machine effort can be spent to reach it.

7.4.2 Processing Costs under Different Strategies

Above, we showed how solution quality varied between the negotiated search and linear compromise strategies. In this section, we examine the processing time required for each of the strategies. Processing time is measured in terms of agent

Table 7.2. *A Comparison of Agent Utilities for the Negotiated Contract Price in Negotiated Search*

Trials	Buyer's Utility	Seller's Utility	Fairness Measure
1	7.87	7.68	0.19
2	7.08	7.57	0.49
3	5.06	5.04	0.02
4	4.99	4.64	0.35
5	7.06	7.02	0.04
6	4.59	5.58	0.99
7	3.89	4.3	0.41
8	5.05	5.07	0.02
9	4.55	4.49	0.06

cycles as described in Chapter 4. Briefly, the TEAM framework iteratively executes two cycles, a framework cycle and an agent cycle. During an agent cycle, each agent is sequentially invoked, executes, and sends any messages regarding its work that are appropriate. During the framework cycle, the TEAM manager processes incoming messages as appropriate, updates the shared memory, and sends outgoing messages to agents. Once the TEAM manager finishes the framework cycle, a new agent cycle is entered, and so on.

The number of agent cycles required to execute a strategy is dependent on the strategy. The linear compromise strategy has an agent-cycle cost of two or three, depending on the role assignments. During the first cycle, the conflict manager chooses the strategy and assigns roles. If it assigns itself the role of point-generator, it will generate the required points and send a message to the TEAM manager to place the points in shared memory. On the second iteration, the other agent receives its role assignment. If it is the point-generator, it will generate the required points. If it is the solution-calculator, it looks for the points generated by the other agent and calculates and installs the solution, thereby terminating the strategy. If the first case occurred (the second agent was the point-generator), on the third cycle, the first agent has the role of solution calculator. It finds the points generated, calculates the intersection, and installs the computed contract price.

For linear compromise, therefore, the total agent-cycle cost of the strategy is either 2 or 3 cycles. In contrast, the agent-cycle cost of the modified negotiated-search strategy described in Section 7.1 is on the order of nr where n is the cardinality of

Table 7.3. *A Comparison of Utilities and Contract Prices under Negotiated Search and Linear Compromise*

	Linear Compromise Contract Price	Negotiated Search Contract Price	Linear Compromise Agent Utility	Negotiated Search Average Agent Utility
1	195638	196178	7.79	7.78
2	445691	443309	7.22	7.32
3	474882	474928	5.04	5.05
4	485873	488217	4.86	4.81
5	488087	488249	7.04	7.04
6	575015	568264	4.96	5.08
7	593710	588534	4.07	4.09
8	738833	738834	5.05	5.06
9	908615	909152	4.53	4.52

the set of utility values and r is the number of agent-search cycles performed before automatic relaxation occurs (see Chapter 5 for a discussion of automatic relaxation in negotiated search). Sample data collected on the agent-cycle cost of the negotiated-search algorithm for the experimental problem specifications are shown in Table 7.1. In these experiments n was set to 10 and r was set to 5. The observed number of agent cycles range from 15 to the worst case of 50, which occurs when no solution is found, and the search fails.

From these results, it is clear that linear compromise is the more effective strategy for the **AGREE** system as it is currently implemented. However, it places very stringent requirements on the agents: one would not often expect to find linear utility functions for solutions. Although developing a taxonomy of search situations and applicable strategies is beyond the scope of this dissertation, we would like to convince the reader that other strategies are both possible and useful. In the next section, we do this by describing two types of distributed binary search that can be applied when the situation is similar to that appropriate for linear compromise except that one or both of the agents have monotonic rather than linear utility functions.

7.5 Compromising over Monotonic Utility Functions

In this section, we describe two strategies, *bin1* and *bin2*, that are slightly less restrictive than the *linear compromise* strategy described above. These strategies have not been implemented, but are included here to give the reader a sense of how different strategies can be developed for different situations. Suppose that instead of the strict linear functions required for *linear compromise*, one or both agents have

non-linear, but monotonic, functions that define the utility of a contract. Figure 7.7 illustrates the case where one agent has a linear utility function and the second has an inverse, monotonic utility function.

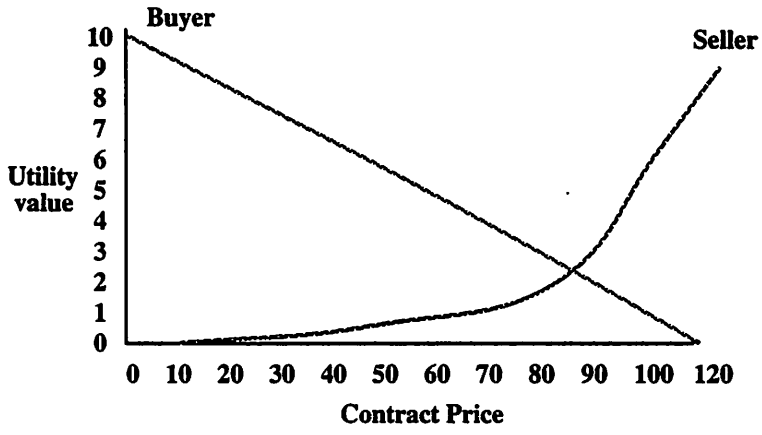


Figure 7.7. *A Situation Suitable for Monotonic Compromise*

In this case, a possible strategy for selecting a contract price would be to apply a distributed binary search to the composite search space. There are two ways in which a distributed binary search could be implemented. The first method (*bin1*) is a binary search over possible solution values from the highest value of the lowest-acceptable-value of any agent to the lowest value of the highest-acceptable-value of any agent. For example, in Figure 7.7, the binary search boundaries would be from a contract price of \$10 to a contract price of \$120. One agent can instantiate an *initiate-solution* operator and the other agent would then instantiate an *evaluate-solution* operator. Note that in TEAM, both agents could actually instantiate both operators and evaluate each other's proposals. The agent that initiates a solution would rate the solution locally, and the evaluating agent would then rate the same solution from its perspective. If the two ratings were equal, the solution would be accepted and the system would halt. If the evaluations were not equal, the initiating agent would determine the next solution value to try using the distributed binary-search algorithm.

The basic idea underlying the algorithm is similar to that in centralized binary search: 1) take the middle value of an interval and determine a key for that value (in this case, the utility ratings for that value). If the other agent's rating is lower than your own, make the current value either the upper or lower boundary of the interval and repeat the process. The decision about whether to make the current value the upper or lower boundary is determined by the direction of your utility function. For example, if your utility increases as the value increases, and your current utility is

higher than the other agent's, the current value should become the upper boundary of the interval.

In the second distributed binary-search algorithm (*bin2*), search occurs on the utility value rather than the solution value. The contract-initiating agent would generate a solution that has a specified local utility value and the second agent would evaluate that solution. If the evaluations were not equal, the initiating agent would determine the next utility value to try and generate a solution at that level. Again, if the evaluations were equal, the solution would be accepted and the system would halt. To use this algorithm, an agent would take the middle value of the utility range (from 0 to 10 in Figure 7.7). It would then generate a solution that had that utility rating. If the other agent's evaluation of the generated solution was lower than the target rating, the target rating would become the new upper boundary and the process repeats. Again, if the two ratings are equal, the solution is accepted, and the system halts.

Table 7.4 contrasts the two binary searches described above. This data was derived by applying *bin1* and *bin2* by hand to a problem in which utility values are integers from 1 to 10 and possible solutions are integers from 10 to 120 as shown in Figure 7.7.

Table 7.4. *Comparison of the BIN1 and BIN2 Strategies*

	Bin1			*Bin2*		
Cycle	Proposed Solution	Utility1	Utility2	Proposed Utility	Proposed Solution	Respondent's Utility
1	65	5	1	5	65	1
2	93	2	4	3	86	2
3	79	4	2	2	93	4
4	86	3	2	3	90	3
5	90	3	3			

For the monotonic compromise strategies described above, agent-cycle cost is on the order of *log n* where n is either the cardinality of the set of possible solution (*bin1*) or the cardinality of the set of possible utility values (*bin2*). Notice that *bin2* generally results in fewer agent cycles than *bin1* because the cardinality of the set of utility values is normally less than that of solution values (see Table 7.4). However, there is a trade-off between the number of agent cycles and the amount of local computation. In order to generate a solution at a particular utility value level (as in

bin2), it may be necessary to generate multiple local solutions. For example, local search may be accomplished through a random generate-and-test paradigm, in which case the number of solutions generated before one is found with the correct utility level follows a geometric probability distribution based on the cardinality of the set of utility values. So, for every agent cycle, multiple local solutions are likely to be generated. In contrast, with *bin1* only one solution is generated for each agent cycle but more cycles are required. The most effective method (*bin1* or *bin2*) depends on factors that include the cost of communication vs. the cost of local solution generation, and the cost of an agent cycle vs. the cost of a local cycle.

7.6 Summary of Results

In the experiments described in this chapter, we wanted to demonstrate that the use of highly focused search strategies in place of more general strategies would improve system performance and solution quality. We devised and implemented a simple but effective strategy, *linear compromise*, that has stringent precondition requirements on its applicability. We then applied the strategy within the AGREE system in a problem situation where the precondition requirements were fulfilled. The system was run on 10 problem specifications as shown in Table 7.1. For each problem specification, the system was run once under the linear compromise strategy and once under the more general negotiated-search strategy.

The observed results from these experiments substantiate the claim that using customized distributed-search strategies when appropriate will improve system performance in terms of solution quality and processing time. In these experiments, we measured the quality of a solution in terms of its *fairness*. That is, an optimal solution is one in which the utility values of the agents are equal. Under this definition, the linear compromise strategy uses an algebraic method to produce optimal solutions, while the negotiated-search strategy does not. We also demonstrated that linear compromise takes substantially fewer processing cycles to derive a solution.

Obviously linear compromise is the strategy of choice when there is a choice. However, the applicability of linear compromise is so severely restricted that it is seldom an option, whereas negotiated search is widely applicable. By comparing these two extremes, we made clear the potential for developing effective search and conflict-resolution methods while also cautioning that there are difficult control issues to be addressed in dynamically selecting appropriate methods and integrating these methods into search systems. We envision the evolutionary development of CR strategies and protocols such as those proposed by Klein[Klein, 1991], Khedro[Khedro and Genesereth, 1993], and Zlotkin[Zlotkin and Rosenschein, 1990] that have useful

properties of completeness and convergence and the incorporation of those strategies into systems as options to be selected when the appropriate problem characteristics exist. Here, we have demonstrated that the TEAM framework provides a functional foundation for integrating these strategies into comprehensive cooperative search among heterogeneous reusable agents.

In summary, there are three important points to extract from this chapter:

1. Many different distributed-search strategies are possible, ranging from very general search to highly focused forms of conflict-resolution that can only be applied in limited situations.

2. Having the ability to select a distributed-search strategy that complements the problem-solving situation leads to better solution quality and processing-time performance.

3. The TEAM framework provides the capabilities required to realize dynamic selection of distributed-search strategies.

Up to this point, the dissertation has discussed general principles in distributed search and conflict resolution. We described the framework we built to support reusable- and heterogeneous-agent search. In the last two chapters, we described the application systems built within the TEAM framework and began to explore some of the system performance questions that arise in those systems. The rest of the dissertation is primarily concerned with empirical evaluation of various framework capabilities. In the next chapter, we discuss the information exchange and assimilation capabilities of the TEAM framework and describe experiments designed to evaluate the effect of those capabilities. In Chapter 9, we empirically investigate whether different role assignments (where a role defines the set of operators that will be active at an agent) affect system performance. Finally, Chapter 10 presents a summary of the dissertation.

CHAPTER 8

INFORMATION EXCHANGE AND ASSIMILATION

When motivating the need for agents to exchange information about their search spaces in Chapter 5, we stated that "sharing information that will specifically help another agent avoid future conflicts is generally cost effective since it eliminates the expense of generating unproductive solution paths". In this chapter, we empirically demonstrate this cost effectiveness. For convenience, we will use the terms *solution requirement* and *constraint* interchangeably throughout this chapter although we do not intend for the word "constraint" to imply a specific representational format.

The experiments described in this chapter were run in the **STEAM** system with seven active agents. In order for an agent to use information received from an external source to guide its local processing (i.e., *learn* about other agents' requirements for solutions), the agent must instantiate the *store-received-information* and *retrieve-information* operators described in our discussion of the *negotiated-search* strategy in Chapter 5. *Store-received-information* basically takes constraining information sent from other agents, translates that information into a locally usable form, and stores the translated information into a local knowledge base, indexed by its source. We call this process *information assimilation.* The assimilated information can then be retrieved by an agent when it is looking for knowledge that will constrain its local search. *Retrieve-information* extends or replaces an agent's default capability to retrieve relevant constraining information from its knowledge base. The goal of the retrieval is to find the most restrictive, but non-conflicting, set of known solution requirements for the current problem using both local and assimilated information.

The information shared in these experiments was limited to simple boundary constraints. These were single-clause constraints of the form $(x < n)$, $(x <= n)$, $(x > n)$, or $(x >= n)$, where x is a shared numeric parameter and n is some numeric value. For example, pump-agent specifies a fixed feasibility constraint (*water-flow-rate* <= 404.34). We will discuss other types of information that could potentially be sent in Section 8.2.1.

We will describe two sets of experiments in this chapter. In the first set of experiments, solution quality and system performance are compared when the agents in the **STEAM** system assimilate information and when they do not. In the second

set of experiments, we categorize the costs associated with information sharing and measure how much time is spent in each category. The chapter then concludes with a summary of the observed results and some speculation as to the significance of these results within the STEAM system and within the more general realm of multiagent systems.

8.1 *Analyzing the Effect of Information Assimilation in* STEAM

In the experiments reported below, we have two categories of experimental trials: *non-assimilation trials* and *assimilation trials.* In the non-assimilation trials, the information-assimilation operators, *store-received-information* and *retrieve-information*, are not active at any agent. Conflicts are detected and constraining information is transmitted identically in both trial categories, but in the non-assimilation category, the transmitted information is not accepted by any receiving agent. For the assimilation trials reported in this chapter, three agents instantiate the assimilation operators: the pump, motor, and heat-exchanger agents.[1] In Chapter 9, we describe experiments in which the instantiation of operators at agents is varied throughout the trials to investigate correlations between agent characteristics and operator effectiveness. However, in these experiments, we are only concerned with the effects of assimilation on system performance.

In the experiments described below, the system was run on each of 100 different feasible problem specifications as described in Appendix B. Two trials were run for each specification: one with the assimilation operators enabled (the *assimilation trial*) and one with these operators disabled (the *non-assimilation trial*).

8.1.1 *Measuring System Performance*

We expected to see that the extra costs associated with assimilating information would be balanced, in the majority of cases, by improvements in performance. To test this hypothesis, we first had to determine how to measure performance.

There are two measures of system performance in the STEAM system: run time and solution quality. In the STEAM domain, solution quality is more important—we don't mind spending extra time in the design process (within reason) to get a high-quality design. In other domains, however, this tradeoff is not always the best one. It is left to the discretion of the system integrator to determine the desired tradeoff between solution quality and processing time.

[1]The selection of these agents is explained in Section 8.3.

The TEAM framework provides some easy ways to adjust the quality/time tradeoff to fit the situation. The amount of effort put into local search is currently determined by individual agents and must be adjusted through changes to the agents. However, TEAM allows the user to change the termination policy of the system through a simple adjustment of system parameters. For example, the default termination policy of the system, as described earlier, is to run until three acceptable solutions have been completed, and then to change to a problem-solving stage in which all partial acceptable solutions are completed. The user could, however, 1) choose to switch stages after a different number of acceptable solutions are completed (e.g., switch after only one acceptable solution), or 2) choose to halt immediately after x number of acceptable solutions are completed without completing outstanding partial solutions. Either of these options will tend to improve (lower) processing time at the expense of solution quality.

In these experiments, we are only concerned with the relative performance of the system in assimilation and non-assimilation trials. We fix the strategy, and the termination and acceptability policies across the trials and directly compare results. The negotiated-search strategy, as described in Chapter 5, was used for all of the runs. The termination policy used was to process until three acceptable solutions were completed. After that time, no new solutions were initiated, but existing acceptable partial solutions were completed before halting. Solutions were considered acceptable if they were locally acceptable to all agents and if the solution cost was below a user-defined threshold.

8.1.2 Solution Quality

We compared the results of running the system when agents assimilated constraining information and when they did not. In STEAM, solution quality is determined by the monetary cost of each solution: the minimum-cost acceptable design is considered the most highly rated. The framework allows operators to be easily enabled and disabled at agents, so each experiment is run first with the information-assimilation operators enabled and then with them disabled. The results of these experiments are tabulated in Appendix F and are graphically summarized in Figure 8.1. In this figure, the results are sorted into ascending order based on cost in the assimilation trial.

For the 100 problem specifications tested, the mean cost in the assimilation trials was \$8504.77, in the non-assimilation trials, it was \$9020.43. The mean cost improvement with assimilation operators enabled was 5.72%, meaning that the monetary cost of the most highly rated solution in an assimilation trial was 5.72% lower

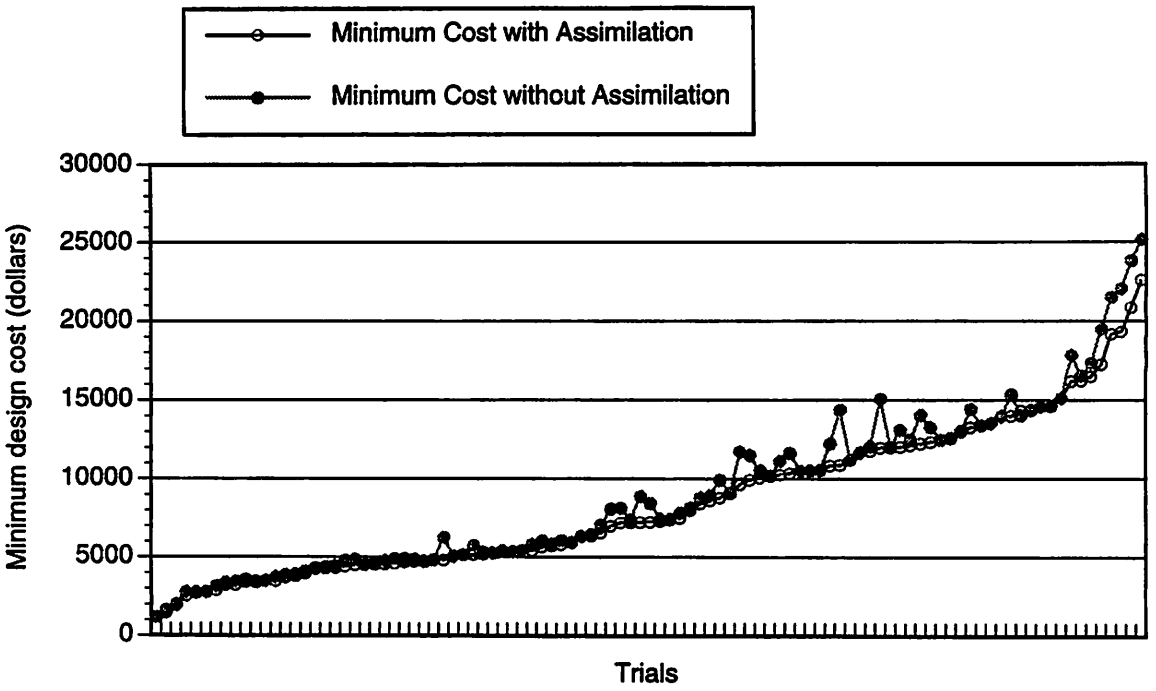


Figure 8.1. *Solution Quality Results in Assimilation Experiments*

on average than that in the associated non-assimilation trial under the identical problem specification. We had hypothesized that enabling assimilation would lower the cost of a design (thereby improving solution quality) and the experimental results appeared to support this hypothesis. To statistically confirm this result, we applied a paired difference test. In this type of test, the results from two matched trials are compared—in our case non-assimilation trials are compared to assimilation trials performed under the same problem specification. For each paired trial, the difference between the resulting design cost is taken, and then the mean of the differences is computed. The null hypothesis in this case is $H_o : \mu_D = 0$ (the population mean of the differences is 0), meaning that the results of the two trials are not significantly different. The alternative hypothesis is $H_a : \mu_D > 0$ (the population mean of the non-assimilation trial results minus the assimilation trial results is greater than 0), meaning that the cost of designs in the non-assimilation trials are higher than those in the assimilation trials. The test statistic used is

$$t_D = \frac{\bar{x}_D}{s_D/\sqrt{n_D}}$$

where $\bar{x}_D$ is the mean of the sample differences, s_D is the standard deviation of the sample differences, and n_D is the number of sample differences (pairs). Applying the test statistic results in a t-score of 6.455, which allows us to reject the null hypothesis with a confidence of more than 99%. We can thus say with a high level of confidence

encouraging because not only is improvement shown with assimilation, but the initial evidence suggests that the improvement aggregates over the set of solutions produced.

8.1.3 Run Time

Run time was directly measured in these experiments as the elapsed real time from the invocation of the system until termination of the system.[2] Run times recorded for the trials are shown in Tables F.5 and F.6 in Appendix F and graphed in Figure 8.3. In this figure, the results are sorted in ascending order of the runtimes with assimilation. The average runtime with assimilation is 121.98 seconds, without

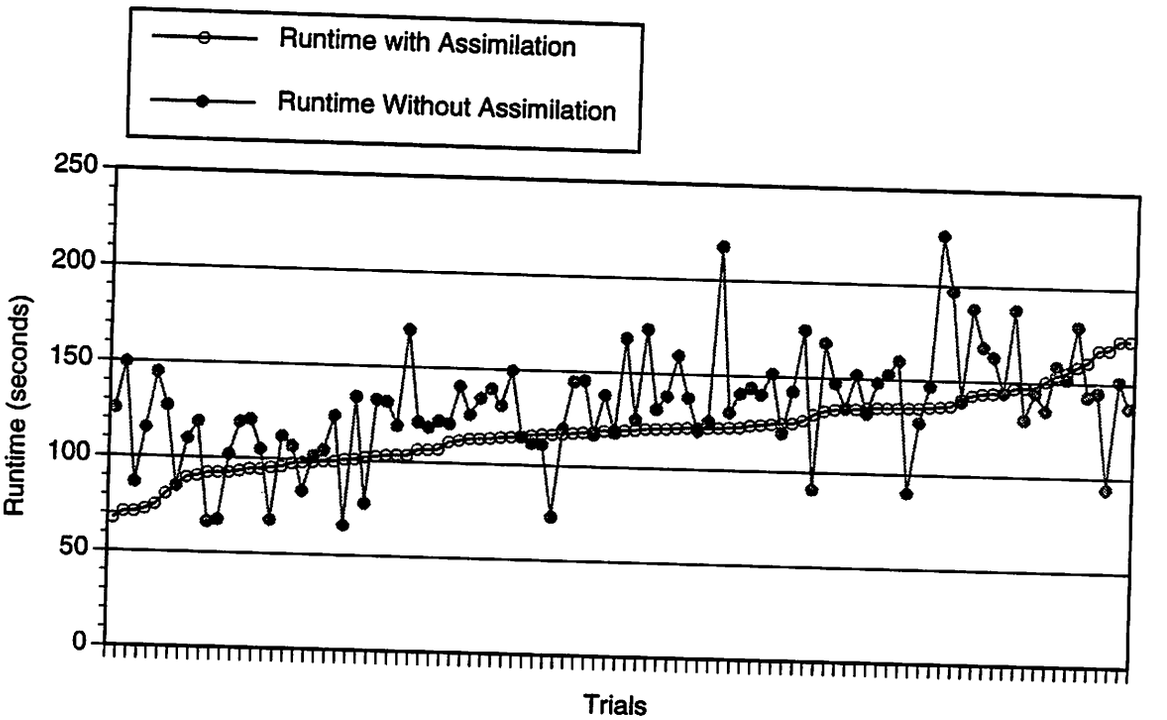


Figure 8.3. *Run Time Results in Assimilation Experiments*

assimilation the average runtime is 132.67 seconds. The assimilation runtimes are, on average, 8.06% lower than the non-assimilation runtimes. However, direct comparison of the run times of assimilation and non-assimilation trials is somewhat misleading. As noted during the discussion of negotiated search in Chapter 5, the termination policy used for negotiated search is that when x acceptable solutions are found, the

[2]These experiments were run on a TI Explorer. Incremental garbage collection was turned off during the runs. However, the recorded time includes time spent on process and memory management tasks such as paging, etc. Therefore, recorded times varied slightly across identical runs.

that when STEAM agents use assimilation operators, the average quality of solutions improves.

An inherent characteristic of the STEAM domain is that good solutions are easy to find under many problem specifications (the solution space is dense). We believe that there is a significant *floor effect* in the domain, meaning that minimum-cost designs are easy enough to find even in the non-assimilation trials that it is difficult to dramatically improve solution quality. However, the ability to consistently lower design costs approximately 5.72% by sharing simple boundary constraints is compelling evidence that information sharing and assimilation is an important technique for improving solution quality in multi-agent systems. Furthermore, if it is the case that a floor effect is influencing our results, larger improvements could be expected in some domains.

Another way to measure solution quality is to look at the average cost of alternative acceptable solutions rather than simply the cost of the single most highly rated solution. The complete results of averaging the three best solutions found in each run are listed in Appendix F and are summarized in Figure 8.2.

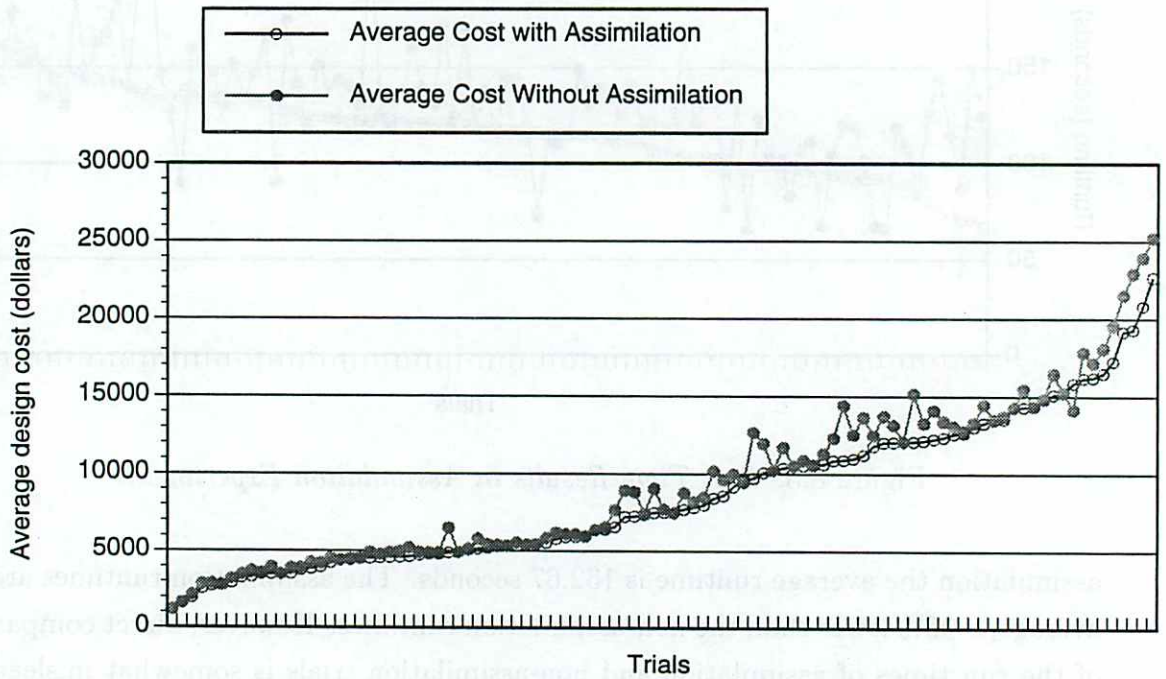


Figure 8.2. *Average Solution Quality Results in Assimilation Experiments*

In this case, the assimilation trials resulted in sets of solutions that were 6.57% better on average. This result suggests a cumulative effect: when larger numbers of solutions are counted in each trial, the percentage improvement increases. This is

system enters a termination phase during which all acceptable partial solutions are completed with the intent of finishing potentially good solutions that are initiated later in problem solving. This policy is appropriate for the STEAM domain since solution quality is a higher priority than run time. If it is assumed that agents are assimilating information, agents are more likely to initiate good solutions as information is incrementally accumulated, i.e, solutions get better as problem solving progresses. This means that there are likely to be more acceptable solutions in the assimilation trials than in the non-assimilation trials and, as shown in Appendix F, this is indeed the case. However, this leads to a bias in which direct runtime measures favor non-assimilation trials. In those trials, the system doesn't complete as many solutions as it does in the matching assimilation trial because it isn't able to focus on mutually acceptable regions of the composite search space.

To make runtime comparisons more meaningful, we divided the run time of each trial by the number of acceptable solutions, resulting in a *runtime-per-solution* measure. The results obtained using this method are listed in Appendix F and graphed in Figure 8.4.

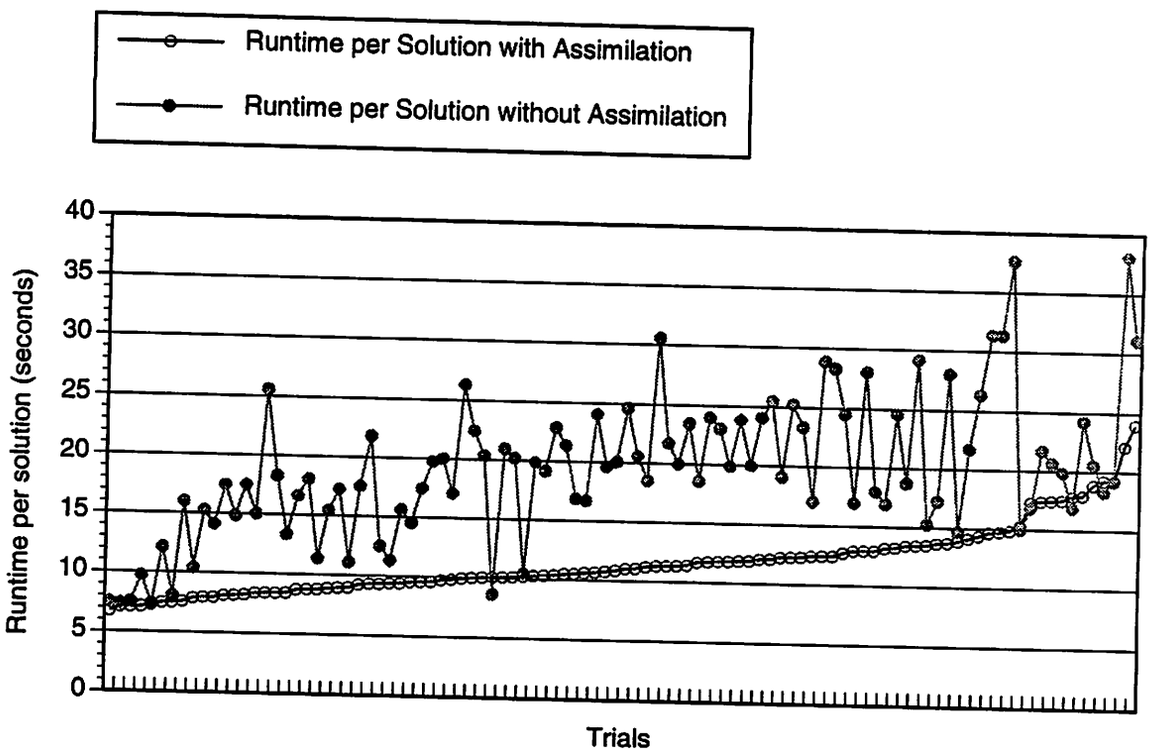


Figure 8.4. *Runtimes-per-Solution Results in Assimilation Experiments*

The *runtime-per-solution* observations in the assimilation and non-assimilation trials were averaged over the 100 experiment sets for comparison. The average runtime

per solution in the assimilation trials was 11.58 seconds and in the non-assimilation trials it was 19.50 seconds. The average percent improvement achieved by the assimilation trials over the non-assimilation trials in runtime-per-solution was 40.62%.

Although comparison of the *runtime-per-solution* measure is more meaningful than that of actual run times, it is still a crude measure. It does not take into account factors that skew the overall run time of the system. For example, there is an initial setup time during which agents have a completely local view (before any constraining information has been shared). During this time, the agents are unable to perform any better in the assimilation than in the non-assimilation trials. If the problems being addressed were "harder" in the sense that they took longer to solve, this setup time would be a less significant part of the overall effort and the comparison between *runtime-per-solution* measures would be expected to be more favorable to the assimilate trials. The opposite is also true: if the problem sets were "easier" it is expected that the measures would be more favorable to the non-assimilate trials.

8.2 Measuring the Costs and Benefits of Information Sharing

Under normal conditions, the STEAM system showed much improvement when information was shared among agents. The solution-quality improvement was approximately 5.72% and the performance improvement, as measured by *runtime-per-solution* was 40.62%. In this section, we move away from general measurements of quality and performance and, instead, try to understand the underlying costs and benefits more directly. We begin by analyzing the components of information sharing: what does information sharing entail at each agent? We continue by reexamining some of the experimental data on system performance and quality with different assumptions about information-sharing costs.

Sharing information has five primary costs:

1. constraint generation costs: generating restricting information that can be transmitted to other agents to guide their local searches;
2. determination of transmission information costs: determining which constraints to transmit to other agents at a given point in problem solving;
3. transmission costs: the actual costs associated with the physical transfer of messages among agents;
4. translation costs: translating constraining information from the local language to a sharable format at the sending agent and then translating the information from a sharable format into a local language at the receiving agent;

5. local management costs: determining the applicability of the information at the receiving agent (sorting, filtering, detecting conflicts and locally resolving those conflicts) and managing the greater volume of restricting information that results from accumulating received information (storage and retrieval costs).

In these experiments, we assume that transmission costs (those associated with the physical transfer of information from one agent to another) are minimal. This is consistent with the current implementation of STEAM in which all agents reside on the same machine and run in the same process. If the TEAM framework is physically distributed across multiple processes and machines in the future (see Chapter 10), this assumption will be reexamined. At present we do not track transmission costs separately, but do break down the other information-sharing costs.

8.2.1 Constraint Generation

The ability of an agent to generate restricting information to send to other agents is highly domain- and representation-dependent. There are three primary types of information that can be sent:

1. constraints that are completely independent of the specific problem being addressed (*independent* constraints)
2. constraints that are dependent only on the problem specification without regard to any particular solution (*problem-dependent* constraints)
3. constraints that are dependent on existing instantiated parameters for a particular solution (*solution-dependent* constraints)

To illustrate these different constraint types, we provide examples from the STEAM domain, beginning with *independent* constraints. In parametric design, there often exists a catalog of stock components, such as pumps. For a particular design, the choice of pump imposes constraints on components it interacts with. A stock pump will specify the minimum power required to drive it (thereby influencing the choice of a motor) and the maximum water flow rate it can generate (thereby influencing the design of a heat exchanger). These characteristics are independent of both the system in which the pump is embedded and any solution that uses that pump, i.e., you can look up the maximum water-flow rate a specific pump can deliver in the catalog without any regard for the problem at hand. This is an example of an *independent* constraint. Independent constraints can be generated prior to runtime problem solving and need only be generated once regardless of the number of different problems the agent addresses. The cost of generating independent constraints is not

included in the experimental results because we consider it a one-shot preprocessing cost rather than a cost that is integral to the problem solving.

With *problem-dependent* constraints, some value that is used to restrict the solution space must be calculated based on the value of a parameter that is specified by the user at runtime. For example, in STEAM the user provides a problem specification that establishes a value for the parameter `required capacity`. The heat-exchanger agent uses this value in calculating water-flow rate and head for a design. No constraining information about values for those variables can be calculated prior to runtime since the required capacity must be specified first. However, once it is specified, some constraining information can be calculated immediately. This information need only be calculated once until a problem specification with a different required capacity is entered.

Solution-dependent constraints are those in which some value of the constraint is dependent upon the instantiation of some other parameter in a solution. For example, the pump agent might attempt to extend a design that has an preexisting water-flow-rate attribute of 200 gallons per minute. The minimum horsepower required for any pump that can deliver that rate is 20 horsepower. The pump agent can send this constraining information to the motor agent so that the motor agent will know not to bother proposing a motor with a lower delivered horsepower, but the constraint should only be applied to solutions where a water-flow rate of 200 gallons per minute has already been established. Solution-dependent constraints must be generated during run time.

In the experiments reported here, we investigated primarily the use of problem-dependent constraints. Costs associated with independent constraints are not considered to be part of the normal cost of developing a solution. Solution-dependent constraint generation and manipulation techniques are difficult in this domain because of the combinatorics of potential value interactions. In some domains, it may be possible to exploit solution-dependent constraints. For example, in their work on multistage negotiation, Conry et. al. have developed a formalism that explicitly represents the interactions of solution-dependent constraints among subplans and uses these constraints to either derive a solution or determine that no solution exists [Conry *et al.*, 1991]. We do not handle solution-dependent constraints in the dissertation however.

8.2.2 Determination of Transmission Information

An agent must decide what information to transmit. As discussed earlier, in negotiated search agents transmit information directly in response to conflict situa-

tions rather than transmitting information that is anticipated to be potentially useful. Therefore, only constraints that are in direct conflict with an existing solution are transmitted. The costs of retrieving potentially conflicting constraints and checking each constraint to see if it conflicts with the existing solution are reported in the determination-of-transmission-information measure.

8.2.3 Translating Constraints from Local to Shared or Shared to Local Format

Local restricting information can be represented at an agent in any form that is appropriate for that agent. However, in our multiagent systems, each agent must take responsibility for representing information to be shared in a globally acceptable format. The cost of translating from local to shared format, or vice versa, can vary greatly depending on exactly what is entailed. Some agents may use the shared language locally and have no translation costs, others may translate using simple syntactic procedures, and others may require complex semantic translation.

In **STEAM**, all agents use the same translation mechanisms. Local languages for all agents are subsets of the shared language. Objects in a local language are resident within the local package. A package is a software engineering tool for large systems that ensures privacy among modules by prefixing each object in that package with a special marker. Specifically, in **TEAM** application systems, each agent has a private package in which all its local objects are defined. There is also a **TEAM** package which is globally accessible to all agents and to the framework mechanisms. In order to share information, an object is copied from the local agent's package into the globally accessible **TEAM** package. Then, for every agent receiving the information, it is copied from the global package into the local package of the receiving agent.

8.2.4 Local Management Costs

Conflict between local and assimilated information is one factor that potentially mitigates the benefits of information sharing: what happens when an agent receives information that contradicts something it already knows? With logically heterogeneous agents, it must be assumed that conflict will occur and that any method of assimilating conflicting information will be costly and will potentially lead to degraded system performance.

When constraining information is received by an agent, the agent first translates the constraint into the local language (from the shared language) and looks to see if it is syntactically identical to a previously received constraint. If so, the constraint is ignored. Otherwise, it is stored in a local database. No attempt is made to sort the

new constraint or determine its applicability at this time. Constraints are stored so that they can be retrieved based on attributes of:

1. the name of the parameter restricted by the constraint;
2. the type of predicate applied by the constraint;
3. parameter names addressed by the predicate;
4. instantiated parameter values;
5. the quality level of the constraint;
6. the agent that originated the constraint;
7. a boolean relaxation-state: true if the constraint has been relaxed, false otherwise.

When an agent is developing a proposal, either to initiate a new design or to extend an existing one, it retrieves all constraints that are relevant to the current situation and finds the most restrictive but non-conflicting set possible. Constraints that are relevant to the search are defined as follows: for each parameter in the search space, the agent retrieves all constraints that 1) name that parameter, 2) are applicable at the agent's current required quality level or lower quality levels, and 3) have not been previously relaxed.

Retrieved constraints that define intervals on a parameter space (see the discussion of parameter spaces in Chapter 3) are sorted into two groups: constraints that define minimum boundary values and those that define maximum boundary values. (*water-flow-rate* <= 404.34) defines a maximum boundary constraint for example. From each group, the most restrictive constraint is chosen. (For maximum boundaries, the most restrictive constraint is that with the lowest value and vice versa.) When an explicit conflict between the minimum and maximum boundary constraints is found (the minimum boundary is greater than the maximum boundary), one of the conflicting constraints is locally relaxed and the process is repeated. The conflict-resolution algorithm we have used for all of our agents is detailed in Section 5.4.3.3. Once a non-conflicting boundary set is found, it is used to restrict the search for a proposal. If no proposal can be found using these constraints, the agent chooses a constraint to relax (the choice process is described in Section 5.4.3.2), thereby expanding the local search space. This process is repeated until a proposal is found, until the local constraint relaxations change the global state of the system by making existing solutions acceptable, or until there are no further relaxations that can be made.

The boundary determination process described is applicable whether the agent has assimilated external constraints or is working only with internal constraints.

However, there are two differences between these situations. First, when working only with internal constraints, each agent is considered to be locally consistent, so explicit conflicts should not occur. Second, the volume of constraints and therefore the costs of storing, retrieving, and sorting constraints will be higher when external constraints are assimilated.

With reference to the assimilation experiments described in Section 8.1, it is encouraging to note that significant performance improvements were achieved even when sharing only the simple form of information embodied in boundary constraints. It would be worthwhile in future work to look at the tradeoffs between sharing more complex or specific forms of information and the costs of embedding the mechanisms required to use that information appropriately into heterogeneous agents.

8.2.5 Experiments with Information-Sharing Costs

In these experiments, the costs attributed to sharing information are broken down into the categories: 1) constraint generation (for problem-dependent constraints); 2) transmission determination; 3) constraint translation; and 4) local management, as described earlier. The observed costs for each of these categories over the 100 feasible problem specifications in Appendix B are tabulated in Appendix F and summarized in Figure 8.5.

The approximate average time spent in constraint generation per problem is 7.3 seconds, constraint transmission is .4 seconds, translation is .5 seconds and local management is 4.2 seconds, for an average total time for information assimilation of approximately 12.4 seconds (out of 121.98 seconds average runtime). The average total percentage of time spent in information sharing in these trials is 10.17%. These figures are highly domain-dependent and each of the different areas could be more or less expensive in other situations. For example, if more elaborate constraints were being generated, the constraint generation time would no doubt be higher and consume a larger proportion of the processing time. Likewise, if translation were more difficult and involved some semantic interpretation as well as strict syntactic replacement, it would take more time. The question that must be asked is not whether information sharing and assimilation takes time–it does. Rather, the questions to ask are:

1. What information can be shared by each agent?
2. For each type of information that can potentially be shared, will sharing it decrease or increase the processing time of the system?
3. For each type of information that can potentially be shared, will sharing it improve solution quality in the system?

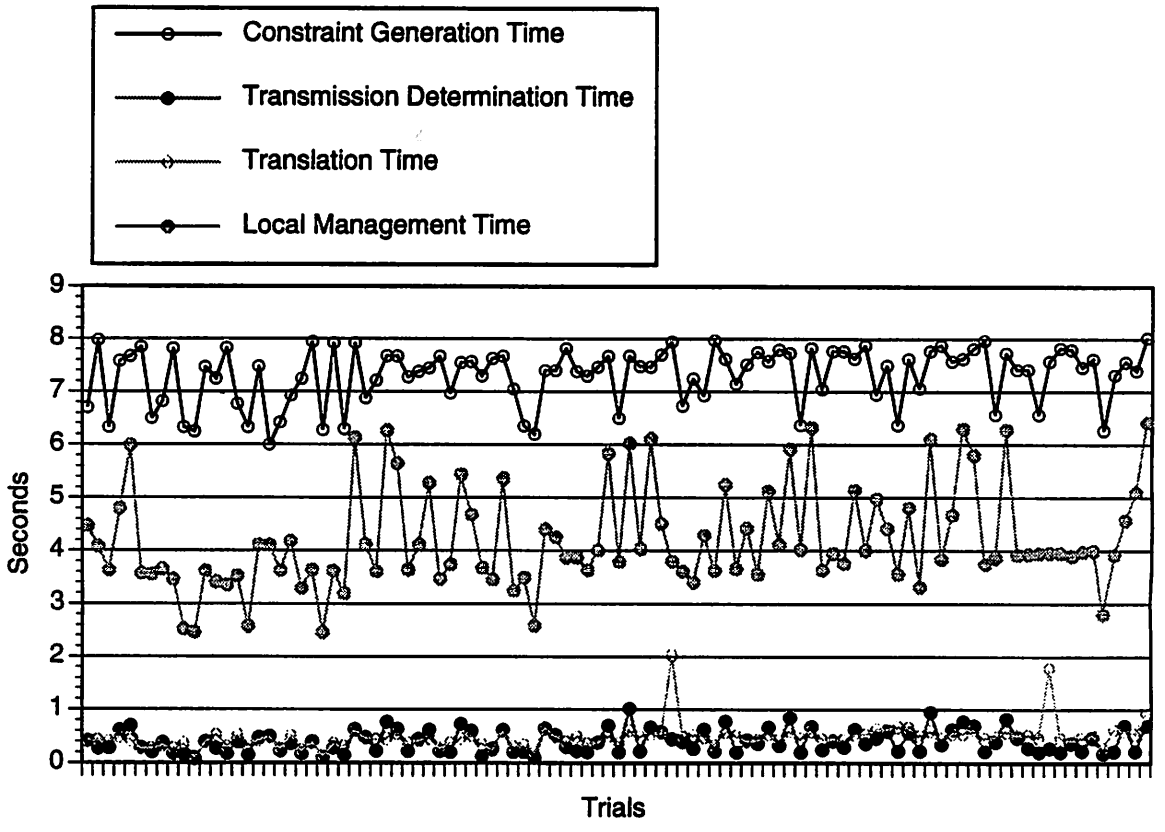


Figure 8.5. *Information-Sharing Costs in* STEAM

Although it is possible to empirically answer these questions for a particular system and, therefore, to tune the system to appropriate tradeoffs in quality and processing time based on information sharing, the questions are much more difficult when agent reusability is an issue. The agent implementor cannot be responsible for determining what information will be relevant in an application system and yet the agent implementor must be responsible for determining what internal information will be sharable.

As discussed above, a major problem with agent reusability is deciding what information to share–will sharing a certain piece of information generally help an application system or hurt it? In the following section, we describe a secondary problem relating to information sharing in reusable-agent sets: how difficult is it for an agent implementor to anticipate the types of information that may become available to the agent during problem solving and to build into the agent the capabilities required to effectively apply that information?

8.3 Using Assimilated Knowledge

In the experiments described here, only three of the seven agents instantiate the information-assimilation operators. The primary reason for this is that implementing these operators is non-trivial and time-consuming. For each agent, the implementation is unique and requires a thorough understanding of the knowledge requirements and search mechanisms of that agent. This suggests that it must be done by the agent implementor at the time the agent is built.

We demonstrate the difficulty inherent in implementing effective information assimilation through an example. Say that the **pump agent** receives a constraint from the **heat-exchanger agent** that restricts the *run-head* parameter of pumps proposed by the pump agent. This constraint is not directly applicable during the search for candidate pumps because the value of *run-head* is computed after the specific pump is chosen: it is an output parameter rather than an input parameter. However, once a candidate pump has been generated, the *run-head* for that pump can be computed and the constraint can be applied as a filtering mechanism to eliminate non-viable candidates. If pump-agent does apply the filtering constraint, it will still have to iteratively generate and test candidate pumps locally, but will eliminate infeasible ones before other agents are asked to respond to them. Therefore, by appropriately applying assimilated information, it can reduce the workload of other agents.

The point here is that is not only necessary to understand the language of received information, it is also necessary that the agent know how to apply it. Applying the knowledge appropriately can be very subtle because it may have to be applied differently than the agent's own local knowledge, for example, as a post-search filter as described above. This implies that an agent must anticipate the kinds of information it may receive and have internal procedures available to effectively use that information.

Reflecting the difficulty of effectively assimilating information, only three of the seven agents in **STEAM** currently have the requisite expertise to do so. In future work, we will expand the set of agents implementing information-assimilating operators and reexamine the issues explored in this chapter on a larger scale.

8.4 Summary

In this chapter, our objective was to show that the ability of agents to assimilate external information about the composite search space and use this information to guide local search affects both solution quality and system performance. The experiments in Section 8.1 demonstrated conclusively that this is so. When external

information is assimilated by an agent, that agent is able to focus its search efforts in areas of its local solution space that are more likely to be contained in the composite solution space as well. By focusing its search in areas that are likely to be mutually acceptable, the agent's work is more likely to be productive and will tend to improve both solution quality and system performance. However, there are implementation and performance costs associated with information sharing and, in some situations, these costs may outweigh the benefits.

In order for an agent to assimilate information, it must implement the assimilation operators described in Chapter 5. These operators are intended to provide a foundation for standardizing the interactions of agents with respect to information sharing. However, the details of how the operators are implemented within a particular agent, what information is available to be shared, how conflicting information is handled within an agent, and how assimilated information is applied by the agent are highly agent- and domain-specific. We provided workable examples for each of these issues within the **STEAM** system and look forward to developing more elaborate techniques in future work.

In the experiments described in this chapter, we controlled the ability of agents to assimilate and apply non-local information through activation and deactivation of assimilation operators. Shared information was limited to simple single-clause constraints that defined boundary values for shared numeric parameters. The experiments were run in **STEAM** under the *negotiated-search* strategy on problem specifications 1-100 as shown in Appendix B. Each experiment consisted of two trials: one in which the assimilation operators were active and one in which they were not.

We found that when the assimilation operators were active, there was an average 5.72% improvement in solution quality as measured by the cost (in dollars) of the lowest-cost design. These results demonstrate that assimilation of non-local information has a significant effect on solution quality. We believe that a floor factor is operating in this domain, that is, the solutions are good enough, even in the non-assimilation trials, that there isn't much room for improvement. It is certainly possible that a more dramatic improvement in solution quality could be obtained in a situation where problems are "harder" though testing that hypothesis is not within the scope of this dissertation.

We also found an average 8.06% improvement in runtime (seconds) and a 40.62% improvement in the runtime-per-solution measure (runtime in seconds over the number of acceptable solutions generated) with the assimilation operators activated. This improvement is significant, particularly in light of the extra overhead involved in transmitting and assimilating information and resolving conflicts that are detected.

It is clear that, in the STEAM system at least, the ability to focus local search on areas of the solution space that are likely to be within the composite solution space greatly enhances system performance. In these experiments, the increase in time spent in control overhead is outweighed by the decrease in time spent generating proposals that will be rejected by other agents and by improvements in solution quality.

After empirically demonstrating the benefits of information assimilation in multiagent problem solving, we took a detailed look at the costs of assimilation. We classified the costs of information sharing as involving: the generation of information to share; the translation of information into and out of a shared language; the determination of what information to communicate at any given time; the transmission of information[3]; and local management (retrieval and use of potentially conflicting assimilated information).

Although we believe that the significance of each of these categories of costs will vary across agent sets and application systems, we measured the costs (in terms of time spent performing the tasks associated with each category) in the STEAM system to provide some initial benchmarks. The average percentage of time spent in information-sharing tasks was approximately 10.17% in the experiments. Although this is not a trivial figure, in this domain, the time spent in sharing information is more than balanced by the productivity enhancement that comes from focusing on mutually acceptable areas of the composite solution space.

Our experience with information assimilation suggests some conflicting perspectives on achieving systems of heterogeneous and reusable agents. First, in the experiments in this chapter, we showed that information sharing and assimilation can be highly effective in improving system performance, both in terms of solution quality and runtime. On the other hand, we found sharing and assimilation difficult to actualize because they require in-depth understanding of the domain characteristics of individual agents. This suggests that reusable-agent systems will not be able to take advantage of information assimilation unless the problem is explicitly addressed at agent-implementation time. The application system developer that is responsible for integrating a set of reusable agents into a system cannot be expected to have a deep enough understanding of individual agent domains to install the necessary mechanisms into the agents.

Information sharing requires that each agent know: 1) what information it can share; 2) what information it can assimilate; and 3) how information that is as-

[3]Although we recognize that transmission of information will add to the cost of information sharing, it was not included as one of the categories in our experiments. Because of the physical environment in which our experiments were run, these costs were trivial as described in Section 8.2.

similated from external sources is to be used. When the mechanisms required for information sharing are installed at agent-implementation time, the implementor will not know whether shared information will ever be used, whether anticipated information will ever arrive, or whether functional capabilities for applying certain types of information will ever be used. If the agent is implemented with highly sophisticated information-sharing capabilities, it must be expected that in any given application system, these capabilities may be well beyond what is required or even usable for the domain. The price of generality goes beyond the implementation costs for the agent as well, since there may be runtime repercussions based on the transmission of unusable information, or on applying assimilated information that degrades system performance rather than enhancing it. Future research in reusable-agent systems should examine questions of balancing the information-sharing capabilities of agents with the benefits of sharing various types of information. It may be that some general guidelines will emerge that can be applied by agent implementors to decide what capabilities are likely to be beneficial in an agent.

In conclusion, we have shown that information sharing and assimilation can enhance system performance in the STEAM system. Can this result be generalized? Although there is no basis on which to generalize any specific figures outside of STEAM, the STEAM domain is typical of a class of small-scale globally cooperative design domains. This leads us to believe that information sharing and assimilation can improve performance in this class of systems. Furthermore, the categories of information-sharing costs described in this chapter hold across all domains. Both the empirical evidence demonstrated here and intuitive arguments for the benefits of focused search suggest that information sharing and assimilation will be effective in more complex domains. However, although information sharing is potentially beneficial, it is not particularly easy to achieve. Most of the work must be done at agent-implementation time when nothing is known about the application system(s) into which the agent will be embedded. The costs of making agents that are highly proficient in sharing and using assimilated information may outweigh the benefits that accrue from those capabilities. Future work may clarify the boundaries of benefit versus hindrance based on types of information and the capabilities required by agents to use those various types. Until that time, we can say that information sharing and assimilation should be considered a potential source of performance enhancement when designing globally cooperative multiagent search systems.

In this chapter, we looked at how information sharing within a statically organized agent set affects system performance. In the next chapter, we will examine another

potential influence on system performance: the assignment of agents to particular roles within a distributed-search strategy.

CHAPTER 9

AGENT/OPERATOR ROLE ASSIGNMENTS IN NEGOTIATED SEARCH

In reusable-agent systems, each agent must be capable of playing some role in the distributed-search strategy that guides the interaction of agents. Within the context of a single solution, each agent has an assigned role (comprising a task or tasks implemented as operators). Agents may be capable of playing multiple roles within a strategy. For example, in the negotiated-search strategy described in Chapter 5, the set of tasks to be performed in constructing a solution include initiating, extending, and critiquing the solution, and relaxing solution requirements. The roles that an agent can play are *initiator*, *extender*, or *critic*. An agent may be an initiator for some solutions and an extender or critic for others (in which case the agent requires some mechanism for deciding which role to undertake with respect to a particular solution). It may also be true that more than one agent can play each role, i.e., two or more agents can initiate, extend, or critique solutions within a single agent set. More than one task may be associated with a single role, and the same task may be associated with multiple roles. In the example given above, the relax-solution-requirement task is associated with all three of the negotiated-search roles. A *role*, therefore, defines a task or set of tasks to be performed with respect to a single solution and a *role assignment* activates the appropriate operators at a particular agent to achieve the desired role.

In [Cammarata *et al.*, 1983], Cammarata, McArthur, and Steeb note:

> Agents typically have differing *appropriateness* for a given task. The appropriateness of an agent for a task is a function of how well the agent's skills match the expertise required to do the task, the extent to which its limited knowledge is adequate for the task, and current processing resources of the agent.

In this chapter, we go beyond investigating the appropriateness of an agent for a particular task and instead ask whether an agent can appropriately play a particular role in a distributed-search strategy. We will describe experiments that were done to investigate the importance of careful assignment of roles to agents. We wanted to determine if the assignments made a difference in the quality of solutions and in the

processing time required to find solutions. Assuming that there would be differences in performance, we wanted to determine if there were characteristics of agents that would predict the effectiveness of a particular agent/role assignment for a particular problem.

For these experiments, we used the negotiated-search strategy within the STEAM system. As described in Chapter 6, there are seven agents in STEAM. Four of these agents can only extend solutions: `motor-agent`, `shaft-agent`, `vbelt-agent`, and `platform-agent`. The `frequency-critic` can only critique solutions. The other two agents, `pump-agent` and `heat-exchanger-agent`, can initiate new solutions or extend existing ones. Given this agent set, and the roles of *solution initiator*, *solution extender*, and *solution critic*, we set up three possible agent/role assignments as shown in Figure 9.1. We ran the system once for each role assignment on problem

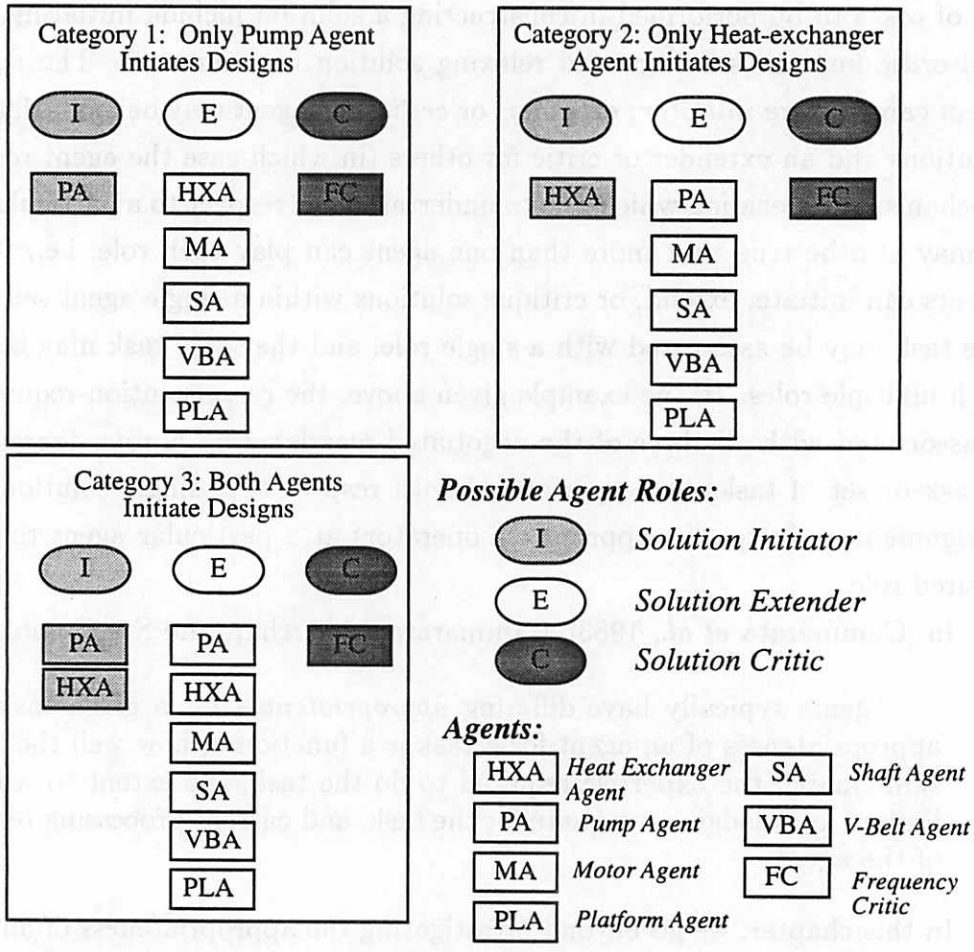


Figure 9.1. *The Experimental Agent/Role Assignments*

specifications 1-100 as shown in Appendix B. In the following sections, we analyze

the results for each configuration, and discuss some of the underlying characteristics that influence the results.

To simplify the discussion of the results, we will first describe a simple, generic, two-agent example. By looking at a simpler problem, we can more easily describe features of the agents and their solution spaces that affect quality and performance results. We will present the experimental results after the example.

9.1 A Two-Agent Role Assignment Example

In our example, there are two agents, A and B. The agents' solution spaces over the shared parameters x and y are shown in Figure 9.2. These agents can communicate

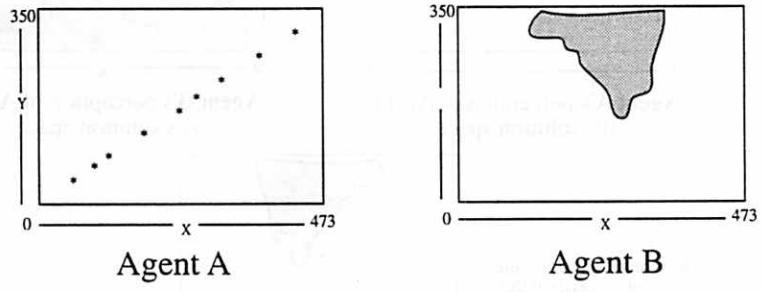


Figure 9.2. *Solution Spaces of Agents A and B over x and y*

simple boundary constraints such as ($x < 473$) that specify the minimum or maximum value that can be assigned to a parameter. They cannot communicate relationships that are procedurally described. For example, let Agent A have a functional relationship between x and y of the form: {`PROCEDURE Calculate-X-From-Y (Y) Return X := Y+1`}. Agent A cannot inform Agent B that x will always equal $y + 1$ because the information is represented as a function. This is a very simplistic example of a functional relationship. In general, functional relationships are more complex, e.g, in the steam condenser domain, the relationship between parameters `water-flow-rate` and `head` is a function involving domain-specific calculations over locally defined variables such as water input temperature, viscosity, and velocity.

If minimum and maximum boundary constraints are available and communicated to and from both A and B, an approximation of the intersection of local solution spaces (the composite space) can be generated by both agents. If every point within the communicated boundaries is an actual solution, either agent should be able to initiate and/or extend solutions equally well since both agents can see exactly where solutions lie. However, as described in Chapter 3, when there are implicit constraints that cannot be communicated, such as those imposed by functional relationships, not

every point within the constrained space will actually be a solution. To clarify this, assume that Agent A transmits its boundary constraints, $(0 < x)$, $(x < 473)$, $(0 < y)$, and $(y < 350)$ from Figure 9.2, to Agent B and, likewise, Agent B transmits its constraints, $(60 < x)$, $(x < 150)$, $(150 < y)$, and $(y < 350)$, to Agent A. The agents would perceive each other's solution spaces as shown in Figure 9.3. Because there are functional relationships that are not communicated, however, the real composite space is the intersection of the actual local spaces as shown in Figure 9.3.

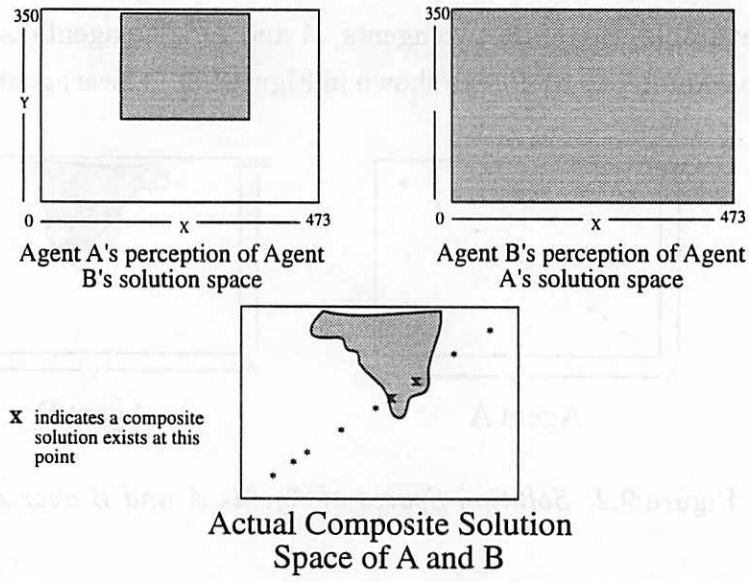


Figure 9.3. *The Local Perceptions and Actual Composite Space of Agents A and B*

Agent A has a very sparse solution space. This means that the other agent is unlikely to propose an acceptable solution. For example, say that only 5% of the points that Agent B believes are in the composite solution space actually represent solutions that will be acceptable to Agent A. Then Agent B will statistically only propose mutually acceptable solutions 5% of the time. Agent A cannot provide any further guidance to B because the required information is not represented in a shareable way.

This situation is not uncommon and occurs in the experiments that were performed (as further described below). The `pump-agent` has a functional relationship between its parameters *water-flow-rate* and *head* that constrains the set of acceptable solutions. This relationship cannot be communicated because, as explained in Chapter 8, `STEAM` agents communicate can only simple boundary constraints.

Returning to our generic example, if A is the only agent in the system with this type of solution space, it should be the controlling agent in the system. That is, A

should initiate proposals and other agents should send back constraining information that would help to guide A's local search. In the context of the distributed-search roles defined above, A should take the role of *solution-initiator* and not the role of *solution-extender*. Agent A is highly constrained by its functional relationships among parameters and can generate solutions with the appropriate relationships, but cannot communicate that knowledge to assist other agents in their local searches. Other agents should only take the role of *solution-extender* since they are unlikely to generate initial proposals with the correct parameter relationships and are better suited to providing guiding information to Agent A.

Now, consider the case where two agents have sparse solution spaces and functional relationships that cannot be communicated. This forms a very problematic agent set: the best-case scenario is that by random initiation and extension of designs, eventually the two agents find an overlapping solution[Lander *et al.*, 1991a]. In this situation, both agents should instantiate *initiate-solution* and *extend-solution* since neither has an advantage in controlling the interactions. If the composite space is large and sparse, any mutually consistent solution found may be considered acceptable. In practical terms, this means that if it is known that the composite solution space is sparse, agents should readily relax constraints. On the other hand, if the composite space is small enough and dense enough that the agents are likely to find a good fit eventually even if they cannot provide good guiding information, the agents can be less eager to relax.

9.2 Experimental Results

In these experiments, we tracked two measures of system performance: the cost of the best design (in dollars) and the time it took for each system run (in seconds). As in Chapter 8, designs are rated by cost and the "best" design is defined to be the lowest-cost complete acceptable design. The full results from running the system under the agent/role assignments described in Figure 9.1 are listed in Appendix G.

9.2.1 Experimental Results on Solution Quality

We will first analyze the results in terms of solution quality. As noted above, for each problem specification, the system was run three times, once with `pump-agent` acting as the only solution initiator (Category 1), once with `heat-exchanger-agent` acting as the only solution initiator (Category 2), and once with both of these agents initiating solutions and extending each other's solutions (Category 3). It was expected that the results in Category 1 would be the best because `pump-agent` had the most

restrictive information that was not communicable, as described above. Figure 9.4 shows how each of the agent/role assignments (Categories 1–3) performed in solution quality as measured by the cost of the best design. In this figure, the results are sorted into ascending order based on the cost of designs in the Category 1 trials. The average

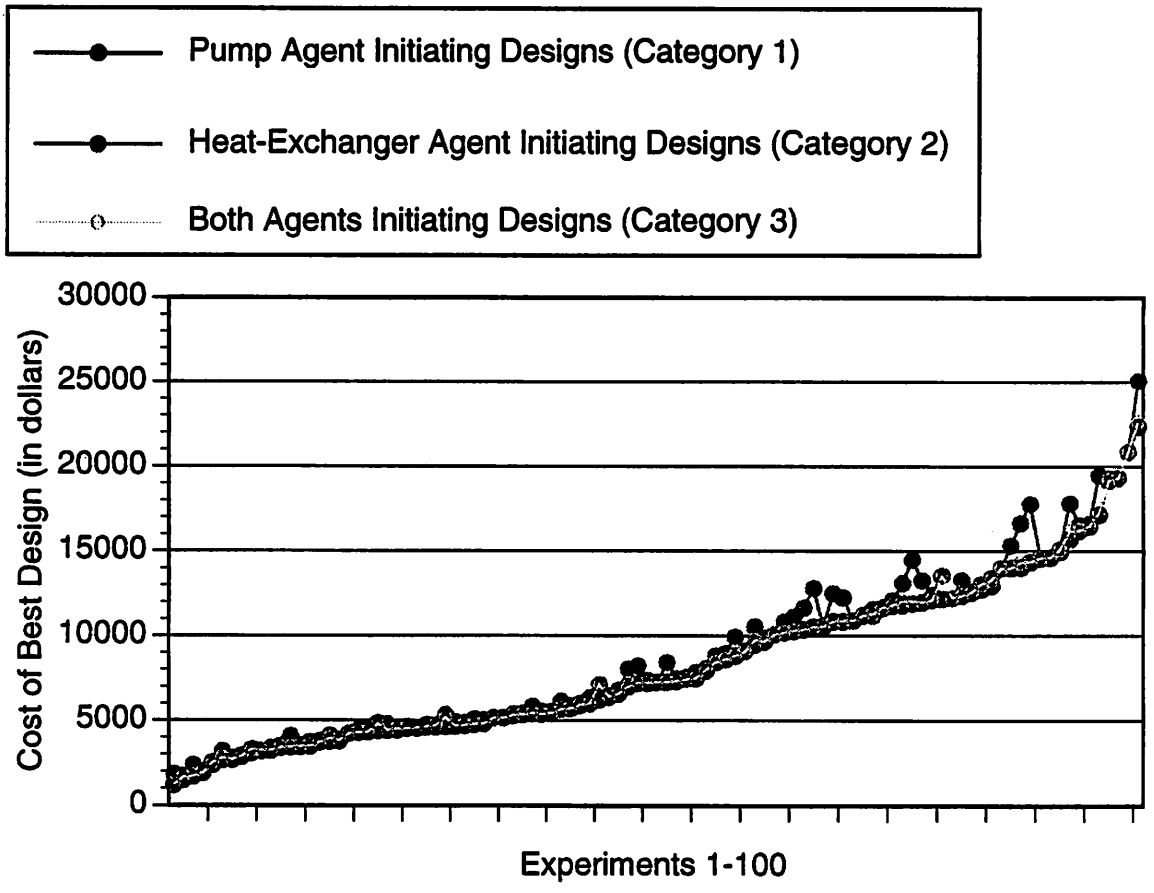


Figure 9.4. *Solution Quality Results in Agent/Role Assignment Experiments*

cost of a design in Category 1 was \$8313.88, Category 2 was \$8798.64, and Category 3 was \$8375.77. The percentage improvement in Category 1 with respect to Category 3 was 5.51%. The best average costs were obtained in the Category 1 role assignment, where `pump-agent` initiated solutions and `heat-exchanger-agent` extended those solutions. Category 2, in which `heat-exchanger-agent` was the controlling agent produced the worst average costs. These results were expected since `pump-agent` had an important functional relationship between two of its parameters which it was unable to share (as discussed above). This means that when `heat-exchanger-agent` acted as initiator, it had no way of intelligently guiding its search process.

The difference between solution costs in Category 1 and Category 3 were marginal. This seems reasonable since `pump-agent` was initiating solutions in both of those cat-

egories. However, in Category 3, **pump-agent** was splitting its time between initiating new solutions and extending solutions generated by **heat-exchanger-agent**. In this category, therefore, **pump-agent** was "distracted" by the need to respond (negatively) to designs initiated by **heat-exchanger-agent** and therefore initiated fewer solutions in a run. The result of this distraction was that the best Category 3 designs were often, but not always, identical to those in Category 1 since the timing of the solution initiation sometimes affected whether a particular solution was ever generated in Category 3.

Distraction is a phenomenon which appears frequently in distributed artificial intelligence literature. Although agents must be responsive to each other, in order to handle distraction, they should also maintain a healthy degree of skepticism[Corkill and Lesser, 1983]. Otherwise, it is possible for an agent to spend too much time trying to respond to poor solutions from other agents. Ideally, an agent should decide what the best solution is to work on at a particular point in processing, whether that solution is locally or externally controlled. In the specific context of the role assignment experiments described here, an agent must judge whether it is most effective to work on its own designs or to respond to designs generated by other members of the group given the current set of conditions. In Section 9.3, we will discuss the possibility of dynamically adjusting agent/role assignments during problem solving as information about both local and external solution spaces becomes available.

Returning to the experiments, another factor that affected the results was that although **pump-agent** generally did better at initiating solutions, occasionally a solution initiated by **heat-exchanger-agent** turned out to be the best. This is in keeping with our earlier discussion of the generic two-agent example where we stated "For example, say that only 5% of the points that Agent B believes are in the composite solution space actually represent solutions. Then Agent B will statistically only propose mutually acceptable solutions 5% of the time." In our experiments, **heat-exchanger-agent** (equivalent to Agent B in the generic example) produced mutually acceptable solutions only a small fraction of the time, but did occasionally produce one and, in fact, occasionally outperformed **pump-agent** in terms of solution quality. Therefore, in the Category 3 experiments with both agents initiating solutions, it is occasionally the case that the best solution is initiated by **heat-exchanger-agent**, and that that solution is better than the associated Category 1 result.

9.2.2 *Analysis of Runtime*

We hypothesized that the runtime in the Category 1 trials would be less than that in the Category 2 and 3 trials, because we believed that **pump-agent** was better at initiating solutions than **heat-exchanger-agent** and would therefore find acceptable solutions faster. System performance in terms of runtime for each of the three agent/role assignments is detailed in Appendix G and summarized in Figure 9.5. In the figure, results are sorted into ascending order based on runtime in Category 1. The average runtime in Category 1 was 173.51 seconds, in Category 2 it was 169.0

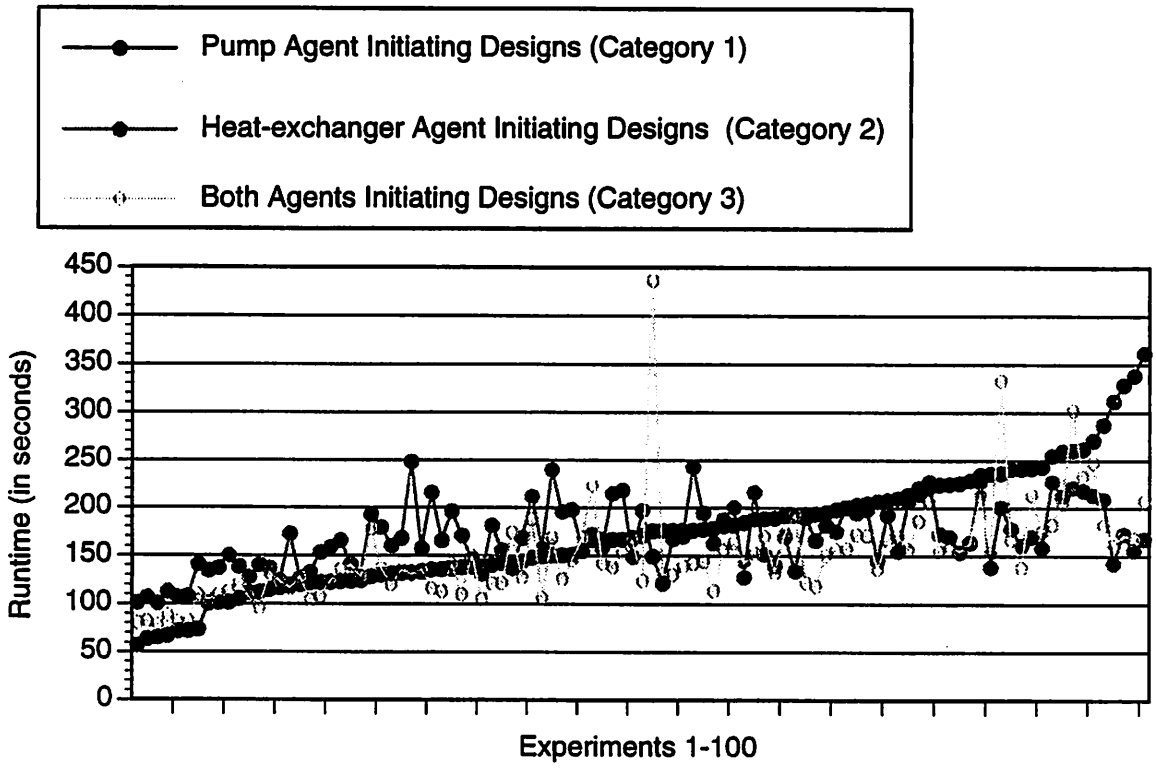


Figure 9.5. *Runtime Results in Agent/Role Assignment Experiments*

seconds, and in Category 3 it was 151.48 seconds. Contrary to our expectation then, the best runtime results were obtained in the Category 3 role assignment where both agents were simultaneously initiating and extending solutions and the worst results occurred in Category 1 which we had expected to be best.

In Chapter 8, we found that runtime was not a particularly good indicator of system performance with respect to information assimilation. This was because when information was assimilated, agents were better at producing good solutions. When the agents were better at producing solutions they actually produced more solutions in a single run than when they did not assimilate solutions because of the

termination policy that was used in STEAM. We therefore looked at the number of solutions produced in each of the different experimental categories to determine if that was also the case in the role-assignment experiments. We found that the mean number of mutually acceptable solutions produced in Category 1 was 14.99, 12.92 in Category 2, and 11.24 in Category 3. We therefore reexamined system performance in terms of *runtime-per-solution* (number of seconds per run/number of mutually acceptable solutions generated). Mean *runtime-per-solution* was 12.47 seconds in Category 1, 13.71 seconds in Category 2, and 13.82 seconds in Category 3. From Category 3 to Category 1, there is a 9.77% reduction in this measure. These results are detailed in Appendix G and summarized in Figure 9.6 where they are sorted in ascending order based on runtime-per-solution in Category 1. The results support

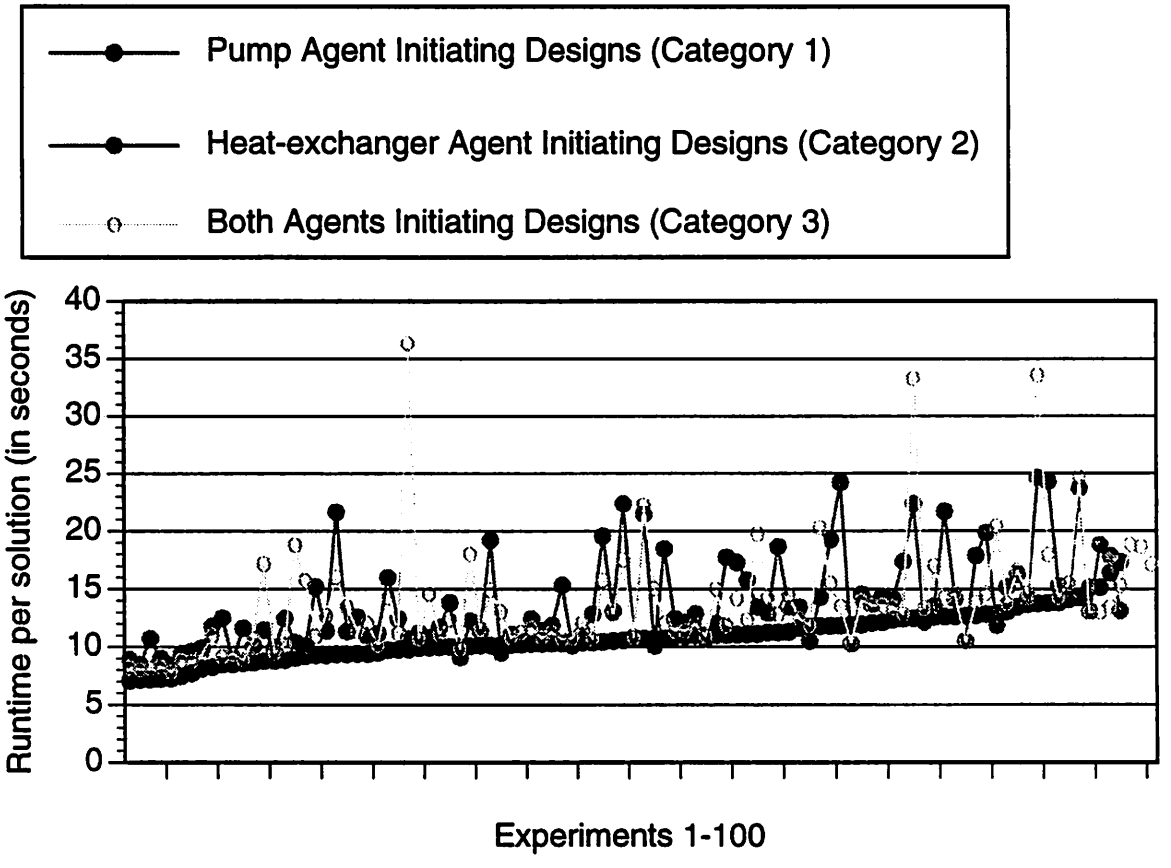


Figure 9.6. *Runtime-per-Solution Results in Agent/Role Assignment Experiments*

our original hypothesis that the best system performance occurs in Category 1, where `pump-agent` exclusively initiates solutions.

9.3 Selecting Agent/Role Assignments

The experimental results presented in this chapter indicate that the assignment of roles to agents within an agent set affect solution quality and system performance. We described characteristics of agents that may contribute to the appropriateness of a particular set of role assignments including the size, density and uniformity of the agents' local spaces, and the ability of agents to share information that allows them to accurately construct an abstraction of the composite solution space. However, in systems comprising heterogeneous and reusable agents, these characterizations cannot be fully determined until run-time. Some characteristics can be determined at agent-design time or system-integration time while others are dependent on the problem specification and must be done at runtime.

In keeping with the problem-independent and problem-dependent constraint categories discussed in Chapter 8, the effect of *problem-independent* constraints on local solution spaces can be determined at agent-design time. A one-shot preprocessing task could be used to categorize the local solution space of an agent along dimensions of size, density, and uniformity with respect to problem-independent constraints. These general guidelines could be used to choose initial agent/role assignments within a distributed-search strategy.

Still looking only at local solution spaces, *problem-dependent* constraints are more difficult to deal with than problem-independent constraints from the perspective of determining agent/role assignments because it can be prohibitively expensive to explicitly analyze the solution spaces of each agent for each new problem addressed. Furthermore, this type of analysis can't be done at agent-design time but instead must occur at run-time when the problem specification is known.

Analysis of the composite solution space of an agent set (as differentiated from the local solution spaces of individual agents) is even more restricted because nothing can be done at agent-design time. Some analysis could be done at system-integration time by preprocessing the entire set of problem-independent constraints to determine gross characteristics of the space. The effect of problem-dependent constraints on the composite solution space of an agent set must be analyzed at run-time however.

Assuming that local and composite solution-space characteristics cannot be precomputed or can only be precomputed at some gross level of abstraction, how will it be possible to assign roles to reusable agents appropriately? The only long-term solution to this problem lies in the development of agents that are able to dynamically adjust their role assignments based on their observed productivity in a particular situation. In other words, reusable agents must be able to *learn* what roles they are best able to play in a particular situation. Furthermore, they should be able to learn

how to characterize situations so as to adjust quickly and accurately when presented with a new situation or when the problem-solving situation changes dynamically over the course of a single run. Although this work is outside the scope of the dissertation, we are continuing to explore what features of problem-solving episodes affect the appropriateness of particular assignments. We plan to incorporate learning algorithms to enable agents to make both initial default role assignments and to dynamically adjust their assignments over the course of a single run as information sharing and assimilation changes local perceptions of the situation.

9.4 Summary

The role of an agent in a distributed-search strategy defines the task(s) that the agent is responsible for achieving as a participant in that strategy. For systems built within the TEAM framework, an agent's role in problem solving is realized through the activation of a set of local operators in response to a role assignment. In this chapter, we explored how particular role assignments affected the performance of the system in terms of both solution quality and processing time. We looked at why some role assignments are better than others within an agent set, focusing primarily on how an agent's local perception of the composite solution space affects its ability to initiate solutions that are likely to be mutually acceptable.

The observed results from the experiments, coupled with the analytical description of a two-agent problem that closely parallels the situation found in the seven-agent experiments, strongly indicate that role assignments do indeed affect system performance. We showed that solution quality (as measured by cost of designs) improved by an average of 5.51% and processing time (as measured in *runtime-per-solution*) improved by an average of 9.77% from the worst assignment to the best assignment. There is more experimental work to be done in isolating the various factors that influence the appropriateness of a particular role assignment, but our results give some insight into one important factor, namely, the accuracy with which an agent can communicate information about its local solution space. Since other agents depend on this information to help them focus their searches, an agent that sends imprecise information can often perform better in roles that allow it to provide more specific information through solution development. For example, if an agent has tight constraints but cannot articulate those constraints, it can instead do early assignment of variable values that other agents can then incorporate into their search.

Within systems of reusable agents, much of the information required to determine good role assignments is not available at agent-design time, and some of the information is not available until run-time. If our initial results hold up and it turns

out that appropriate role assignments are very important in system performance, the implication is that agents will have to classify role-assignment situations quickly and accurately at run-time and dynamically monitor and adjust as necessary. We see this as an important area of future work.

In this chapter, we wrapped up the presentation of new material with our final set of experiments designed to examine the assignment of roles to agents within an agent set. In Chapter 10, we will summarize the work that has been presented, and digress into the more speculative areas of future work and the implications of this work for the multiagent community in general.

CHAPTER 10

CONCLUSIONS

In this dissertation, we take an encompassing view of multiagent problem solving that spans many different issues. Some of the issues have been discussed in the Distributed Artificial Intelligence literature for some time (agent coordination, conflict resolution) and some of the issues are on the cutting edge (reusable agents, distributed search). Stated from a very high-level perspective the problem that has been addressed is how to get systems of cooperative, heterogeneous, and reusable agents to interact coherently and effectively to solve search problems when the required information and capabilities are distributed among the agents. In this chapter, we reexamine the contributions of the dissertation and present an overview of major themes and their interrelationships. We then briefly summarize what was learned through experience and experimentation. Finally, we discuss areas of potential future work.

10.1 Contributions Revisited

In Chapter 1, we introduced the contributions to the field that were expected from this dissertation. Here, we restate the list of contributions and discuss the degree to which we were able to meet those initial expectations. We also add several items that were not initially recognized as contributions: the relative importance of various techniques and concepts has been reevaluated over time.

Reusable-Agent Systems: The potential of reusable-agent systems is enormous, promising cheaper, more reliable, and easier-to-build application systems. Because of this, a great deal of attention has recently been focused on various aspects of the infrastructure necessary to support this type of system, particularly on the information-processing needs. In this dissertation, we have assumed that research on information processing in reusable-agent systems is well under way and, instead, have focused on other aspects of the problem. Reusable-agent systems must be able to: 1) solve large-scale search problems by producing solutions that satisfy all agents; 2) take advantage of knowledge embodied in existing agents; 3) effectively coordinate the actions of agents to achieve efficient and coherent problem solving; and 4) effectively

manage conflicts among agents, allowing them to find high-quality solutions despite inherent inconsistency.

The TEAM framework, described below, was developed to provide the needed infrastructure for agents to interact. Beyond the issues of physical communication and interaction, however, was the need for theories of cooperation, distributed search, coordination, and conflict management. In earlier chapters, we discussed various approaches taken by researchers to address parts of the general problem being investigated. Formal models have been used to describe different types of cooperation and negotiation under restricted conditions, assuming highly accurate and predictable information sharing. Human models of conflict management were discussed, particularly with reference to situations of adversary conflict. Multiagent computational systems have been designed to explore specific aspects of cooperation, conflict management, information processing and its necessary infrastructure, and agent coordination. Some initial work has been done on designing and investigating the properties of search algorithms for distributed systems.

All of these approaches address some aspect of the more general problem and all have contributed conceptual understanding and/or computational techniques. However, for many of those issues, new theories and techniques are needed that extend existing research to handle the requirements of reusability and scope. Much of our work has looked at how to incorporate both new and existing theories and techniques into a general conceptual framework that supports a full range of capabilities. For each specific concept, technique, or strategy that was developed, the conceptual framework required an abstraction of the entity.

There are many example of the tension between specific techniques and conceptual ideas in this research. For example, we developed the negotiated-search strategy. However, within the conceptual framework, negotiated search is simply an instance of a distributed-search strategy that happens to have a wide range of applicability conditions and that was designed based on theories of information-sharing and conflict management. At some point it may be possible to step back even farther and look at a distributed-search strategy as just an instance of a coordination algorithm for distributed problem solving. In other examples, we developed a relaxation algorithm for numeric boundary constraints, but this is simply an instance of a relaxation algorithm that can be applied to a specific type of information. We developed the initiate-solution search operator, but this is simply an instance of an operator that can be applied when certain conditions exist in shared memory. We provided solution acceptability and termination policies, but these are again just instances of a more general concept. Although much work is still needed to formalize the initial theories

and techniques, we have made significant progress in understanding and isolating the issues while also supporting the necessary interplay among elements.

Reusable Agents: Agent reusability pervades every aspect of system-building. A major objective was to understand how to build expert computational agents that: 1) use the most appropriate representations, algorithms, and inference engines for their area of expertise; 2) can cooperate with other agents to satisfy local and/or global goals; 3) can coordinate with other agents to effectively work within a team; and 4) can be developed outside the scope of a particular application system. An important concept is that agents must be highly flexible and able to respond dynamically to situations they encounter. Like Rip van Winkle, an agent may awake at any moment to find itself ensconced in a society of agents that was never anticipated and, further, the agent must be prepared to immediately contribute to the society.

In response to the issues of agent reusability, the internal structure of an agent must provide some capabilities and representations that would not be expected in a non-reusable agent. For example, as described in Chapter 5 conflict among reusable agents is inevitable. Therefore, agents must incorporate explicit techniques to resolve conflicts that occur between local knowledge and externally supplied knowledge. Because it is impossible to predict at agent-design time what system(s) an agent will be embedded in, agents must be flexible about what information may be available from external sources and about how other agents will react to shared information. Information that is intended to be shared must be represented in a format that can easily be assimilated by an external agent. Because an application developer must be able to understand what an agent does, what it requires, and what it can produce, agents must explicitly represent the information required for integrating agents into systems. Because an agent must coordinate its actions with those of other agents, its capabilities must be encapsulated as operators with specific functionality and application conditions. These examples demonstrate that off-the-shelf programs cannot simply be plugged in to an integrating framework and create an effective application system. An agent must be designed (or revised) with reusability in mind, and the required capabilities strongly influence every aspect of the agent's structure.

Development of a Generic Reusable-Agent Framework: One of our primary goals was the development of a generic framework that would support the integration of multiple reusable agents into effective application systems. The TEAM framework fulfills that goal and has proven to be robust, easy-to-use, and flexible. This flexibility is demonstrated by the two application systems that were developed within the framework, the seven-agent STEAM system and the two-agent AGREE system. TEAM is still in a prototype stage and, as with all prototypes, there is much work that could

be done to make it more elegant and efficient. More importantly, there are several significant extensions to the framework that would broaden the scope of problems that can be addressed and improve the processing of problems that are now addressed. These extensions will be discussed below in Section 10.3.

The TEAM framework provides the necessary infrastructure for supporting agent communication and coordination in reusable agent sets. It was developed as an application-independent reusable-agent architecture. It is built on top of GBB, a blackboard-system shell, extended to accommodate multiple persistent agents, explicit conflict management, and mutual solution evaluation. The architecture provides a central communication medium for agents to share both emerging solutions and control information about the state of search such as the strategy being applied and the role each agent plays in that strategy. It isolates the agents from the specifics of accessing and managing shared information while providing efficient storage and retrieval.

TEAM also provides a basic extendible object language for reusable-agent systems including such concepts as solutions, constraints, strategies, and agents. When an application system is built, the language must be extended to domain-level concepts–a solution becomes a design with additional domain attributes for example. The language defined for the application is the shared language used for communication among agents. If any agent does not support the shared language locally, translation mechanisms must be provided to and from the agent's local language (only information that is expected to be shared must be translated).

Implementation of Prototype Application Systems: Two prototype application systems were developed. The first system, STEAM, is a globally cooperative, parametric-design system for steam condensers (see Chapter 6). The second system, AGREE, is a locally cooperative buy/sell negotiation system (see Chapter 7). The two systems are different in their cooperation structure (see the discussion of local vs. global cooperation below), the number and type of agents, the redundancy of agent processing, the determination of mutual acceptability, and the distributed-search strategies they use. The STEAM system was developed concurrently with the TEAM framework. It has proven to be very effective, significantly outperforming its parent system, IRSys (see the discussion of experimental results below). The AGREE system has not been compared to any other system, but does attest to the ease of system-building within TEAM since it was implemented in less than two weeks of design and programming time by a single person. The successful implementation of the two systems demonstrates the versatility and effectiveness of the TEAM framework.

Distributed Search: The capacity of heterogeneous and reusable agents to participate in a distributed-search problem depends upon a combination of local knowledge

(declarative and procedural), local capabilities (local-search operators), and social rules (distributed-search strategies) that bind the agents into a coherent system. Local knowledge and capabilities can be implemented in an agent in whatever way is most suitable for the type of knowledge and the type of agent. However, when building an agent, the social agenda of the agent must be recognized and addressed even though the specifics of required behavior cannot be fully predicted. A distributed-search strategy defines a set of coordinated agent activities (operator applications) that will solve a problem in a specific situation. It can be defined independently from any specific agent set, assuming that the agents in the set will jointly provide appropriate operators. From an agent's local perspective, it defines both the set of operators to be applied and agent-independent preconditions on the application of those operators that will lead to coherent and effective problem solving with other agents. The effectiveness of a particular search strategy depends on characteristics of all of the agents in the agent set and the characteristics of the problem itself. When an agent is built, it isn't possible to predict what those characteristics will be. It follows that agent reusability doesn't come cheaply. Agents must be equipped with operators, representing domain-level expertise, that may never be used. They must know about search strategies, representing coordination expertise, that may never be applicable. Finally, they must know how to select an appropriate strategy, a meta-level control capability, in order to map their knowledge and capabilities to each specific agent set and problem.

Design and Application of Negotiated Search: A specific distributed-search strategy, *negotiated search*, was developed that: a) demonstrates the representation, implementation, and application of strategies within the system; b) supports the incremental construction and integration of component solutions developed by multiple expert agents; c) defines a generic set of search operators to be distributed among agents; and d) explicitly incorporates agent-controlled conflict management to achieve mutually acceptable solutions.

Negotiated search was developed as a default search strategy with wide applicability. It allows flexibility in the range of problems it can solve, characteristics of agents involved, and the amount of information that must be shared among agents. It is a weak search strategy, however, in that it is not highly directed and no guarantees can be made about the quality of solutions that will be found or whether any solution will be found. More powerful strategies can certainly be applied in specific circumstances (see the discussion of customized strategies below) and the TEAM framework supports the selection of those strategies over negotiated search when appropriate. However, with reusable agents, the convergence of all required characteristics for a particular

strategy to be applied cannot be determined at agent-design time. It is necessary to have some weak search strategy that does not have stringent requirements on intra- and interagent characteristics to fall back on, and negotiated search fills that niche.

One goal of the research that was not met with respect to distributed-search strategies was the automatic conversion of a strategy from a paper design to usable code. Strategies are currently specified as transition networks of search operators that describe the coordination-space preconditions of each operator and that provide a well-defined ordering on operator application with respect to a single solution (see Chapter 5). A programmer can map the specifications to a set of operators and implemented precondition functions for each operator. A potential area of future work would be to provide a mechanism to automatically generate the necessary code from a strategy specification.

Customized Search Strategies: There were two goals in the dissertation with respect to customized search strategies: 1) to show that applying a search strategy customized to the application could improve system performance and 2) to show that powerful customized search strategies could be applied within the same framework used to support the more general negotiated-search strategy. We designed and implemented an effective, but highly restricted, strategy called *linear compromise* to demonstrate the feasibility and power of the approach. We also designed several unimplemented strategies in order to demonstrate the feasibility of customization and to illustrate how various strategies have different restrictions on problem characteristics. Through analysis and experimentation, we were able to demonstrate that customization of search strategies can greatly improve system performance. These results are summarized in Section 10.2 below and detailed in Chapter 7. That chapter also describes the mechanisms used by agents in AGREE to support the dynamic selection of search strategies, thereby allowing the system to choose a strategy that is appropriate for the problem and agent set at hand.

An issue in strategy selection that has not fully investigated is to what degree negotiation should be incorporated into the selection process itself. In the current TEAM implementation, a contract manager is given control of strategy selection and will always give top priority to its own preferences. However, negotiating over possible strategy selections can itself be treated as a conflict situation with the same type of iterative search/relaxation methodology that applies to domain problem-solving. As always, a tradeoff exists between the flexibility and possible performance improvement of a negotiation approach and the agent complexity and overhead required to implement the approach.

Development of a Comprehensive Conflict-Management Package: Conflict management includes both conflict avoidance and conflict resolution. Klein has inves-

tigated developing a taxonomy of conflict-management techniques that is domain-independent at its highest levels [Klein, 1991]. At the implementational level, however, instances of conflict-management techniques are domain-specific, depending on the type of information being handled and the capabilities of the agents involved to assimilate and apply the information.

Because agents are reusable, it is not possible to knowledge-engineer an application system. When the agents are designed, it is not known what other information may be available in the completed application. Therefore, it is impossible to remove inter-agent conflicts at that time and, instead, conflict must be considered an inherent and unavoidable part of problem solving. Conflict in cooperative multiagent systems is not attributable to hostile behavior by agents, but instead occurs due to inconsistent or incomplete knowledge, priorities, goals, and evaluation criteria. There are two basic methods for handling conflict among agents: 1) find a solution that avoids the conflict; and 2) manipulate local requirements on solutions until the current solution is no longer considered a conflict.

Conflict avoidance is incorporated into the framework through the communication and assimilation of information among agents. When an agent detects a conflict, if it can articulate its local perspective, it transmits information to other agents about why the conflict occurred. If the other agents can understand and assimilate that information, they will attempt to find other solutions that avoid the problem. We simplified our investigation of conflict avoidance by restricting the information that could be shared among agents to simple numeric boundary constraints. This restriction allowed us to develop domain-independent algorithms for information assimilation and retrieval that worked consistently on either local or externally supplied knowledge and that handled inconsistent assimilated knowledge by explicitly selecting some subset of information to apply. Each type of shared information requires customized assimilation and retrieval mechanisms. Therefore, although the restriction hampers agents to some extent by not taking full advantage of potential information sharing, it is justifiable in reusable-agent systems because it limits the amount of overhead that must be built into each agent.

Conflict avoidance doesn't always work, either because no solution exists that will not cause some conflict or because agents are not capable of articulating the necessary information to focus the search. It is sometimes, though not always, possible to determine which case is responsible for a lack of problem-solving progress. Conflict-resolution techniques that manipulate local solution requirements can be applied in either case. Usually conflict resolution is applied as a secondary option after avoidance, since manipulation of solution requirements inevitably reduces some

agent's (or agents') satisfaction with a solution. Three primary forms of requirement manipulation were identified, *unilateral relaxation*, *responsive relaxation*, and *automatic relaxation*, along with the situations in which each form should be applied.

Although the details of conflict handling change from system to system and expert to expert, the negotiated-search strategy described above provides a conceptual foundation for integrating conflict management into a general multiagent search. The feasibility of the conceptual approach was validated through the development of a package of specific conflict-management techniques used within the **STEAM** and **AGREE** systems. In contrast to the general techniques used in negotiated search, the linear-compromise strategy used in **AGREE** provides an example of a highly focused conflict-management approach with restricted applicability. The integration of conflict-management techniques from both negotiated search and linear compromise in **AGREE** illustrates the ease with which conflict management can be tailored to the application system within the **TEAM** framework.

The only conflict-management goal that was not fully realized in the dissertation research was embedding dynamically selected conflict-resolution strategies into general distributed-search strategies. This is discussed below in Section 10.3 with respect to extensions to **TEAM**. A full description of the problem and an initial specification of an extended version of negotiated search to address it are described in Chapter 5.

Investigation of Agent/Role Assignments: The role of an agent in a distributed-search strategy defines the task(s) that the agent is responsible for achieving as a participant in that strategy. For systems built within the **TEAM** framework, an agent's role in problem solving is realized through instantiation of a set of local operators in response to a role assignment. The effect of different role assignments on system performance and characteristics of agents that influence their appropriateness for a particular role are detailed in Chapter 9 and are summarized below in Section 10.2. These results support our early intuition that role assignment affects system performance. We identified several characteristics of agents that affect the suitability of role assignments. However, the analysis done in this dissertation is just a beginning. It is clear that there are many complex interacting factors that influence role assignment in reusable-agent systems. We have substantiated that the problem exists; solving the problem is left to future work.

We see two ways to approach this problem of selecting good agent/role assignments. The first is to try to manually decompose and analyze the various factors that affect the suitability of an agent for a role within a given agent set and the interactions of those factors. The second is to take a more probabilistic approach and make choices based on what seems to work in a particular situation rather than on

complete understanding of how it works. In Section 10.3, we will discuss potential future work on applying machine-learning mechanisms to the problem of determining initial role assignments for agents and to dynamic adjustment of role assignments based on continuous feedback. This learning takes two forms: 1) learning information to be used within the current application system; and 2) learning information to be used across the boundaries of application systems.

Cooperation Structure: Systems of agents investigated in this dissertation have two primary cooperation structures: *local cooperation* and *global cooperation.* In local cooperation, agents have separate local goals and interact only to further those goals. An example of this is a consumer/producer system in which some agents need to acquire resources and other agents have resources available to sell. The agents interact occasionally and serendipitously when external assistance will further their local goals. Locally cooperative agents may also be competitive since the satisfaction one agent achieves with a solution may be inversely related to the satisfaction another agent achieves. The agents are willing to search for a mutually acceptable solution because it is possible that both agents can achieve local goals through cooperation, however, there is no guarantee that they will find a mutually acceptable solution. The AGREE system described in Chapter 7 has this type of cooperation structure.

Another type of cooperation occurs when agents have a global goal to fulfill that overrides their local goals when necessary. In *global cooperation*, interaction is an integral part of each agent's agenda. Each agent takes responsibility for some part of an overall solution and has local evaluation criteria for that part. The complete solution, however, is evaluated based on its composite attributes, for example, the total cost of a design, rather than on the attributes of individual components. Agents still have local goals that guide their individual search processes and these local goals may still be competitive. For example, each agent may have the local goal of developing a minimum-cost component. However, when the agents attempt to integrate these minimum-cost components it may be found that the components cannot be merged due to inconsistent requirements. One or more of the agents will have to produce a locally higher-cost component in order to find a globally consistent solution. With global cooperation, however, the utility of a particular compromise can be evaluated by its effect on the global solution, whereas in local cooperation, the utility of any compromise is evaluated individually by each agent and may be different for each agent. The STEAM system described in Chapter 6 has a globally cooperative structure.

The type of cooperation embodied in a system affects the selection of a search strategy and, possibly, what information should be shared by the agents in the system.

In competitive situations, agents may not be willing to share their "bottom-line" acceptability thresholds for solutions because the motivation to satisfy local evaluation criteria is much stronger than that of satisfying external criteria.

Determining the Mutual Acceptability of Solutions: Determining mutual acceptability of solutions has two components: agent-level evaluation and optional framework-level evaluation. At the agent level, each agent must be able to evaluate a solution from its own perspective and determine whether or not it is willing to accept the solution. For locally cooperative systems, mutual acceptability is determined solely by local acceptability. It is enough that all parties to the transaction are willing to accept the proposed solution. However, in globally cooperative situations, no one agent may have enough global expertise to reasonably evaluate a complete solution. Evaluation at this level can be done by an expert agent, designed specifically to evaluate complete solutions. Alternatively, it can be done through application of a utility function defined in the TEAM framework and applied to completed solutions. Usually, in globally cooperative systems, mutual acceptability is defined as some combination of agent acceptability and global evaluation. The mechanisms for collecting and combining agent and evaluations were described in Chapter 4.

Termination Policies: In satisficing situations where there is no absolute scale of correctness or "goodness" by which to measure potential solutions (except relative to other known solutions), it is very difficult to know when to stop looking for a better solution. Different policies are suitable for different situations. With reusable agents, the best time to choose a particular termination policy is when an application system is put together. Not enough is known about the problem, agent set, and time/quality tradeoff requirements at agent-design time. The TEAM framework was designed to allow maximum flexibility in specifying these policies by providing system variables and 'hooks' for policy definition. A default policy was provided for the STEAM system, but it can be easily modified or replaced.

10.2 Experimental Results

The feasibility and effectiveness of the framework for building and integrating systems of heterogeneous and reusable agents were tested through the implementation of two prototype application systems, the globally cooperative STEAM system and the locally cooperative AGREE system. These two systems were then used as testbeds for various experiments.

The STEAM system was developed using domain-level code taken from an iterative respecification system, IRSys, developed by Meunier and Dixon in the Mechanical Engineering Department at the University of Massachusetts (see Chapter 6). In order to

evaluate the effectiveness of the system, performance in STEAM was directly compared to the performance of IRSys by applying measures of solution quality and processing time. Details of those experiments are contained in Chapter 6. STEAM substantively outperformed IRSys in terms of solution quality (12.81% improvement), the number of component proposals generated (58.82% improvement), and the amount of time it took to solve a problem (52.32% reduction in time). Because STEAM is a complex system that exercises many of the features of both the conceptual and architectural frameworks developed for this dissertation, we believe the performance of STEAM is an apt indicator of the potential of the general approach.

There are several reasons for the effectiveness of the STEAM system as compared to IRSys. The underlying search strategy used by STEAM, *negotiated search*, is well-suited to the domain, perhaps more so than the implicit single-path backtracking strategy used by IRSys. In STEAM, solutions are initiated by multiple agents, leading to diverse starting points for designs and broad coverage of the composite solution space. This characteristic of STEAM makes it more likely that the 'best' composite solutions will be found, rather than the 'best' solutions from one agent's local perspective. A second reason that STEAM outperformed IRSys is that STEAM's information exchange, assimilation, and conflict avoidance is more comprehensive than that of IRSys. Agents focus more quickly and with better information on areas of the composite solution space that are likely to be mutually acceptable. The positive results from these experiments indicate that negotiated search is an effective strategy and that the STEAM system is highly competitive.

A second set of experiments was performed to investigate the efficiency of highly focused distributed-search strategies and to compare their performance with more general strategies. These experiments were done in the two-agent, locally cooperative, AGREE system. Our results substantiate the claim that customizing search strategies can strongly affect the performance of a system. However, they also point out that the appropriateness of a search strategy is tied very strongly to characteristics of the problem and agent set that cannot be determined at agent-design time. This leads us to the conclusion that reusable-agent systems require dynamic selection capabilities for search algorithms and also that reusable agents need to be flexibly built to participate in multiple algorithms. Although we have assumed agents need to have explicit knowledge about the existence of potential search strategies, this assumption is quite restrictive. As we look at possible future research, one promising area to explore is whether agents can learn to participate in strategies that were not explicitly anticipated at the time the agent was designed. To take this one step

further, it might be possible for agents in an agent set to cooperatively design new strategies to fit the situation.

Information exchange and assimilation is a dominant factor in conflict management because it allows an agent to understand the external forces that will determine the eventual acceptability of any proposed solutions. In Chapter 8, we described experiments in which the ability of agents to assimilate external information was controlled and the resulting system performance was compared. This set of experiments showed that when agents exchange and assimilate information about the composite search space, solution quality improves and the time required for processing is reduced. Information sharing assists agents in focusing in areas of the composite search space that are likely to be mutually acceptable. This leads to higher productivity for the agent and, ultimately, improved system performance. Performance improvements in the STEAM system when assimilation capabilities were enabled included a 5.72% improvement in solution quality, a 8.06% improvement in system runtime, and a 40.62% improvement in the runtime-per-solution measure as described in Chapter 8. These results demonstrated that information exchange and assimilation do affect both solution quality and processing time. In fact, the effect of information exchange and assimilation is strong enough that it should be considered an important factor in the design of any multiagent system.

Although these performance improvements were substantial, the costs associated with information sharing can be quite high and may offset the productivity enhancements in some situations. This is particularly true with reusable agents because they cannot rely on receiving particular types of information from external sources. Therefore, they must be highly flexible in their assimilation capabilities, leading to high cost in agent design and potentially high cost in performance due to a lack of specificity. At agent-design time, the agent must be provided with mechanisms that allow it to articulate important information to other agents and effectively assimilate external information. Assimilating external information requires that the agent be able to 1) resolve any conflicts that may arise from combining the external information with local knowledge and 2) apply the assimilated knowledge to its search process appropriately. Since every type of information that is assimilated may require specific conflict-resolution algorithms and specific application mechanisms, the overhead of using information must be balanced with the potential of that information for improving the productivity of the system.

One goal of the dissertation was to classify the types of costs that occur in information sharing and estimate their effect on overall processing. We focused primarily on costs as measured by processing time. We identified five primary categories of

costs that relate to information sharing: constraint generation (putting constraining information into a sharable declarative form); transmission determination (determination of what to transmit at a given time); transmission (the actual physical transfer of information); translation (translating information from a local to shared language and vice versa); and local management (local storage and retrieval, resolution of conflicting external and local information). In Chapter 8, experiments were detailed that determined how expensive these operations were within the context of the STEAM system. The total amount of time spent in information-sharing activities in the STEAM trials with assimilation enabled was approximately 10.17%. Although it is expected that the costs of the operations will vary to some degree from system to system, the results from these experiments provide approximate guidelines that can assist in determining the potential gain or loss in processing time that can be attributed to information exchange and assimilation.

The final set of experiments, presented in Chapter 9, investigated how the assignment of particular roles to particular agents affected system performance. The experiments substantiated that performance can be improved through judicious selection of role assignments. We observed an average 5.51% improvement in solution quality and an average 9.77% improvement in runtime-per-solution when moving from a poor selection of role assignments to a good selection. We found that some agents are inherently better suited for some roles than others but, at the same time, it is necessary to consider the relative suitability of an agent with respect to other agents in the system. An analysis of a generic two-agent problem was used to show that one factor that can affect an agent's suitability for a role is the precision of the information it is able to communicate to other agents.

10.3 Future Work

There are two primary areas of future work to pursue: the first, a practical approach to improving and solidifying the work that has already been done, and the second, a freewheeling approach to expanding the horizons of building reusable-agent systems. The work that has been presented represents significant progress toward enabling reusable-agent technology. Using the framework and agent-level mechanisms described, it is possible to build a reusable agent and to integrate an agent set into an application system. There were a number of restricting assumptions made, however, and the future of the technology involves loosening those assumptions. We first describe improvements and extensions that would make the current framework more flexible and effective and then move on to the more long-term and unrestrained territory of where to go from here.

10.3.1 Improvements and Extensions

In this section, we describe extensions to the TEAM architecture, the agent specifications, and the underlying conceptual framework that can be realized in the near future. These extensions improve upon what has already been done without jumping into new terrain.

Distributing the Framework: The current implementation of TEAM uses a centralized shared memory to store and communicate emerging solutions. This is appropriate for the problems that have been investigated to date. However, we anticipate future work will involve applications in domains that would benefit from a fully distributed architecture. A major extension of the framework and of the functional specification for agents would involve designing and implementing a physically distributed version over multiple processes or machines.

Embedding Focused Conflict Resolution: A second area for future research is in embedding focused conflict resolution into the *negotiated-search strategy* described in Chapter 5. In that chapter, Figure 5.10 illustrates an extension to the current version of negotiated search. This extension would allow a subset of the agent set to temporarily suspend their normal roles within the strategy and instead apply a specific conflict-resolution algorithm to a detected conflict. For example, the *linear-compromise strategy* described in Chapter 7 could be jointly applied by a buyer and seller to determine a fair price for a house within the more general negotiated search. Other decisions, such as what furnishings or appliances might be included in the transaction, cannot be addressed through linear compromise because of its limited applicability. Those decisions could be determined through the more general negotiated search. The system currently must choose one strategy to apply to the entire problem. If any part of the problem does not meet the restrictions of a strategy, that strategy cannot be applied at all.

Adding Optimization to Negotiated Search: An optional third phase could be added to the negotiated-search algorithm to optimize solutions that were developed in previous stages. Specifically, a distributed neighborhood search could be implemented in which agents would attempt to generate solutions that define the local maxima of the composite space around existing solutions. This would require deciding which parameters would be best suited to guiding the global search (which parameters are most likely to effect changes in a solution's rating), and giving either some agent or the framework the functionality to generate the anchor points for potential solutions.

Expanding the Range of Search and Conflict-Resolution Strategies: Currently, only two search strategies are implemented: *negotiated search* and *linear compromise*. Other strategies, such as those suggested by Khedro [Khedro and Genesereth, 1993]

and Klein [Klein, 1991], could be implemented in TEAM. When a strategy has been implemented in another system, it would be possible to compare performance with the TEAM version, leading to a better understanding of the strengths and weaknesses of the framework. For strategies that have designed but not implemented, it would be possible to determine the feasibility and effectiveness of the strategy while expanding the range of problem solving possible in TEAM. A secondary goal in expanding the set of strategies available is to develop a deeper understanding of the mappings between problem-solving situations and applicable strategies, both from an analytical perspective and from a practical one.

Using Default-Assumption Reasoning: It is often the case that agents must execute sequentially due to domain-level input/output interactions (one agent uses the output from another agent as an input). There are situations where this sequentiality is undesirable because it leads to a depth-first type of search that can be very costly when a deep branch of the search fails. For example, in concurrent engineering applications, each state in the search tree may involve considerable time and resource commitments. An extension to the negotiated-search strategy has been developed that would enable more concurrency among agents through the use of default reasoning. In default reasoning, agents use a 'guess' about possible values of input variables to execute immediately rather than waiting for the real values to become available. The extended strategy, shown in Figure 5.11, has been implemented but has not yet been tested. In future work, this strategy can be evaluated and various methods of estimating input values can be compared.

Adding Feedback-Based Reasoning to Automatic Relaxation: Automatic relaxation is a form of requirement relaxation that is used to ensure that progress is made on a problem when there is no explicit information about conflict in the agent set, but no solutions are being found. It is, therefore, response to a lack of problem-solving progress, as described in Chapter 5. The current realization of automatic relaxation forces each agent to relax requirements after some number of processing cycles. The number is determined by the value of a user-specified timing parameter. The original concept, however, was to use explicit information about the progress of problem solving rather than a timing parameter to determine when to relax the requirements. For example, an agent might monitor the solutions it initiates and use information about their acceptability to other agents and their global acceptability to decide if it is being too tight in its own requirements. Alternatively, an agent might relax a requirement if its last n proposals were rejected by other agents. It is expected that adding more sophisticated reasoning capabilities to the application of this form of

relaxation would improve solution quality by not forcing agents to relax requirements too quickly.

10.3.2 New Horizons

In the work done to date, we have made a number of limiting assumptions in order to restrict the work to a manageable scope. However, as pieces of the problem become better understood, it will be possible to expand the goals of the technology. In this section, we speculate on some of the more long-term research that is needed to create truly flexible and generic reusable agents. This work will focus on increasing the capacity of agents to flexibly adapt to the situations in which they are placed. Agents will need sophisticated built-in reasoning capabilities and they will need to be able to *learn* how to best interact within a particular agent set. Perhaps even further along the path, agents will be able to learn across agent sets, learning a new strategy from agents in one set and teaching that strategy to agents in the next agent set for example.

Agents cannot always depend upon being embedded in a benevolent, cooperative agent set. For example, one highly plausible use for a reusable agent would be to act as a sales agent for a parts supplier. Companies needing the parts would interact directly with the agent to order stock, perhaps through reusable inventory-control agents. This is an extremely competitive situation and the incentive to cheat is high. One deceptive agent acting in a network of honest agents could benefit greatly. Even if all agents were deceitful, the most devious could still have an advantage. Some of the ways to cheat would be to provide false information about prices, availability, or quality of products.

Ephrati and Rosenschein described the Clarke Tax, a voting protocol that could be applied over an entire system of agents to allow the system to reach consensus without providing an advantage to agents that lie [Ephrati and Rosenschein, 1991]. However, this protocol, like other market-based protocols, is vulnerable to the formation of coalitions, and, therefore, provides only a first layer of protection. From the perspective of an individual agent, the best protection is to provide the agent with the sophistication to recognize the potential for dishonesty and to adjust its information sharing accordingly. For example, an agent that is not sure of the integrity of other agents should not believe everything it learns from external sources and should be less forthcoming about its own bottom-line criteria.

Learning Agent/Role Assignments: We showed in the experiments in Chapter 9 that assignment of roles to agents affects solution quality and processing time in an application system. We identified some characteristics of agents that are important

in determining how well an agent will be able to perform a particular role. However, there is no comprehensive list of characteristics that affect assignments and, even for those that are known, the interactions between various characteristics are not well understood. This makes it difficult for an application-system developer or for the agent set itself to specify appropriate role assignments. One approach to solving this problem is to apply machine-learning techniques during execution of the application system to dynamically adjust role assignments. Initially, the system would be set up to accommodate all possible agent/role assignments (each feasible assignment would be applied to some subset of solutions generated by the system). Each agent would use feedback about the merit of a solution and the role it played in developing that solution to incrementally adjust its activity to favor those roles that led to good solutions. This form of learning can be used both to improve initial role assignments over the life of an application system and to adjust to a changing situation during a single execution. However, it might be possible to extend the value of learned information across the boundaries of application systems if an agent could learn to recognize and classify features of generic situations. Initial work in this direction is now being done by Maram V. Nagendra Prasad within the TEAM project.

Learning Search Strategies From Other Agents and Creating New Strategies: Supplying every agent with every possible strategy at agent-design time is unrealistic since many strategies are quite limited in scope and may never be used. However, in order to apply a strategy, every agent must understand what role it is to play within the strategy. It should be possible for an agent that knows a strategy to transmit the strategy specification to other agents. Assuming that a receiving agent has the requisite capabilities to participate in the strategy, it could then become a permanent part of the agent's repertoire of strategies. However, before this capability can be realized, a general strategy-specification language must be incorporated into the TEAM language and agents must be given the ability to interpret that language. Currently, strategies are specified as transition networks offline and then explicitly implemented into an agent's operators and operator preconditions. It would be necessary to provide structures and functionality to isolate and use strategy information in a declarative form. Once the appropriate mechanisms were in place, however, it might be possible for an agent set to create new strategies through experimentation and feedback with operator invocations.

Although many of the system extensions described in this section have a pie-in-the-sky flavor, the fact that it is possible to vaguely describe how they might become reality has exciting implications for the future of reusable-agent technology. Until agents can move beyond what is explicitly provided to them by their programmers,

reusability will remain limited in scope. But the achievement of these capabilities does not seem impossible. In fact, in looking back over what has been already been achieved, the extended capabilities seem well within reach. The initial architectural foundation, along with beginning theories of distributed search, conflict management, information exchange, and role assignment provide a basis upon which to build. Work from related fields such as machine learning, operations research, organizational science, and management science provide many techniques that can be utilized. Hopefully, the next few years will bring significant progress toward agents that are independent, self-sufficient, and adaptable, both from a theoretical perspective and in fielded applications.

APPENDIX A

A SYSTEM TRACE

In this appendix, we provide an example of a system trace from the **STEAM** system. There are seven agents in the complete **STEAM** system, as will be described in Chapter 6. However, agents can be trivially added to and deleted from the system. For this example, our purpose is to show how a distributed-search strategy defines the role of an agent in problem-solving and coordinates agent activities. Therefore, we use only two agents, each of which can both initiate and extend designs. (In Chapter 5, Section 5.2, a highly detailed trace of a run in the complete seven-agent example is given.)

In this two-agent example, the agents are required to find three mutually acceptable designs, where a mutually acceptable design is defined as a complete design that is judged to be acceptable by both agents. In the course of problem-solving, each agent initiates new designs and extends designs initiated by the other agent. When extending another agent's design, an agent sometimes finds a conflict with a previously assigned attribute value of the design. The agent extending the design cannot change the existing value, but must instead try to find the best local solution possible that is consistent with that value. If the best local solution violates a local solution requirement (a soft preference or hard constraint), a conflict has been detected, and a message is sent to other agent describing the conflict. Agents receiving the message will attempt to assimilate that information into their local knowledge bases and use it to guide future processing.

The trace alternates between agent activation cycles and framework ks activation cycles. Since we have not yet introduced the **TEAM** framework, we will not discuss the details of these cycles here. **TEAM** will be described in Chapter 4. Briefly, the agents execute during the agent activation cycles and their results are integrated into the shared memory during the framework ks activation cycles.

TRACE OF A TWO-AGENT EXECUTION OF STEAM

Each agent plays two roles: design initiation *and* design extension.

```
LISP: (run-system)
```

```
SYSTEM-INITIALIZATION --> Control Shell Invocation

 STARTING THE FRAMEWORK KS ACTIVATION CYCLE

KS-INVOCATION ---------> ksa.00051 initial nil

  ;;; The system is initialized by inputing a new problem
  ;;; specification.  This specification is placed in
  ;;; shared memory.

PROBLEM-SPECIFICATION-CREATION -->
  Platform Side Length: 120
  Required Capacity: 500
  Maximum Platform Deflection: 0.01
QUIESCENCE-EVENT ------> Scheduling Queue

==============================================================

 STARTING AN AGENT ACTIVATION CYCLE

##############################################################

;;;  Running the pump designer ...

==============================================================

KS-INVOCATION ---------> ksi.00018 pump-system-initial-ks
KS-INVOCATION ---------> ksi.00019 initiate-solution
   ;;; The pump designer runs and creates a new pump
   ;;; design, PUMP-1, that will be the basis for a new
   ;;; steam-condenser design.
==============================================================

##############################################################

;;;  Running the heat exchanger designer ...

==============================================================

KS-INVOCATION ---------> ksi.00019 he-system-initial-ks
KS-INVOCATION ---------> ksi.00020 initiate-solution
   ;;; The heat-exchanger designer runs and creates a new
   ;;; heat-exchanger design, HEATX-1, that will be
   ;;; the basis for a new steam-condenser design.
==============================================================
##############################################################

 STARTING THE FRAMEWORK KS ACTIVATION CYCLE

KS-INVOCATION ---------> ksa.00052 build-initiated-design pump-1
   ;;; A new design, DESIGN_1, is built in shared memory from the
   ;;; pump proposal, PUMP-1.
KS-INVOCATION ---------> ksa.00053 build-initiated-design heatx-1
   ;;; A new design, DESIGN_2, is built in shared memory from the
```

```
   ;;; heat-exchanger proposal, HEATX-1.
KS-INVOCATION ---------> ksa.00054 update-solution-features design_1
   ;;; Features of DESIGN_1 are updated.
KS-INVOCATION ---------> ksa.00055 update-completeness
   ;;; DESIGN_1 is checked for completion, but it only has one
   ;;; component so far, PUMP-1.
KS-INVOCATION ---------> ksa.00056 update-solution-features design_2
   ;;; Features of DESIGN_2 are updated.
KS-INVOCATION ---------> ksa.00057 update-completeness
   ;;; DESIGN_2 is checked for completion, but it only has one
   ;;; component so far, HEATX-1.
QUIESCENCE-EVENT ------> Scheduling Queue
   ;;; There is nothing else to do with the current designs
   ;;; so execute the agents again.
==============================================================

 STARTING AN AGENT ACTIVATION CYCLE

##############################################################

;;;  Running the pump designer ...

==============================================================
KS-INVOCATION ---------> ksi.00021 extend-solution design-2
   ;;; The pump designer runs and creates a pump design,
   ;;; PUMP-2, that extends and is consistent with DESIGN_2.

==============================================================
##############################################################

;;;  Running the heat exchanger designer ...

==============================================================
KS-INVOCATION ---------> ksi.00022 extend-solution design_1
   ;;; The heat-exchanger designer runs and creates a
   ;;; heat-exchanger design, HEATX-2, that extends
   ;;; and is consistent with DESIGN_1.

==============================================================
##############################################################

 STARTING THE FRAMEWORK KS ACTIVATION CYCLE

KS-INVOCATION ---------> ksa.00058 extend-design pump-2 design-2
   ;;; PUMP-2 is added to DESIGN_2 in shared memory.
KS-INVOCATION ---------> ksa.00059 extend-design heatx-2 design-1
   ;;; HEATX-2 is added to DESIGN_1 in
   ;;; shared memory.
KS-INVOCATION ---------> ksa.00060 update-completeness design_1
   ;;; DESIGN_1 is checked and found to be complete with
   ;;; PUMP-1 and HEATX-2.
KS-INVOCATION ---------> ksa.00061 update-solution-features  design_1
   ;;; Features of DESIGN_1 are updated.
KS-INVOCATION ---------> ksa.00062 calculate-global-utility design_1
```

```
   ;;; The global utility function is applied to DESIGN_1.
KS-INVOCATION ---------> ksa.00063 update-completeness design_2
   ;;; DESIGN_2 is checked and found to be complete with
   ;;; HEATX-1 and PUMP-2.
KS-INVOCATION ---------> ksa.00065 update-solution-features design_2
   ;;; Features of DESIGN_2 are updated.
KS-INVOCATION ---------> ksa.00064 calculate-global-utility design_2
   ;;; The global utility function is applied to DESIGN_2.
QUIESCENCE-EVENT ------> Scheduling Queue
   ;;; There is nothing else to do with the current designs
   ;;; so execute the agents again.
============================================================

 STARTING AN AGENT ACTIVATION CYCLE

############################################################

;;;  Running the pump designer ...

============================================================

KS-INVOCATION ---------> ksi.00023 store-received-information
  ;;; The heat-exchanger designer sent information to the
  ;;; pump designer about a conflict it detected when trying to
  ;;; extend DESIGN_1.  This conflict information is stored
  ;;; by the pump designer to guide its search in generating
  ;;; new solutions.
KS-INVOCATION ---------> ksi.00024 initiate-solution
   ;;; The pump designer runs and creates a new pump design,
   ;;; PUMP-3, that will be the basis for a new steam-condenser
   ;;; design.
============================================================
############################################################

;;;  Running the heat exchanger designer ...

============================================================

KS-INVOCATION ---------> ksi.00024 store-received-information
  ;;; The pump designer sent information to the heat-exchanger
  ;;; designer about a conflict it detected when trying to
  ;;; extend DESIGN_2.  This conflict information is stored
  ;;; by the heat-exchanger designer to guide its search in
  ;;; generating new solutions.
KS-INVOCATION ---------> ksi.00025 initiate-solution
   ;;; The heat-exchanger designer runs and creates a new
   ;;; heat-exchanger design, HEATX-3, that will be
   ;;; the basis for a new steam-condenser design.
============================================================
############################################################

 STARTING THE FRAMEWORK KS ACTIVATION CYCLE

KS-INVOCATION ---------> ksa.00068 build-initiated-design pump-3
```

```
   ;;; A new design, DESIGN_3, is built in shared memory from the
   ;;; pump proposal, PUMP-3.
KS-INVOCATION ---------> ksa.00069 build-initiated-design heatx-3
   ;;; A new design, DESIGN_4, is built in shared memory from the
   ;;; heat-exchanger proposal, HEATX-3.
KS-INVOCATION ---------> ksa.00070 update-solution-features design_3
   ;;; Features of DESIGN_3 are updated
KS-INVOCATION ---------> ksa.00071 update-completeness design_3
   ;;; DESIGN_3 is checked and found to be incomplete with only
   ;;; a pump component, PUMP-3.
KS-INVOCATION ---------> ksa.00072 update-solution-features design_4
   ;;; Features of DESIGN_4 are updated.
KS-INVOCATION ---------> ksa.00073 update-completeness design_4
   ;;; DESIGN_4 is checked and found to be incomplete with only
   ;;; a heat-exchanger component, HEATX-3.
QUIESCENCE-EVENT ------> Scheduling Queue
   ;;; There is nothing else to do with the current designs
   ;;; so execute the agents again.
================================================================

 STARTING AN AGENT ACTIVATION CYCLE

################################################################

;;;  Running the pump designer ...

================================================================
KS-INVOCATION ---------> ksi.00026 extend-solution design_4
   ;;; The pump designer runs and creates a pump design,
   ;;; PUMP-4, that extends and is consistent with DESIGN_4.
================================================================
################################################################

;;;  Running the heat exchanger designer ...

================================================================
KS-INVOCATION ---------> ksi.00027 extend-solution design_3
   ;;; The heat-exchanger designer runs and creates a
   ;;; heat-exchanger design, HEATX-4, that extends
   ;;; and is consistent with DESIGN_3.
================================================================
################################################################

 STARTING THE FRAMEWORK KS ACTIVATION CYCLE

KS-INVOCATION ---------> ksa.00074 extend-design pump-4 design-4
   ;;; PUMP-4 is added to DESIGN_4 in shared memory.
KS-INVOCATION ---------> ksa.00075 extend-design heatx-4 design-3
   ;;; HEATX-4 is added to DESIGN_3 in shared memory.
KS-INVOCATION ---------> ksa.00076 update-completeness design-3
   ;;; DESIGN_3 is checked and found to be complete with
   ;;; PUMP-3 and HEATX-4.
KS-INVOCATION ---------> ksa.00077 update-solution-features design-3
   ;;; Features of DESIGN_3 are updated.
```

```
KS-INVOCATION ---------> ksa.00078 calculate-global-utility design-3
   ;;; The global utility function is applied to DESIGN_3.
KS-INVOCATION ---------> ksa.00079 update-completeness design-4
   ;;; DESIGN_4 is checked and found to be complete with
   ;;; HEATX-3 and PUMP-4.
KS-INVOCATION ---------> ksa.00081 update-solution-features design-4
   ;;; Features of DESIGN_4 are updated.
KS-INVOCATION ---------> ksa.00080 calculate-global-utility design-4
   ;;; The global utility function is applied to DESIGN_4.
QUIESCENCE-EVENT ------> Scheduling Queue
   ;;; There is nothing else to do with the current designs
   ;;; so execute the agents again.
==============================================================

 STARTING AN AGENT ACTIVATION CYCLE

##############################################################

;;;  Running the pump designer ...

==============================================================
KS-INVOCATION ---------> ksi.00028 store-received-information
  ;;; The heat-exchanger designer sent information to the
  ;;; pump designer about a conflict it detected when trying to
  ;;; extend DESIGN_3.  This conflict information is stored
  ;;; by the pump designer to guide its search in generating
  ;;; new solutions.
KS-INVOCATION ---------> ksi.00029 initiate-solution
   ;;; The pump designer runs and creates a new pump design,
   ;;; PUMP-5, that will be the basis for a new steam-condenser
   ;;; design.
==============================================================
##############################################################

;;;  Running the heat exchanger designer ...

==============================================================
KS-INVOCATION ---------> ksi.00029 store-received-information
  ;;; The pump designer sent information to the heat-exchanger
  ;;; designer about a conflict it detected when trying to
  ;;; extend DESIGN_4.  This conflict information is stored
  ;;; by the heat-exchanger designer to guide its search in
  ;;; generating new solutions.
KS-INVOCATION ---------> ksi.00030 initiate-solution
   ;;; The heat-exchanger designer runs and creates a new
   ;;; heat-exchanger design, HEATX-5, that will be
   ;;; the basis for a new steam-condenser design.
==============================================================
##############################################################

 STARTING THE FRAMEWORK KS ACTIVATION CYCLE

KS-INVOCATION ---------> ksa.00084 build-initiated-design pump-5
   ;;; A new design, DESIGN_5, is built in shared memory from the
```

```
   ;;; pump proposal, PUMP-5.
KS-INVOCATION ---------> ksa.00085 build-initiated-design heatx-5
   ;;; A new design, DESIGN_6, is built in shared memory from the
   ;;; heat-exchanger proposal, HEATX-5.
KS-INVOCATION ---------> ksa.00086 update-solution-features design-5
   ;;; Features of DESIGN_5 are updated
KS-INVOCATION ---------> ksa.00087 update-completeness design-5
   ;;; DESIGN_5 is checked and found to be incomplete with only
   ;;; a pump component, PUMP-5.
KS-INVOCATION ---------> ksa.00088 update-solution-features design-6
   ;;; Features of DESIGN_6 are updated
KS-INVOCATION ---------> ksa.00089 update-completeness design-6
   ;;; DESIGN_6 is checked and found to be incomplete with only
   ;;; a heat-exchanger component, HEATX-5.
QUIESCENCE-EVENT ------> Scheduling Queue
   ;;; There is nothing else to do with the current designs
   ;;; so execute the agents again.
================================================================

 STARTING AN AGENT ACTIVATION CYCLE

################################################################

;;;  Running the pump designer ...

================================================================
KS-INVOCATION ---------> ksi.00032 relax-solution-requirement
  ;;; The pump designer relaxes a solution requirement
KS-INVOCATION ---------> ksi.00031 extend-solution design-6
   ;;; The pump designer runs and creates a pump design,
   ;;; PUMP-6, that extends and is consistent with DESIGN_6.
================================================================
################################################################

;;;  Running the heat exchanger designer ...

================================================================
KS-INVOCATION ---------> ksi.00033 relax-solution-requirement
   ;;; The heat-exchanger designer relaxes a solution requirement
KS-INVOCATION ---------> ksi.00032 extend-solution design-5
   ;;; The heat-exchanger designer runs and creates a
   ;;; heat-exchanger design, HEATX-6, that extends
   ;;; and is consistent with DESIGN_5.
================================================================
################################################################

 STARTING THE FRAMEWORK KS ACTIVATION CYCLE

KS-INVOCATION ---------> ksa.00090 extend-design pump-6 design-6
   ;;; PUMP-6 is added to DESIGN_6 in shared memory.
KS-INVOCATION ---------> ksa.00091 extend-design heatx-6 design-5
   ;;; HEATX-6 is added to DESIGN_5 in shared memory.
KS-INVOCATION ---------> ksa.00094 update-solution-features design-5
   ;;; Features of DESIGN_5 are updated
```

```
KS-INVOCATION ---------> ksa.00092 update-completeness design-5
   ;;; DESIGN_5 is checked and found to be complete with
   ;;; PUMP-5 and HEATX-6.
KS-INVOCATION ---------> ksa.00093 calculate-global-utility design-5
   ;;; The global utility function is applied to DESIGN_5.
KS-INVOCATION ---------> ksa.00096 update-solution-features design-6
   ;;; Features of DESIGN_6 are updated
KS-INVOCATION ---------> ksa.00095 update-completeness design-6
   ;;; DESIGN_6 is checked and found to be complete with
   ;;; HEATX-5 and PUMP-6.
KS-INVOCATION ---------> ksa.00097 calculate-global-utility design-6
   ;;; The global utility function is applied to DESIGN_6.
KS-INVOCATION ---------> ksa.00100 terminate-search
   ;;; At this point in processing, the goal of finding 3 mutually
   ;;; acceptable solutions is satisfied in the system.  This event
   ;;; triggers the system to terminate the search and return the
   ;;; designs that were found.
QUIESCENCE-EVENT ------> Scheduling Queue
============================================================
############################################################

***** Explicit STOP returned from *control-shell-exit-hook* *****
```

A simplified view of one of the three solutions developed by the system is shown in Figure A.1. The language used to specify objects such as solutions is described in detail in Chapter 4.

Design solutions are explained in more detail in Chapter 6 during our discussion of the STEAM system. The design shown in Figure A.1 has been greatly simplified to show a minimal set of attributes. Many of the actual attributes will be introduced in the context of particular mechanisms for distributed search such as determining the acceptability of a solution or detecting and resolving conflicts. Here we are primarily concerned with showing how agents play different roles in the development of a design. Figure A.1 shows a design that was initiated by the heat-exchanger designer and extended by the pump designer.

```
Design_2
  acceptability                :acceptable
  completeness                 :complete
  agents                       (:pump-agent :heat-exchanger-agent)
  available-head               310.02
  minimum-head                 275.39
  water-flow-rate              104.12
  run-speed-range              (2160 2640)
  required-power               18.14
  required-capacity            500
  maximum-platform-deflection  .01
  platform-side-length         120
  cost                         1299.62
  evaluations                  ((:pump-agent :good)
                                (:heat-exchanger-agent :excellent))
  history-id                   "heatx-id-4"
  creation-time                1
  initiating-agent             :heat-exchanger-agent
  heat-exchanger               #<he::heat-exchanger "heat-exchanger-1">
  pump                         #<pp::pump "pump-2">
```

Figure A.1. *A Simplified Two-Agent Design Developed by* STEAM

APPENDIX B

EXPERIMENTAL DATA APPENDIX

This appendix contains the 101 problem specifications that were used in all experiments run in the **STEAM** system. The first four were taken from Meunier's thesis work on steam condenser design [Meunier, 1988], the others were randomly generated. The first problem specification is infeasible and results in a system failure. It was used primarily to test the system and for comparison purposes with the **IRSys** system described in Chapter 6. Other than those in that chapter, the experiments were run without the first problem specification.

Table B.1 lists the values assigned to each of the variables in a problem specification. These variables are the *required capacity* of the condenser in pounds/hour, the *platform side length* in inches, and the *maximum platform deflection*, also in inches.

Table B.1. *Problem Specifications Used in the Dissertation Experiments*

Trial #	Required Capacity (lb/hour)	Platform Side Length (inches)	Maximum Platform Deflection (inches)	Trial #	Required Capacity (lb/hour)	Platform Side Length (inches)	Maximum Platform Deflection (inches)
0	3000	120	0.01	51	1131	182	0.03
1	2000	120	0.01	52	1144	101	0.06
2	1000	40	0.10	53	1444	197	0.03
3	100	72	0.01	54	533	163	0.08
4	736	185	0.03	55	1415	180	0.07
5	1654	50	0.09	56	1104	103	0.10
6	500	110	0.06	57	1276	76	0.09
7	1323	86	0.01	58	1262	143	0.10
8	1411	223	0.03	59	1478	202	0.07
9	1726	163	0.05	60	755	40	0.05
10	613	220	0.03	61	1038	179	0.05
11	771	95	0.04	62	875	138	0.02
12	1584	235	0.07	63	1550	147	0.03
13	113	181	0.01	64	1337	79	0.07
14	233	111	0.05	65	1011	199	0.03
15	1174	107	0.08	66	1170	41	0.08
16	977	70	0.05	67	1454	135	0.07
17	1583	178	0.05	68	1323	183	0.08
18	734	211	0.08	69	1672	87	0.06
19	144	234	0.04	70	1402	177	0.05
20	1230	167	0.03	71	406	161	0.10
21	1230	161	0.04	72	1968	156	0.09
22	520	152	0.09	73	919	39	0.05
23	809	108	0.02	74	1137	64	0.03
24	1017	185	0.07	75	1565	208	0.07
25	1750	182	0.06	76	1332	160	0.02
26	184	135	0.06	77	1740	193	0.08
27	1849	43	0.07	78	816	38	0.03
28	396	48	0.05	79	1215	117	0.01
29	1924	154	0.01	80	370	226	0.04
30	806	86	0.08	81	1335	146	0.03
31	1037	129	0.04	82	884	85	0.06
32	1447	172	0.03	83	1443	155	0.02
33	1316	92	0.06	84	1772	153	0.01
34	1118	155	0.05	85	1242	63	0.01
35	1223	150	0.08	86	1422	234	0.05
36	1292	59	0.10	87	1379	207	0.04
37	1455	106	0.09	88	1607	111	0.07
38	906	64	0.09	89	692	221	0.08
39	1362	228	0.02	90	1407	119	0.03
40	1262	100	0.04	91	1180	78	0.04
41	1068	98	0.07	92	1113	57	0.04
42	1457	51	0.10	93	569	209	0.08
43	1295	62	0.08	94	1134	68	0.01
44	967	104	0.06	95	1108	110	0.09
45	541	36	0.02	96	1693	43	0.07
46	143	71	0.09	97	1074	162	0.07
47	1242	144	0.07	98	1178	144	0.02
48	1264	56	0.07	99	187	218	0.08
49	1840	128	0.07	100	1075	166	0.09
50	1087	171	0.02				

APPENDIX C

SOLUTION ACCEPTABILITY POLICIES

In this appendix, we present the solution-acceptability policy used by the **STEAM** system. The policy begins with agent acceptability: a design is considered acceptable only if all agents have had an opportunity to review the design and have rated it as acceptable. When that is the case, an objective function defined at the framework level by the user is applied to the design and returns a rating. This rating is then compared to a threshold that is defined by the user, and if the rating is within the threshold guidelines, the solution is considered globally acceptable.

The code used to implement the global acceptability policy of the **STEAM** system is shown below. This code is taken directly from the system and is explained only through embedded comments. Comment lines are those beginning with 2 semicolons. `defvar` is a Common Lisp construct used to define a system variable. `defun` is a Common Lisp construct that is used to define a function. `ks` and `ksa` are part of the blackboard vocabulary of the underlying framework and are not important to understanding the functionality of the code. A construct of the form (design$x$ design$_y$) is an accessor for attribute x of design$_y$. The `ACCEPTABILITY-POLICY` function is only applied to completed designs that all agents have had the opportunity to extend or critique.

```
(defun ACCEPTABILITY-POLICY (ks ksa)

  "This function is applied to completed solutions."

  ;; get the design to be evaluated
  (let* ((design (first (ksa.stimulus-units ksa))))

    ;; if any agent has rated the design as infeasible, mark it as
    ;; globally infeasible
    (cond ((eq (design$agent-acceptability design) :infeasible)
           (setf (design$acceptability design) :infeasible))

          ;; otherwise, if any agent has rated the design as
          ;; unacceptable, mark it as globally unacceptable
          ((eq (design$agent-acceptability design) :unacceptable)
           (setf (design$acceptability design) :unacceptable))

          ;; otherwise, if the design is acceptable to all the
          ;; agents
          ((eq (design$agent-acceptability design) :acceptable)

           ;; set the rating of the design by applying the
           ;; user-defined objective function
           (setf (design$utility-value design)
                 (funcall *OBJECTIVE-FUNCTION* design))

           ;; if the user has defined any further acceptability
           ;; criteria (such as a threshold on the rating) apply
           ;; those now
           (cond ((meets-global-acceptability-criteria? design)

                  ;; if the design is acceptable based on those
                  ;; criteria, mark it as globally acceptable
                  (setf (design$acceptability design) :acceptable))

                 ;; otherwise mark it as unacceptable
                 (t (setf (design$acceptability design)
                          :unacceptable)))))))
```

```
(defvar *OBJECTIVE-FUNCTION*
        #'(lambda (design) (get-cost design))
  "A function used to rate a complete design.  In this case, we
   are just using the cost of the design.  Could use more complex
   criteria if desired however, e.g., 2 X cost + weight.")

(defvar *THRESHOLD-RATING* 10000
  "A threshold value describing the maximum value that will be
   accepted as the utility value of a design.  In STEAM, we are
   minimizing cost, so the threshold rating would be the maximum
   acceptable cost.  If we were maximizing some other utility
   value, this would be the minimum acceptable value.")

(defun MEETS-GLOBAL-ACCEPTABILITY-CRITERIA? (design)

  ;; This is used for minimization problems.
  ;; If a threshold has been defined for the global rating
  (cond (*threshold-rating*

         ;; check to be sure that the global rating is below
         ;; the threshold (for minimization problems)
         (<= (design$utility-value design) *threshold-rating*))

        ;; otherwise if no threshold has been defined, just
        ;; return true.
        (t t)))
```

APPENDIX D

TERMINATION POLICIES

In this appendix, we present the termination policy used by the **STEAM** system. The policy states that when three complete acceptable solutions are created by the system, no new solutions are created but any existing partial acceptable solutions are finished (brought to termination states). This requires two distinct search phases (*:initial* and *:stop*). After the :initial phase is complete (three acceptable solutions have been created), the framework controller notifies the agents to switch to the :stop phase. The agents respond by activating and deactivating appropriate operators. The system halts when all solutions are in termination states.

The code used to implement the termination policy of the **STEAM** system is shown below. This code is taken directly from the system and is explained only through embedded comments. Comment lines are those beginning with 2 semicolons. **defvar** is a Common Lisp construct used to define a system variable. **defun** is a Common Lisp construct that is used to define a function. **find-units** is a pattern-matching function that retrieves structures from the underlying blackboard memory. **ks** and **ksa** are part of the blackboard vocabulary of the underlying framework and are not important to understanding the functionality of the code. A construct of the form (design$\$x$ design$_y$) is an accessor for attribute x of design$_y$.

```
(defvar *NUMBER-OF-DESIRED-DESIGNS* 3
  "A threshold value used to control how many acceptable
   designs should be generated before stopping.")


(defun STOP? ()
  "STOP? executes at the end of every framework cycle to determine
   whether the stopping criteria for the system are met.  If not,
   STOP? returns nil and processing continues.  If so, STOP? returns
   the value :stop."

  ;; retrieve all complete acceptable designs from the shared
  ;; memory
  (let ((acceptable-designs

          ;; find-units is a GBB pattern-matching function that
          ;; describes the type of structure to be retrieved and
          ;; relevant constraints on attribute values.
          ;; In this case, we are looking for instances of
          ;; designs with completeness attributes set to the
          ;; value complete and acceptability attributes set to
          ;; the value acceptable.
          (find-units
            'design ;; the type of structure
            (make-paths '(framework-bb solutions)) ;; a description of
                                                   ;; the blackboard
                                                   ;; location of
                                                   ;; designs
            '(:and (:element-match :exact ;; a description of designs
                    :pattern-object       ;; to be retrieved
                       (:index-type (:dimension completeness
                                     :type :label)
                        :index-object :complete))
                   (:element-match :exact
                    :pattern-object
                       (:index-type (:dimension acceptability
                                     :type :label)
                        :index-object :acceptable))))))


    (when
      (and
        ;; if the system is currently in the initial phase
        (eq (instantiated-strategy$termination-phase *strategy*)
            :initial)
```

```
  ;; and if the required number of complete designs have been
  ;; found
  (>= (length acceptable-designs)
      cs::*number-of-desired-designs*))

;; reset the termination phase of the strategy to :stop
(setf (instantiated-strategy$termination-phase *strategy*)
      :stop)

;; return
t)))
```

APPENDIX E

EXPERIMENTAL COMPARISON OF STEAM AND IRSYS

This appendix contains the complete results from comparing the **STEAM** system to **IRSys**. A summary of the results appears in Chapter 6. One hundred and one comparative trials were run, where each trial consisted of running **STEAM** and **IRSys** on identical problem specifications. The first trial was run with an infeasible problem specification. The first four problem specifications were taken from [Meunier, 1988], while the rest were derived by generating random values of each specification parameter within the known parameter boundaries. The quality of solutions is measured in terms of the costs of the designs found by each system, where each system attempts to minimize the cost of its designs. Processing efficiency is measured both in terms of the number of component designs agents create and the real time each system takes. The results are listed in Tables E.1 and E.2.

Solution Quality: The average minimum cost for a design was $8375.77 in the **STEAM** system. In **IRSys**, the average cost was $9605.95. The **STEAM** designs were, on average, 12.81 percent less costly than those produced by **IRSys**.

System Performance: The average number of component proposals produced by the **STEAM** system was 84.24 compared to 203.82 for **IRSys**. We also measured real time and found that the **STEAM** system average was 114.53 seconds per run while **IRSys** averaged 240.21 seconds. If system performance is measured in terms of the number of component proposals produced, **STEAM** averaged a 58.67 improvement over **IRSys**. Measured in terms of real time per run, **STEAM** averaged a 52.32 percent improvement over **IRSys**.

Table E.1. *Comparison of Results from* **IRSys** *and* **STEAM**, *Part 1*

Problem Specification			Minimum Design Cost (Dollars)		Number of Proposals Generated		Processing Time (Seconds)	
Required Capacity (lb/hr)	Platform Side Length (inches)	Maximum Platform Deflection (inches)	IRSys	STEAM	IRSys	STEAM	IRSys	STEAM
3000	120	0.01	Fail	Fail	12	2	12.27	43.48
2000	120	0.01	11840.54	11281.71	131	99	158.37	130.72
1000	40	0.10	3351.45	3287.34	133	70	152.13	93.00
100	72	0.01	3214.76	1726.62	306	65	346.15	60.12
736	185	0.03	13016.13	12586.44	136	91	161.05	91.83
1654	50	0.09	5273.82	5361.99	167	90	192.27	112.80
500	110	0.06	4490.58	3454.88	136	82	162.38	88.18
1323	86	0.01	6535.89	5995.90	289	86	335.43	122.40
1411	223	0.03	21850.96	19327.01	95	86	115.12	117.47
1726	163	0.05	13285.73	12368.04	136	98	156.43	116.95
613	220	0.03	19240.94	16493.61	136	82	156.78	88.52
771	95	0.04	4553.63	3800.91	172	84	198.33	94.02
1584	235	0.07	21407.41	19173.54	253	81	293.47	98.82
113	181	0.01	15023.16	10518.12	306	65	351.47	72.65
233	111	0.05	4382.96	2785.41	306	65	352.42	58.52
1174	107	0.08	5511.62	5341.43	172	82	203.67	88.87
977	70	0.05	4006.45	3490.49	136	70	157.55	83.87
1583	178	0.05	14345.11	13288.29	253	81	294.45	87.23
734	211	0.08	15960.07	11191.36	255	91	306.60	112.52
144	234	0.04	21083.36	14378.82	306	65	357.47	73.37
1230	167	0.03	12422.38	10857.74	290	99	345.27	109.43
1230	161	0.04	11831.98	10365.74	290	99	346.97	101.18
520	152	0.09	6691.38	5152.76	136	84	162.27	88.38
809	108	0.02	6260.87	4819.96	218	90	248.72	138.65
1017	185	0.07	12087.78	10085.11	181	81	205.25	106.50
1750	182	0.06	15456.00	14006.79	132	94	153.73	120.88
184	135	0.06	5563.76	3423.57	306	65	347.73	73.30
1849	43	0.07	5993.82	5922.94	131	102	155.35	114.20
396	48	0.05	2149.16	1666.36	265	77	300.47	87.22
1924	154	0.01	14601.51	14601.51	131	91	158.47	121.42
806	86	0.08	3871.07	3226.76	218	90	249.27	120.72
1037	129	0.04	7693.11	6575.36	172	75	198.95	90.08
1447	172	0.03	13453.70	11986.35	289	82	336.97	129.42
1316	92	0.06	5422.33	5416.50	95	86	117.72	117.33
1118	155	0.05	9800.52	8429.08	208	73	240.12	102.38
1223	150	0.08	8555.68	7260.49	290	99	338.73	117.03
1292	59	0.10	4500.83	4320.95	131	90	157.50	120.48
1455	106	0.09	6263.90	5713.95	289	82	337.73	98.35
906	64	0.09	3429.67	3239.70	95	81	117.80	98.72
1362	228	0.02	25110.69	22602.71	254	90	295.75	119.60
1262	100	0.04	6055.68	5646.90	290	94	348.48	128.02
1068	98	0.07	4973.46	4803.27	172	81	205.83	108.10
1457	51	0.10	4838.60	4720.40	289	82	342.93	104.40
1295	62	0.08	4537.13	4512.55	130	86	156.65	111.78
967	104	0.06	5434.65	4622.89	172	70	199.37	91.18
541	36	0.02	2355.41	1932.82	177	81	204.27	90.73
143	71	0.09	2422.86	1193.92	306	65	351.00	73.52
1242	144	0.07	8202.88	6995.89	290	99	347.65	124.48
1264	56	0.07	4369.28	4296.18	290	94	343.20	129.75
1840	128	0.07	9085.72	9085.72	131	102	160.57	144.47

Table E.2. *Comparison of Results from* IRSys *and* STEAM, *Part 2*

Problem Specification			Minimum Design Cost (Dollars)		Number of Proposals Generated		Processing Time (Seconds)	
Required Capacity (lb/hr)	Platform Side Length (inches)	Maximum Platform Deflection (inches)	IRSys	STEAM	IRSys	STEAM	IRSys	STEAM
1087	171	0.02	13767.21	12134.97	172	81	203.27	131.72
1131	182	0.03	13731.47	13561.28	208	73	242.65	89.18
1144	101	0.06	5834.47	5154.23	208	69	244.47	93.50
1444	197	0.03	18161.65	16233.05	289	82	343.52	113.73
533	163	0.08	8722.86	7117.02	95	86	119.60	120.42
1415	180	0.07	12545.81	10888.31	95	86	114.48	128.92
1104	103	0.10	5351.64	4423.57	205	79	232.50	106.98
1276	76	0.09	4633.28	4720.29	290	102	341.52	135.82
1262	143	0.10	8145.48	7214.25	290	94	348.50	139.17
1478	202	0.07	16819.70	13044.24	289	77	349.68	99.20
755	40	0.05	2908.63	2527.16	136	87	165.75	121.88
1038	179	0.05	13087.52	9655.36	168	75	195.05	105.32
875	138	0.02	8723.24	7420.71	172	96	202.32	132.70
1550	147	0.03	11415.33	10285.69	132	74	155.62	115.62
1337	79	0.07	5253.24	4771.00	289	94	338.02	130.13
1011	199	0.03	17221.84	15120.41	256	81	302.02	127.83
1170	41	0.08	3962.37	3708.13	172	94	205.52	125.68
1454	135	0.07	8223.50	7324.10	289	82	341.82	113.92
1323	183	0.08	12689.34	10844.70	289	86	349.42	124.93
1672	87	0.06	6450.38	6377.29	208	94	249.68	141.35
1402	177	0.05	13844.51	12240.56	95	86	114.75	130.47
406	161	0.10	7102.96	5324.11	265	88	305.40	122.82
1968	156	0.09	11667.73	11667.73	131	95	158.68	146.68
919	39	0.05	3415.54	2906.15	136	77	162.65	117.57
1137	64	0.03	4408.67	4238.48	208	69	244.08	98.25
1565	208	0.07	17911.83	16183.29	132	81	156.07	110.37
1332	160	0.02	13277.09	12042.68	289	86	347.85	119.27
1740	193	0.08	14795.82	14627.28	95	94	119.77	127.32
816	38	0.03	2906.07	2704.83	218	87	253.08	122.90
1215	117	0.01	8844.52	7992.19	168	99	197.18	157.62
370	226	0.04	19795.36	14329.37	306	77	354.57	122.73
1335	146	0.03	10711.89	9737.31	289	86	349.78	146.50
884	85	0.06	4093.79	3409.66	172	77	212.08	123.52
1443	155	0.02	11786.00	11797.85	289	82	342.27	141.15
1772	153	0.01	13992.03	13992.03	208	97	250.53	141.05
1242	63	0.01	5047.93	4679.29	290	99	342.92	142.05
1422	234	0.05	20872.61	20835.11	95	86	119.57	145.47
1379	207	0.04	17171.79	17263.01	218	98	254.40	151.83
1607	111	0.07	7486.15	7215.44	136	78	161.92	127.83
692	221	0.08	17183.60	12015.36	218	96	255.77	122.35
1407	119	0.03	7986.06	7240.51	95	86	115.68	138.03
1180	78	0.04	4706.87	4536.68	172	82	201.13	148.38
1113	57	0.04	4281.62	3948.98	208	73	246.12	114.02
569	209	0.08	15340.14	10539.42	95	81	115.70	95.85
1134	68	0.01	4950.27	4548.88	208	73	248.27	103.67
1108	110	0.09	5347.86	5177.67	172	79	201.45	117.28
1693	43	0.07	5499.93	5410.61	208	94	246.72	146.48
1074	162	0.07	10093.86	8611.47	172	81	204.95	139.27
1178	144	0.02	10015.07	8808.08	172	82	201.08	121.97
187	218	0.08	16175.96	10194.62	306	65	356.22	127.42
1075	166	0.09	10421.86	7496.07	172	81	208.70	111.55

APPENDIX F

RESULTS OF INFORMATION-ASSIMILATION EXPERIMENTS

In this appendix, we tabulate the results of the information-assimilation experiments described in Chapter 8. The first two tables (Tables F.1 and F.2) compare quality measures for the results generated by the system under each problem specification with assimilation operators active and inactive. The quality of a trial is defined as the cost of the best design found during that trial (best is defined as minimum cost). The improvement in quality when the assimilation operators are active is expressed as a percentage. The mean cost in assimilation trials was $8504.77 and in the non-assimilation-trials its was $9020.43 for a mean cost improvement of 5.72%.

The second set of tables, Tables F.3 and F.4, shows the average cost of the three best designs found in each trial. Under this measurement, the average cost improvement was 6.57%.

The third set of tables, Tables F.5 and F.6, presents measures of processing efficiency, the runtime of the system (actual runtime in seconds) and the runtime-per-solution (runtime / number-of-acceptable-solutions) for each trial. The average runtime with assimilation enabled is 121.98 seconds, with assimilation disabled, it is 132.67 seconds. Runtimes were, on average, 8.06% lower with assimilation enabled. However, direct comparison of runtimes is somewhat misleading as described in Chapter 8. To make the comparisons more meaningful, we also measured runtime-per-solution. In the assimilation trials, the average runtime-per-solution was 11.58 seconds and in the non-assimilation trials, it was 19.5 seconds. The average percent improvement under this standard of measurement is 40.62%.

In the last set of tables, Tables F.7 and F.8, the costs of different activities relating to information sharing are detailed in the STEAM system. These costs are constraint-generation (CG), determination-of-transmission costs (TD), translation costs (TR), and local-management costs (LM). These categories are described in Chapter 8, Section 8.2. These are all costs associated with the processing efficiency of the system and are measured in seconds. The average constraint-generation time was found to be 7.3 seconds, determination-of-transmission time was .4 seconds, translation time was .5 seconds, and local management time was 4.2 seconds, for an average total time for information assimilation of 12.4 seconds, or 10.17% of runtime. These figures are highly domain-dependent and are to be used only as benchmarks.

Table F.1. *Design Quality in Information-Assimilation Trials, Part 1*

Experiment	Minimum Cost With Assimilation (dollars)	Minimum Cost Without Assimilation (dollars)	Cost Improvement (percentage)
1	12586.44	12597.57	0.09
2	5346.52	5346.52	0.00
3	3451.29	3499.51	1.38
4	6346.95	6346.95	0.00
5	19324.36	22044.69	12.34
6	12354.16	13285.73	7.01
7	16479.72	17323.20	4.87
8	3797.33	3911.53	2.92
9	19170.88	21479.88	10.75
10	10506.65	10506.65	0.00
11	2778.23	2778.23	0.00
12	5338.78	5414.87	1.41
13	3489.40	3733.58	6.54
14	13285.63	14417.58	7.85
15	11188.71	11231.62	0.38
16	14371.65	14371.65	0.00
17	10855.09	12235.55	11.28
18	10363.09	11645.15	11.01
19	5150.11	5150.11	0.00
20	4817.31	6259.06	23.03
21	10071.23	10553.44	4.57
22	13994.47	15326.81	8.69
23	3416.40	3416.40	0.00
24	5922.94	5922.94	0.00
25	1665.27	1500.42	-10.99
26	14609.81	14609.81	0.00
27	3224.11	3378.87	4.58
28	6571.78	7036.64	6.61
29	11983.70	15068.16	20.47
30	5413.85	5820.95	6.99
31	8426.43	8806.53	4.32
32	7257.84	8432.44	13.93
33	4318.30	4476.25	3.53
34	5711.30	5837.55	2.16
35	3238.61	3446.55	6.03
36	22600.06	25183.16	10.26
37	5644.25	6055.77	6.80
38	4800.62	4850.46	1.03
39	4717.75	4844.00	2.61
40	4512.55	4512.55	0.00
41	4621.80	4865.98	5.02
42	1918.93	1996.29	3.88
43	1182.45	1182.45	0.00
44	6993.24	8079.64	13.45
45	4282.30	4369.37	1.99
46	9094.02	9094.02	0.00
47	12132.32	12480.74	2.79
48	13544.64	13544.64	0.00
49	5151.58	5742.28	10.29
50	16230.40	16523.48	1.77

Table F.2. *Design Quality in Information-Assimilation Trials, Part 2*

Experiment	Minimum Cost With Assimilation (dollars)	Minimum Cost Without Assimilation (dollars)	Cost Improvement (percentage)
51	5792.03	6073.69	4.64
52	10885.66	14359.54	24.19
53	4420.92	4769.34	7.31
54	4720.29	4720.29	0.00
55	7211.60	8145.57	11.47
56	13041.58	13041.58	0.00
57	2523.58	2781.71	9.28
58	9651.78	11718.69	17.64
59	7419.62	7419.62	0.00
60	10283.03	11135.78	7.66
61	5029.76	5066.41	0.72
62	15106.53	15123.88	0.11
63	3705.48	3865.62	4.14
64	7321.45	7447.70	1.70
65	12500.40	12500.40	0.00
66	6364.96	6450.38	1.32
67	12237.91	14038.24	12.82
68	5323.02	5273.37	-0.94
69	11676.04	11676.04	0.00
70	2905.06	3119.63	6.88
71	4235.83	4316.48	1.87
72	16180.63	17817.55	9.19
73	12028.80	13090.26	8.11
74	14627.28	14627.28	0.00
75	2702.18	2751.47	1.79
76	7989.54	8162.42	2.12
77	14322.20	14040.44	-2.01
78	10525.06	10525.06	0.00
79	3407.01	3566.08	4.46
80	11795.20	12088.28	2.42
81	13992.03	13992.03	0.00
82	4676.64	4861.10	3.79
83	20832.46	23804.14	12.48
84	17260.36	19474.44	11.37
85	7212.78	7376.58	2.22
86	12011.78	12012.71	0.01
87	7237.86	8887.84	18.56
88	4534.03	4610.12	1.65
89	3946.33	4094.79	3.63
90	10525.53	10543.82	0.17
91	4546.23	4763.44	4.56
92	5175.02	5312.52	2.59
93	5410.61	5410.61	0.00
94	8608.82	8957.24	3.89
95	8805.43	9918.32	11.22
96	10187.45	10187.45	0.00
97	7493.42	7841.84	4.44
98	9938.04	11501.44	13.59
99	4495.37	4843.79	7.19
100	13397.35	13397.35	0.00

Table F.3. *Average Design Quality in Information-Assimilation Trials, Part 1*

Experiment	Average Cost With Assimilation	Average Cost Without Assimilation	Cost Improvement (percentage)
1	12630.54	12796.39	1.30
2	5409.52	5422.07	0.23
3	3483.44	3600.94	3.26
4	6374.07	6374.07	0.00
5	19324.36	22928.32	15.72
6	12354.68	13387.99	7.72
7	16491.29	18087.09	8.82
8	3849.04	4257.98	9.60
9	19170.88	21546.82	11.03
10	10509.51	10509.51	0.00
11	2782.81	2782.81	0.00
12	5364.14	5565.19	3.61
13	3510.49	3925.20	10.57
14	13285.63	14484.52	8.28
15	11214.12	13651.58	17.85
16	14371.65	14371.65	0.00
17	10855.09	12297.69	11.73
18	10363.09	11707.29	11.48
19	5214.16	5431.08	3.99
20	4888.97	6506.38	24.86
21	10082.79	11960.69	15.70
22	14321.42	15476.56	7.46
23	3489.96	3489.96	0.00
24	5967.93	5967.93	0.00
25	1671.07	1609.85	-3.80
26	15244.75	15244.75	0.00
27	3295.77	3741.36	11.91
28	6573.54	7674.40	14.34
29	12081.39	15186.10	20.44
30	5550.22	5848.07	5.09
31	8426.43	10196.76	17.36
32	7257.84	8827.33	17.78
33	4423.60	4503.37	1.77
34	5906.69	6110.06	3.33
35	3248.32	3481.22	6.69
36	22678.80	25214.75	10.06
37	5913.70	6104.02	3.12
38	4800.62	4893.46	1.90
39	4900.81	4942.90	0.85
40	4517.92	4539.67	0.48
41	4642.89	5227.16	11.18
42	1930.50	2144.69	9.99
43	1185.31	1185.31	0.00
44	7705.87	8769.39	12.13
45	4388.60	4417.62	0.66
46	9605.96	9451.96	-1.63
47	12132.32	13270.17	8.57
48	13553.97	13725.61	1.25
49	5151.58	5817.64	11.45
50	16230.40	17167.46	5.46

Table F.4. *Average Design Quality in Information-Assimilation Trials, Part 2*

Experiment	Average Cost With Assimilation	Average Cost Without Assimilation	Cost Improvement (percentage)
51	5792.03	6215.30	6.81
52	10885.66	14414.35	24.48
53	4595.19	4973.55	7.61
54	4721.33	4903.13	3.71
55	7829.35	8193.82	4.45
56	13041.58	13289.39	1.86
57	2537.22	2833.12	10.44
58	9718.44	12659.10	23.23
59	7518.28	7477.74	-0.54
60	10605.60	11290.72	6.07
61	5080.60	5164.89	1.63
62	15118.09	16465.98	8.19
63	3758.86	3924.25	4.21
64	7516.84	7720.21	2.63
65	12573.95	13085.67	3.91
66	6393.44	6575.69	2.77
67	12237.91	14093.05	13.16
68	5343.71	5344.40	0.01
69	12073.22	12073.22	0.00
70	2928.42	3198.37	8.44
71	4235.83	4528.37	6.46
72	16180.63	17867.05	9.44
73	12028.80	13142.50	8.47
74	14819.04	14847.51	0.19
75	2716.00	2881.43	5.74
76	7989.54	8515.33	6.17
77	15852.19	14154.22	-12.00
78	10577.30	10577.30	0.00
79	3488.79	3968.28	12.08
80	11795.20	12485.86	5.53
81	14262.04	14290.38	0.20
82	4809.17	4923.24	2.32
83	20832.46	23932.71	12.95
84	17260.36	19579.81	11.85
85	7358.50	7456.34	1.31
86	12025.42	13730.64	12.42
87	7237.86	8942.65	19.06
88	4559.39	4760.44	4.22
89	3946.33	4223.29	6.56
90	10537.10	10872.58	3.09
91	4546.23	4923.34	7.66
92	5303.33	5419.47	2.14
93	5453.50	5453.50	0.00
94	8608.82	9646.77	10.76
95	9176.39	9976.95	8.02
96	10261.01	10261.01	0.00
97	7493.42	9049.81	17.20
98	10968.67	12509.19	12.32
99	4582.51	4910.82	6.69
100	13506.40	13506.40	0.00

Table F.5. *Run Time and Runtime-per-Solution Measurements, Part 1*

Experiment	With Assimilation			Without Assimilation		
	Run Time	# of Solutions	Runtime per Solution	Run Time	# of Solutions	Runtime per Solution
1	265.95	12	22.16	228.78	6	38.13
2	136.22	13	10.48	149.77	7	21.40
3	80.73	8	10.09	86.67	10	8.67
4	117.90	10	11.79	115.42	5	23.08
5	106.77	10	10.68	144.57	6	24.10
6	103.47	13	7.96	127.28	9	14.14
7	89.62	12	7.47	85.00	7	12.14
8	92.62	11	8.42	109.90	6	18.32
9	97.98	12	8.16	118.85	8	14.86
10	70.92	10	7.09	66.38	9	7.38
11	67.38	10	6.74	67.45	9	7.49
12	90.52	7	12.93	102.03	6	17.00
13	94.58	8	11.82	119.20	6	19.87
14	100.70	12	8.39	120.58	8	15.07
15	98.53	12	8.21	105.08	6	17.51
16	70.95	10	7.09	67.85	9	7.54
17	113.53	15	7.57	112.17	7	16.02
18	119.33	15	7.96	106.98	7	15.28
19	94.60	12	7.88	83.37	8	10.42
20	100.95	12	8.41	102.28	4	25.57
21	92.45	10	9.24	105.53	6	17.59
22	106.82	12	8.90	123.53	8	15.44
23	72.50	10	7.25	66.02	9	7.34
24	118.37	14	8.45	133.73	10	13.37
25	85.38	12	7.11	77.72	8	9.72
26	124.97	9	13.89	132.53	9	14.73
27	99.43	10	9.94	131.58	5	26.32
28	99.38	8	12.42	119.15	7	17.02
29	111.98	9	12.44	169.80	6	28.30
30	121.78	12	10.15	121.23	6	20.20
31	92.28	7	13.18	118.63	7	16.95
32	122.05	15	8.14	122.20	7	17.46
33	116.47	13	8.96	120.83	7	17.26
34	103.57	11	9.42	140.72	9	15.64
35	111.20	11	10.11	125.88	6	20.98
36	120.02	12	10.00	134.72	6	22.45
37	134.05	14	9.57	139.92	8	17.49
38	102.50	11	9.32	131.03	6	21.84
39	96.15	11	8.74	149.68	9	16.63
40	123.87	12	10.32	114.83	6	19.14
41	93.48	8	11.69	111.52	6	18.59
42	98.83	11	8.98	111.13	10	11.11
43	74.95	10	7.49	73.17	9	8.13
44	147.28	15	9.82	119.72	7	17.10
45	122.77	14	8.77	144.77	8	18.10
46	133.58	14	9.54	145.60	10	14.56
47	116.90	11	10.63	117.05	7	16.72
48	95.68	7	13.67	138.00	8	17.25
49	107.10	6	17.85	118.23	7	16.89
50	116.47	9	12.94	168.42	6	28.07

Table F.6. *Run Time and Runtime-per-Solution Measurements, Part 2*

Experiment	With Assimilation			Without Assimilation		
	Run Time	# of Solutions	Runtime per Solution	Run Time	# of Solutions	Runtime per Solution
51	121.45	13	9.34	125.43	10	12.54
52	124.45	10	12.44	173.45	6	28.91
53	120.90	9	13.43	131.20	7	18.74
54	144.10	15	9.61	138.37	7	19.77
55	158.85	14	11.35	160.02	8	20.00
56	135.23	10	13.52	137.72	9	15.30
57	119.58	11	10.87	120.52	6	20.09
58	103.62	8	12.95	125.80	7	17.97
59	168.03	7	24.00	217.60	7	31.09
60	113.17	10	11.32	130.60	6	21.77
61	125.60	11	11.42	140.80	6	23.47
62	117.43	10	11.74	143.97	6	24.00
63	136.72	14	9.77	140.37	7	20.05
64	115.83	11	10.53	151.95	9	16.88
65	120.63	10	12.06	120.38	5	24.08
66	140.82	14	10.06	142.53	7	20.36
67	134.67	10	13.47	175.05	6	29.18
68	114.18	13	8.78	91.15	8	11.39
69	172.93	10	17.29	168.60	10	16.86
70	119.20	9	13.24	147.45	6	24.57
71	115.30	6	19.22	134.37	7	19.20
72	134.83	11	12.26	152.48	8	19.06
73	143.68	10	14.37	131.83	5	26.37
74	134.80	12	11.23	148.28	8	18.53
75	135.78	12	11.31	152.97	5	30.59
76	155.73	15	10.38	160.13	7	22.88
77	112.98	12	9.41	90.52	8	11.31
78	120.95	10	12.09	127.50	5	25.50
79	114.52	9	12.72	146.97	6	24.50
80	133.88	9	14.88	226.07	6	37.68
81	132.53	12	11.04	197.18	8	24.65
82	154.12	15	10.27	139.22	7	19.89
83	147.57	10	14.76	188.27	6	31.38
84	150.90	11	13.72	168.32	6	28.05
85	135.22	11	12.29	163.30	7	23.33
86	143.95	13	11.07	144.23	7	20.60
87	146.85	10	14.69	188.27	6	31.38
88	121.78	7	17.40	130.30	6	21.72
89	122.13	7	17.45	144.63	7	20.66
90	168.25	11	15.30	135.35	9	15.04
91	122.58	7	17.51	158.93	8	19.87
92	127.28	9	14.14	152.25	7	21.75
93	142.68	12	11.89	179.92	9	19.99
94	130.60	11	11.87	143.05	6	23.84
95	125.45	7	17.92	145.47	6	24.25
96	102.00	10	10.20	94.35	9	10.48
97	134.97	11	12.27	151.47	6	25.25
98	161.50	15	10.77	137.48	7	19.64
99	207.33	11	18.85	143.90	7	20.56
100	172.90	9	19.21	164.48	9	18.28

Table F.7. *Information-Sharing Costs, Part 1*

Experiment	With Assimilation				Without Assimilation			
	CG	TD	TR	LM	CG	TD	TR	LM
1	6.70	0.42	80.42	14.47	6.70	0.53	55.00	20.15
2	7.98	0.27	0.42	4.08	7.98	0.60	0.53	3.74
3	6.33	0.28	0.40	3.63	6.33	0.33	0.24	3.07
4	7.58	0.62	0.49	4.80	7.58	0.52	0.47	3.44
5	7.67	0.70	0.42	6.00	7.67	1.07	0.89	5.88
6	7.85	0.28	0.27	3.58	7.85	0.53	0.54	3.79
7	6.50	0.20	0.27	3.55	6.50	0.35	0.34	3.01
8	6.82	0.38	0.32	3.66	6.82	0.70	0.53	3.72
9	7.82	0.17	0.26	3.46	7.82	0.57	0.54	3.77
10	6.33	0.15	0.36	2.53	6.33	0.12	0.05	2.29
11	6.25	0.08	0.11	2.46	6.25	0.10	0.05	2.29
12	7.47	0.40	0.39	3.62	7.47	0.45	0.47	3.48
13	7.25	0.27	0.52	3.41	7.25	0.52	0.31	3.97
14	7.83	0.17	0.30	3.35	7.83	0.55	0.55	3.76
15	6.77	0.40	0.52	3.52	6.77	0.48	0.48	3.38
16	6.33	0.15	0.31	2.57	6.33	0.13	0.05	2.29
17	7.48	0.48	0.42	4.11	7.48	0.53	0.75	3.85
18	0.00	0.50	0.48	4.11	0.00	0.58	0.71	3.85
19	6.42	0.23	0.27	3.63	6.42	0.45	0.29	2.80
20	6.93	0.37	0.50	4.18	6.93	0.70	0.64	4.00
21	7.25	0.17	0.23	3.29	7.25	0.37	0.27	3.30
22	7.95	0.40	0.37	3.64	7.95	0.60	0.55	3.76
23	6.28	0.10	0.14	2.45	6.28	0.13	0.05	2.29
24	7.92	0.28	0.35	3.62	7.92	0.75	0.54	3.83
25	6.30	0.15	0.31	3.19	6.30	0.40	0.40	2.71
26	7.92	0.63	0.60	6.13	7.92	0.57	0.44	4.13
27	6.88	0.48	0.46	4.12	6.88	0.62	0.46	5.01
28	7.22	0.23	0.43	3.61	7.22	0.47	0.47	3.66
29	7.68	0.77	0.45	6.27	7.68	1.02	0.99	6.61
30	7.67	0.63	0.55	5.65	7.67	0.58	0.68	3.82
31	7.28	0.22	0.37	3.63	7.28	0.53	0.52	3.86
32	7.38	0.45	0.41	4.12	7.38	0.60	0.86	3.85
33	7.45	0.62	0.46	5.28	7.45	0.58	0.69	3.92
34	7.67	0.22	0.27	3.46	7.67	0.75	0.57	4.33
35	6.97	0.20	0.37	3.74	6.97	0.52	0.48	3.98
36	7.55	0.73	0.46	5.44	7.55	0.80	0.88	4.91
37	7.57	0.60	0.51	4.68	7.57	0.68	0.80	4.30
38	7.30	0.13	0.34	3.68	7.30	0.50	0.73	3.48
39	7.63	0.27	0.24	3.46	7.63	0.77	0.60	4.34
40	7.67	0.62	0.59	5.37	7.67	0.50	0.71	3.87
41	7.05	0.20	0.31	3.25	7.05	0.47	0.44	3.65
42	6.35	0.20	0.33	3.49	6.35	0.55	0.59	3.43
43	6.20	0.10	0.18	2.59	6.20	0.12	0.06	2.29
44	7.40	0.65	0.66	4.41	7.40	0.47	0.65	3.82
45	7.40	0.52	0.49	4.26	7.40	0.68	0.70	4.34
46	7.82	0.30	0.39	3.87	7.82	0.58	0.54	3.92
47	7.38	0.22	0.49	3.88	7.38	0.57	0.77	3.51
48	7.30	0.20	0.35	3.63	7.30	0.55	0.76	3.88
49	7.47	0.38	0.37	4.01	7.47	0.63	0.68	3.86
50	7.67	0.70	0.50	5.84	7.67	1.00	1.94	6.02

Table F.8. *Information Sharing Costs, Part 2*

Experiment	With Assimilation				Without Assimilation			
	CG	TD	TR	LM	CG	TD	TR	LM
51	6.50	0.20	0.37	3.80	6.50	0.47	0.45	3.21
52	7.68	1.02	0.61	6.03	7.68	1.05	0.88	6.06
53	7.48	0.22	0.49	4.03	7.48	0.53	0.66	3.86
54	7.47	0.67	0.53	6.12	7.47	0.58	0.64	4.10
55	7.70	0.58	0.54	4.51	7.70	0.67	0.66	4.57
56	7.95	0.45	2.03	3.80	7.95	0.57	0.40	3.99
57	6.73	0.40	0.49	3.61	6.73	0.58	0.48	3.81
58	7.25	0.27	0.51	3.40	7.25	0.43	0.51	3.67
59	6.93	0.63	0.48	4.29	6.93	0.58	0.34	4.60
60	7.97	0.22	0.32	3.62	7.97	0.50	0.57	3.72
61	7.62	0.78	0.58	5.26	7.62	0.75	0.81	4.64
62	7.15	0.20	0.50	3.66	7.15	0.40	0.52	3.64
63	7.52	0.43	0.40	4.43	7.52	0.63	0.69	4.19
64	7.75	0.37	0.42	3.56	7.75	0.77	0.60	4.41
65	7.58	0.67	0.54	5.13	7.58	0.57	0.62	3.62
66	7.80	0.33	0.43	4.12	7.80	0.55	0.54	3.54
67	7.73	0.85	0.60	5.92	7.73	1.03	0.94	6.36
68	6.37	0.20	0.45	4.02	6.37	0.40	0.35	2.94
69	7.83	0.68	0.59	6.32	7.83	0.62	0.56	4.52
70	7.05	0.25	0.44	3.63	7.05	0.58	0.42	4.05
71	7.77	0.42	0.44	3.95	7.77	0.60	0.70	3.91
72	7.77	0.30	0.43	3.76	7.77	0.62	0.54	3.84
73	7.63	0.63	0.53	5.15	7.63	0.57	0.63	3.62
74	7.88	0.37	0.53	4.00	7.88	0.52	0.48	4.15
75	6.95	0.47	0.63	4.98	6.95	0.75	0.68	5.49
76	7.50	0.60	0.62	4.42	7.50	0.58	0.67	4.21
77	6.37	0.23	0.67	3.56	6.37	0.43	0.32	2.78
78	7.62	0.57	0.66	4.81	7.62	0.55	0.60	3.82
79	7.07	0.23	0.50	3.31	7.07	0.70	0.44	4.24
80	7.77	0.95	0.60	6.11	7.77	1.03	1.32	6.17
81	7.88	0.35	0.46	3.85	7.88	0.60	0.89	4.22
82	7.58	0.63	0.55	4.68	7.58	0.55	0.58	3.92
83	7.63	0.78	0.55	6.30	7.63	1.12	0.91	6.24
84	7.82	0.70	0.62	5.81	7.82	0.97	0.90	5.49
85	7.97	0.22	0.45	3.75	7.97	0.50	0.53	4.08
86	6.57	0.40	0.48	3.86	6.57	0.60	0.49	4.24
87	7.73	0.83	0.58	6.28	7.73	1.07	0.89	6.26
88	7.42	0.47	0.51	3.93	7.42	0.53	0.60	3.53
89	7.42	0.28	0.51	3.94	7.42	0.62	0.68	3.86
90	6.57	0.20	0.47	3.95	6.57	0.53	0.44	3.75
91	7.58	0.28	1.77	3.96	7.58	0.60	0.68	4.23
92	7.83	0.20	0.46	3.96	7.83	0.57	0.59	4.23
93	7.80	0.37	0.42	3.90	7.80	0.60	0.63	4.46
94	7.47	0.23	0.46	3.98	7.47	0.58	0.56	3.86
95	7.62	0.47	0.52	4.00	7.62	0.53	0.57	3.81
96	6.27	0.17	0.28	2.80	6.27	0.22	0.23	2.58
97	7.32	0.22	0.62	3.93	7.32	0.52	0.62	3.84
98	7.57	0.70	0.60	4.58	7.57	0.52	0.58	3.85
99	7.40	0.22	0.54	5.11	7.40	0.57	0.63	3.68
100	8.03	0.70	0.93	6.42	8.03	0.63	0.74	4.41

APPENDIX G

RESULTS OF AGENT/ROLE ASSIGNMENT EXPERIMENTS

In this appendix, we tabulate the results of the agent/role assignment experiments described in Chapter 9. The trial categories are described in detail in Chapter 9. Briefly, in the first category, only **pump-agent** initiates solutions, in the second category, only **heat-exchanger-agent** initiates solutions, and in the third category, both **pump-agent** and **heat-exchanger-agent** initiate solutions. The problem specifications are numbers 1-100 as shown in Appendix B.

Tables G.1 and G.2 show solution quality (as measured by the cost of the best design found) and system runtime (measured in elasped real time) for each problem specification for three categories of trials. The average cost of a design in Category 1 was $8313.88, runtime was 173.51 seconds. In Category 2, cost was $8798.64 and runtime was 169 seconds. In Category 3, cost was $8375.77 and runtime was 151.48 seconds.

Tables G.3 and G.4 show the number of solutions generated and the runtime-per-solution for each problem specification. The average runtime-per-solution for Category 1 was 12.47 seconds, Category 2 was 13.71 seconds, and Category 3 was 13.82 seconds. These results are analyzed in Chapter 9, Section 9.2.

Table G.1. *Agent/Role Assignment Quality Results, Part 1*

Problem Specification Number	Minimum Cost Category 1 (dollars)	Minimum Cost Category 2 (dollars)	Minimum Cost Category 3 (dollars)	Runtime Category 1 (seconds)	Runtime Category 2 (seconds)	Runtime Category 3 (seconds)
1	11534.64	11195.12	11281.71	361.83	168.03	207.28
2	3101.16	3287.34	3287.34	122.17	132.68	104.37
3	1726.62	2375.90	1726.62	65.55	100.87	78.23
4	12554.66	12597.57	12586.44	120.62	127.28	127
5	5491.94	5361.99	5361.99	166.87	172.40	223.02
6	3454.88	3502.16	3454.88	101.95	150.55	112.88
7	5995.90	6346.95	5995.90	176.43	121.47	131.42
8	19327.01	19327.01	19327.01	241.65	160.88	138.37
9	12368.04	13285.73	12368.04	166.90	158.35	142.28
10	16498.16	16619.71	16493.61	100.72	134.63	104.68
11	3800.91	3840.91	3800.91	117.10	124.38	123.17
12	19173.54	19307.39	19173.54	193.83	166.50	118.85
13	10518.12	12805.45	10518.12	72.10	107.15	81.27
14	2671.62	3187.25	2785.41	63.88	107.02	82.27
15	5341.43	5421.18	5341.43	142.70	156.28	121.82
16	3490.49	3737.16	3490.49	149.78	157.63	106.68
17	12962.94	13422.14	13288.29	176.78	170.93	139.45
18	11191.36	11234.27	11191.36	123.98	141.40	139.98
19	14378.82	17765.91	14378.82	72.47	108.48	82.72
20	10857.74	12245.92	10857.74	192.37	170.22	173.48
21	10365.74	11655.52	10365.74	226.23	154.17	154.82
22	5152.76	5152.76	5152.76	101.68	137.47	109.70
23	4819.96	5019.24	4819.96	123.53	166.12	135.13
24	10092.16	10092.16	10085.11	122.57	159.13	123.95
25	14006.79	15342.27	14006.79	195.35	182.38	150.72
26	3423.57	4072.85	3423.57	67.38	113.23	88.23
27	5985.66	5874.69	5922.94	241.33	178.43	166.50
28	1507.60	1717.56	1666.36	111.73	127.67	106.45
29	14593.35	14601.51	14601.51	338.37	156.13	167.85
30	3226.76	3386.04	3226.76	133.22	157.57	135.55
31	6575.36	6775.36	6575.36	135.85	215.90	115.98
32	11986.35	13270.12	11986.35	200.75	201.40	157.47
33	5416.50	5820.95	5416.50	225.33	169.63	161.72
34	8429.08	8809.18	8429.08	150.92	195.92	126.08
35	7260.49	8406.42	7260.49	205.15	198.90	171.17
36	4320.95	4476.25	4320.95	168.93	149.37	166.70
37	5713.95	5840.20	5713.95	207.02	145.70	136.62
38	3200.39	3200.39	3239.70	132.47	248.40	130.48
39	22341.30	25072.29	22602.71	210.00	155.43	170.33
40	5646.90	6128.15	5646.90	150.67	240.05	168.63
41	4407.56	4850.46	4803.27	167.10	218.47	153.62
42	4720.40	4846.65	4720.40	229.20	164.33	162.82
43	4440.08	4512.55	4512.55	189.02	151.88	170.40
44	4622.89	4723.66	4622.89	171.05	197.22	124.07
45	1937.37	1998.94	1932.82	130.45	179.63	136.30
46	1193.92	1843.20	1193.92	74.12	141.35	110.02
47	6995.89	8053.62	6995.89	235.15	228.80	213.82
48	4296.18	4441.75	4296.18	242.23	170.32	213.82
49	9077.56	9085.72	9085.72	203.40	194.98	172.23
50	12134.97	12396.38	12134.97	176.43	167.83	130.45

Table G.2. *Agent/Role Assignment Quality Results, Part 2*

Problem Specification Number	Minimum Cost Category 1 (dollars)	Minimum Cost Category 2 (dollars)	Minimum Cost Category 3 (dollars)	Runtime Category 1 (seconds)	Runtime Category 2 (seconds)	Runtime Category 3 (seconds)
51	12183.61	13561.28	13561.28	138.03	171.00	110.08
52	5154.23	5154.23	5154.23	135.97	166.17	112.87
53	16233.05	16526.14	16233.05	177.90	194.72	144.12
54	6222.48	7108.01	7117.02	105.78	138.82	121.57
55	10888.31	10888.31	10888.31	224.80	173.23	154.98
56	4423.57	4772.00	4423.57	140.75	181.47	121.10
57	4633.28	4720.29	4720.29	143.63	136.95	173.90
58	7214.25	8217.95	7214.25	139.07	145.93	149.88
59	12751.15	13044.24	13044.24	184.78	127.93	142.88
60	2401.43	2527.16	2527.16	124.53	128.43	128.05
61	9655.36	9855.36	9655.36	119.58	172.75	116.33
62	7448.16	7551.79	7420.71	128.67	192.92	177.88
63	10285.69	11146.15	10285.69	178.77	163.53	113.87
64	4771.00	5081.87	4771.00	192.57	134.50	193.80
65	14981.56	15127.46	15120.41	131.58	160.18	120.28
66	3708.13	3795.15	3708.13	155.77	156.43	168.47
67	7324.10	7450.35	7324.10	189.57	140.00	132.98
68	10844.70	12500.40	10844.70	236.40	138.93	153.78
69	6377.29	6450.38	6377.29	287.33	209.12	181.90
70	12240.56	12240.56	12240.56	198.53	175.93	157.25
71	5280.55	5375.31	5324.11	115.25	137.10	132.63
72	11659.57	11667.73	11667.73	328.70	172.97	171.42
73	2906.15	2866.84	2906.15	131.93	168.47	135.00
74	4238.48	4238.48	4238.48	193.30	190.08	122.53
75	15699.44	17819.11	16183.29	213.83	206.65	157.12
76	11826.90	13105.72	12042.68	311.82	142.77	161.28
77	14627.28	14627.28	14627.28	208.10	192.88	166.95
78	2704.83	2773.21	2704.83	181.60	188.07	156.17
79	7934.76	8083.20	7992.19	221.28	214.82	186.50
80	14047.61	16661.91	14329.37	144.23	168.67	127.70
81	9475.90	10540.52	9737.31	243.53	158.88	175.43
82	3409.66	3513.29	3409.66	151.58	198.22	145.70
83	11797.85	12090.94	11797.85	260.77	221.65	302.17
84	13992.03	13992.03	13992.03	260.10	212.80	204.55
85	4679.29	4876.56	4679.29	222.17	227.58	206.63
86	20835.11	20835.11	20835.11	270.95	213.42	247.23
87	17151.97	19475.84	17263.01	261.87	218.68	233.35
88	7215.44	7349.29	7215.44	255.37	227.72	182.77
89	11919.25	14497.41	12015.36	148.27	211.87	188.05
90	7240.51	7240.51	7240.51	236.87	201.22	332.92
91	4536.68	4616.43	4536.68	181.97	200.83	163.02
92	3772.06	4105.16	3948.98	167.00	215.10	138.28
93	10546.47	10605.54	10539.42	136.72	196.30	141.42
94	4548.88	4548.88	4548.88	176.93	243.05	141.93
95	4657.16	5321.42	5177.67	188.92	216.53	153.13
96	5499.93	5410.61	5410.61	176.08	149.63	436.40
97	8611.47	8959.90	8611.47	122.35	153.45	108.43
98	8808.08	9924.63	8808.08	139.83	131.05	105.33
99	10194.62	10843.90	10194.62	57.38	101.35	80.32
100	7496.07	7844.50	7496.07	112.13	140.93	96.27

Table G.3. *Agent/Role Assignment Runtime Results, Part 1*

Problem Specification Number	Number of Solutions Category 1	Number of Solutions Category 2	Number of Solutions Category 3	Runtime per Solution (seconds) Category 1	Runtime per Solution (seconds) Category 2	Runtime per Solution (seconds) Category 3
1	8	17	11	45.23	9.88	18.84
2	12	14	8	10.18	9.48	13.05
3	9	12	10	7.28	8.41	7.82
4	12	14	13	10.05	9.09	9.81
5	19	15	13	8.78	11.49	17.16
6	12	17	12	8.50	8.86	9.41
7	19	8	12	9.29	15.18	10.95
8	19	9	10	12.72	17.88	13.84
9	19	16	15	8.78	9.90	9.49
10	13	14	12	7.75	9.62	8.72
11	12	10	11	9.76	12.44	11.20
12	19	15	11	10.20	11.10	10.80
13	10	10	10	7.21	10.72	8.13
14	9	12	10	7.10	8.92	8.23
15	15	14	12	9.51	11.16	10.15
16	12	13	8	12.48	12.13	13.34
17	19	15	11	9.30	11.40	12.68
18	12	14	13	10.33	10.10	10.77
19	10	12	10	7.25	9.04	8.27
20	19	15	15	10.12	11.35	11.57
21	19	15	15	11.91	10.28	10.32
22	12	11	12	8.47	12.50	9.14
23	12	14	12	10.29	11.87	11.26
24	12	15	11	10.21	10.61	11.27
25	19	16	14	10.28	11.40	10.77
26	9	12	10	7.49	9.44	8.82
27	19	17	16	12.70	10.50	10.41
28	13	11	12	8.59	11.61	8.87
29	5	14	9	67.67	11.15	18.65
30	12	10	11	11.10	15.76	12.32
31	12	16	8	11.32	13.49	14.50
32	19	9	9	10.57	22.38	17.50
33	19	7	12	11.86	24.23	13.48
34	15	16	7	10.06	12.24	18.01
35	19	16	15	10.80	12.43	11.41
36	15	8	13	11.26	18.67	12.82
37	19	12	12	10.90	12.14	11.38
38	12	14	11	11.04	17.74	11.86
39	19	9	12	11.05	17.27	14.19
40	14	13	14	10.76	18.47	12.05
41	14	15	11	11.94	14.56	13.97
42	19	12	12	12.06	13.69	13.57
43	15	7	12	12.60	21.70	14.20
44	12	14	8	14.25	14.09	15.51
45	13	13	11	10.03	13.82	12.39
46	9	12	10	8.24	11.78	11.00
47	17	15	15	13.83	15.25	14.25
48	14	13	14	17.30	13.10	15.27
49	19	18	16	10.71	10.83	10.76
50	15	16	11	11.76	10.49	11.86

Table G.4. *Agent/Role Assignment Runtime Results, Part 2*

Problem Specification Number	Number of Solutions Category 1	Number of Solutions Category 2	Number of Solutions Category 3	Runtime per Solution (seconds) Category 1	Runtime per Solution (seconds) Category 2	Runtime per Solution (seconds) Category 3
51	15	17	7	9.20	10.06	15.73
52	15	16	6	9.06	10.39	18.81
53	19	9	9	9.36	21.64	16.01
54	13	14	13	8.14	9.92	9.35
55	19	9	10	11.83	19.25	15.50
56	15	16	9	9.38	11.34	13.46
57	14	11	15	10.26	12.45	11.59
58	14	13	14	9.93	11.23	10.71
59	19	8	10	9.73	15.99	14.29
60	12	10	12	10.38	12.84	10.67
61	12	16	8	9.97	10.80	14.54
62	12	9	8	10.72	21.44	22.24
63	19	13	10	9.41	12.58	11.39
64	19	7	13	10.14	19.21	14.91
65	12	15	11	10.97	10.68	10.93
66	15	14	14	10.38	11.17	12.03
67	19	12	12	9.98	11.67	11.08
68	19	8	12	12.44	17.37	12.81
69	19	16	14	15.12	13.07	12.99
70	19	9	10	10.45	19.55	15.73
71	13	11	13	8.87	12.46	10.20
72	4	15	10	82.18	11.53	17.14
73	12	14	9	10.99	12.03	15.00
74	15	16	6	12.89	11.88	20.42
75	19	16	11	11.25	12.92	14.28
76	19	8	12	16.41	17.85	13.44
77	19	15	14	10.95	12.86	11.93
78	12	10	12	15.13	18.81	13.01
79	19	16	15	11.65	13.43	12.43
80	14	11	12	10.30	15.33	10.64
81	19	8	12	12.82	19.86	14.62
82	12	14	10	12.63	14.16	14.57
83	19	9	9	13.72	24.63	33.57
84	19	15	14	13.69	14.19	14.61
85	17	15	15	13.07	15.17	13.78
86	19	9	10	14.26	23.71	24.72
87	19	9	13	13.78	24.30	17.95
88	19	14	11	13.44	16.27	16.62
89	12	15	14	12.36	14.12	13.43
90	19	9	10	12.47	22.36	33.29
91	15	14	12	12.13	14.35	13.59
92	15	16	7	11.13	13.44	19.75
93	13	15	11	10.52	13.09	12.86
94	15	17	7	11.80	14.30	20.28
95	15	16	9	12.59	13.53	17.01
96	18	14	12	9.78	10.69	36.37
97	13	14	9	9.41	10.96	12.05
98	13	13	7	10.76	10.08	15.05
99	8	12	10	7.17	8.45	8.03
100	13	14	9	8.63	10.07	10.70

REFERENCES

[Adler *et al.*, 1989] Adler, M. R., Davis, A. B., Weihmayer, R., and Worrest, R. W. Conflict-resolution strategies for non-hierarchical distributed agents. In Gasser, L. and Huhns, M., editors, *Distributed Artificial Intelligence, Volume 2*, Research Notes in Artificial Intelligence. Pitman, London, 1989.

[Barr and Feigenbaum, 1981] Barr, A. and Feigenbaum, E. A. *The Handbook of Artificial Intelligence, Volume 1.* William Kaufmann, Inc., 1981.

[Blackboard Technology Group, Inc., 1991] Blackboard Technology Group, Inc., Amherst, MA. *The GBB Version 2.0 Reference Manual*, January 1991.

[Bond and Gasser, 1988] Bond, A. and Gasser, L., editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann Publishers, 1988.

[Bond, 1989] Bond, A. H. The cooperation of experts in engineering design. In Gasser, L. and Huhns, M. N., editors, *Distributed Artificial Intelligence, Volume 2*, pages 463–484. Pitman/Morgan Kaufmann, 1989.

[Brodley, 1993] Brodley, C. E. Addressing the selective superiority problem: Automatic algorithm/model class selection. In *Tenth International Conference on Machine Learning*, pages 17–24, Amherst, MA, June 1993.

[Cammarata *et al.*, 1983] Cammarata, S., McArthur, D., and Steeb, R. Strategies of cooperation in distributed problem solving. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 767–770, Karlsruhe, Federal Republic of Germany, August 1983.

[Conry *et al.*, 1991] Conry, S., Kuwabara, K., Lesser, V., and Meyer, R. Multistage negotiation for distributed satisfaction. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1462–1477, November/December 1991.

[Corkill and Lesser, 1983] Corkill, D. D. and Lesser, V. R. The use of meta-level control for coordination in a distributed problem solving network. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 748–756, Karlsruhe, Germany, August 1983.

[Corkill, 1991] Corkill, D. D. Introducing blackboard systems. *AI Expert*, 6(1):40–47, September 1991.

[Davis and Smith, 1983] Davis, R. and Smith, R. G. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[de Bono, 1971] de Bono, E. *Lateral Thinking for Management, A handbook of creativity.* American Management Association, 1971.

[Decker and Lesser, 1992a] Decker, K. and Lesser, V. The analysis of quantitative coordination relationships. In *Workshop on Distributed Artificial Intelligence*, Michigan, February 1992.

[Decker and Lesser, 1992b] Decker, K. S. and Lesser, V. R. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(2):319–346, June 1992.

[Durfee and Lesser, 1988] Durfee, E. H. and Lesser, V. R. Predictability versus responsiveness: Coordinating problem solvers in dynamic domains. In *The Seventh National Conference on Artificial Intelligence*, pages 66–71, Saint Paul, Minnesota, August 1988.

[Durfee and Lesser, 1991] Durfee, E. H. and Lesser, V. R. Partial global planning: A coordination framework for distributed hypothesis formation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5):1167–1183, September/October 1991.

[Durfee and Montgomery, 1990] Durfee, E. H. and Montgomery, T. A. A hierarchical protocol for coordinating multiagent behaviors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 86–93, Boston, Massachusetts, August 1990.

[Durfee and Montgomery, 1991] Durfee, E. H. and Montgomery, T. Coordination as distributed search in a hierarchical behavior space. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1363–1378, November/December 1991.

[Durfee *et al.*, 1987] Durfee, E. H., Lesser, V. R., and Corkill, D. D. Coherent cooperation among communicating problem solvers. *IEEE Transactions on Computers*, 36(11):1275–1291, November 1987.

[Durfee, 1991] Durfee, E. H. The distributed artificial intelligence melting pot. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1301–1306, November/December 1991.

[Ephrati and Rosenschein, 1991] Ephrati, E. and Rosenschein, J. S. The Clarke Tax as a consensus mechanism among automated agents. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 173–184, Anaheim, California, July 1991.

[Ephrati and Rosenschein, 1992] Ephrati, E. and Rosenschein, J. S. Constrained intelligent action: Planning under the influence of a master agent. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 263–268, San Jose, California, July 1992.

[Erman *et al.*, 1980] Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, June 1980.

[Finin and Wiederhold, 1991] Finin, T. and Wiederhold, G. An overview of KQML: A knowledge query manipulation language. Technical report, Department of Computer Science, Stanford University, Stanford, CA, 1991.

[Fisher and Ury, 1981] Fisher, R. and Ury, W. *Getting to Yes: Negotiating Agreement without Giving In.* Houghton Mifflin, 1981.

[Fox *et al.*, 1982] Fox, M., Allen, B., and Strohm, G. Job-shop scheduling: an investigation in constraint-directed reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 155–158, Pittsburgh, Pennsylvania, August 1982.

[Fox *et al.*, 1989] Fox, M., Sadeh, N., and Baykan, C. Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 309–315, Detroit, Michigan, August 1989.

[Fox, 1987] Fox, M. S. *Constraint-Directed Search: A Case Study of Job Shop Scheduling.* Research Notes in Artificial Intelligence. Pitman Publishing, London, 1987.

[Gasser, 1991] Gasser, L. Social conceptions of knowledge and action: DAI foundations and open systems semantics. *Artificial Intelligence*, 47:107–138, 1991.

[Genesereth *et al.*, 1986] Genesereth, M. R., Ginsberg, M. L., and Rosenschein, J. S. Cooperation without communication. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 51–57, Philadelphia, PA, August 1986.

[Genesereth, 1990] Genesereth, M. Knowledge interchange format. Technical Report 90-4, Department of Computer Science, Stanford University, Stanford, CA, 1990.

[Guha and Lenat, 1990] Guha, R. and Lenat, D. B. CYC: A midterm report. *AI Magazine*, 11(3):32–59, Fall 1990.

[Hayes-Roth, 1985] Hayes-Roth, B. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, July 1985. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 503–540, Morgan Kaufmann, 1988.).

[Hewitt, 1986] Hewitt, C. Offices are open systems. *ACM Transactions on Office Information Systems*, 4(3):271–287, July 1986.

[Hewitt, 1991] Hewitt, C. Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, 47(1):79–106, January 1991.

[Hogg and Williams, 1993] Hogg, T. and Williams, C. P. Solving the really hard problems with cooperative search. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 231–236, Washington, DC, July 1993.

[Huhns and Bridgeland, 1991] Huhns, M. N. and Bridgeland, D. M. Multiagent truth maintenance. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1437–1445, November/December 1991.

[Jagannathan *et al.*, 1991] Jagannathan, V., Cleetus, J., Matsumoto, A., Kannan, R., and Lewis, J. Computer support for concurrent engineering. *Concurrent Engineering: Issues, Technology, and Practice*, pages 14–30, September 1991.

[Khedro and Genesereth, 1993] Khedro, T. and Genesereth, M. R. Progressive negotiation: A strategy for resolving conflicts in cooperative distributed multidisciplinary design. In *Proceedings of the Conflict Resolution Workshop, IJCAI-93*, Chambery, France, September 1993.

[Klein and Lu, 1989] Klein, M. and Lu, S. C.-Y. Conflict resolution in cooperative design. *International Journal of Artificial Intelligence in Engineering*, 4(4), 1989.

[Klein, 1991] Klein, M. Supporting conflict resolution in cooperative design systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1379–1390, November/December 1991.

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

[Kraus and Wilkenfeld, 1990] Kraus, S. and Wilkenfeld, J. The function of time in cooperative negotiations. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 179–184, Anaheim, California, July 1990.

[Laasri *et al.*, 1992] Laasri, B., Laasri, H., Lander, S., and Lesser, V. Toward a general model of intelligent negotiating agents. *The International Journal of Intelligent and Cooperative Information Systems*, 1(2):291–317, 1992.

[Lander and Lesser, 1991] Lander, S. E. and Lesser, V. R. Negotiated search: A framework for cooperative design. Technical Report 91-79, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, November 1991.

[Lander and Lesser, 1992a] Lander, S. E. and Lesser, V. R. Customizing distributed search among agents with heterogeneous knowledge. In *Proceedings of the First International Conference on Information and Knowledge Management*, pages 335–344, Baltimore, Maryland, November 1992.

[Lander and Lesser, 1992b] Lander, S. E. and Lesser, V. R. Negotiated search: Organizing cooperative search among heterogeneous expert agents. In *Proceedings of the Fifth International Symposium on Artificial Intelligence, Applications in Manufacturing and Robotics*, pages 351–358, Cancun, Mexico, December 1992.

[Lander and Lesser, 1993] Lander, S. E. and Lesser, V. R. Understanding the role of negotiation in distributed search among heterogeneous agents. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 438–444, Chambery, France, August/September 1993.

[Lander *et al.*, 1991a] Lander, S., Lesser, V. R., and Connell, M. E. Conflict resolution strategies for cooperating expert agents. In Deen, S., editor, *Cooperating Knowledge Based Systems 1990*, pages 183–198. Springer-Verlag, 1991.

[Lander *et al.*, 1991b] Lander, S. E., Lesser, V. R., and Connell, M. E. Knowledge-based conflict resolution for cooperation among expert agents. In Sriram, D., Logher, R., and Fukuda, S., editors, *Computer-Aided Cooperative Product Development*, pages 253–268. Springer-Verlag, 1991.

[Lee *et al.*, 1993] Lee, K.-C., Mansfield, W., and Sheth, A. A framework for controlling cooperative agents. *Computer*, 26(7):8–17, July 1993.

[Lesser and Corkill, 1983] Lesser, V. and Corkill, D. The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3):15–33, 1983. (also in *Blackboard Systems*, R. Engelmore and T. Morgan (eds.), pp 353-386, Addison-Wesley, 1988).

[Lesser *et al.*, 1993] Lesser, V., Nawab, H., Gallastegi, I., and Klassner, F. IPUS: An architecture for integrated signal processing and signal interpretation in complex environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 249–255, Washington, DC, July 1993.

[Lesser, 1990] Lesser, V. R. An overview of DAI: Viewing distributed AI as distributed search. *Journal of the Japanese Society for Artificial Intelligence*, 5(4):392–400, July 1990.

[Lesser, 1991] Lesser, V. R. A retrospective view of fa/c distributed problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1347–1362, November/December 1991.

[MacLeod and Moll, 1991] MacLeod, B. and Moll, R. OPL: An environment for posing and solving discrete optimization problems. Technical Report 91-20, Computer Science Department, University of Massachusetts, Amherst, MA, April 1991.

[Mammen and Lesser, 1992] Mammen, D. L. and Lesser, V. R. Using textures to control distributed problem solving. In *Working Notes from the AAAI Workshop on Cooperation among Heterogeneous Intelligent Systems, AAAI-92*, San Jose, California, July 1992.

[Marshall and et. al., 1982] Marshall, R. and et. al. Investigation of the Kansas City Hyatt Regency walkways collapse. Technical Report Technical Report Science Series 143, National Bureau of Standards, Washington, D.C., May 1982.

[Meunier, 1988] Meunier, K. L. Iterative respecification: A computational model for automating parametric mechanical system design. Master's thesis, University of Massachusetts, Amherst, Massachusetts, February 1988.

[Moehlman and Lesser, 1990] Moehlman, T. and Lesser, V. Cooperative planning and decentralized negotiation in multi-fireboss phoenix. In *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, Texas, November 1990.

[Neches *et al.*, 1991] Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. R. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[Newell, 1982] Newell, A. The knowledge level. *Artificial Intelligence*, 18(1):87–127, January 1982.

[Nii, 1986] Nii, H. P. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(2):38–53, Summer 1986.

[Nishibe *et al.*, 1992] Nishibe, Y., Kuwabara, K., and Ishida, T. Effects of heuristics in distributed constraint satisfaction: Towards satisficing algorithms. In *Workshop on Distributed Artificial Intelligence*, pages 285–302, Michigan, February 1992.

[Orelup *et al.*, 1988] Orelup, M., Dixon, J., Cohen, P., and Simmons, M. Dominic II: Meta-level control in iterative redesign. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 25–30, Saint Paul, Minnesota, August 1988.

[Pruitt, 1981] Pruitt, D. G. *Negotiation Behavior*. Academic Press, 1981.

[Reddy *et al.*, 1993] Reddy, Y., Srinivas, K., Jagannathan, V., and Karinthi, R. Computer support for concurrent engineering. *Computer*, 26(1):12–16, January 1993.

[Robinson, 1991] Robinson, W. N. A decision theoretic perspective of multiagent requirements negotiation. In *Proceedings of the AAAI Workshop on Cooperation Among Heterogeneous Intelligent Systems*, Anaheim, California, July 1991.

[Rosenschein and Genesereth, 1985] Rosenschein, J. S. and Genesereth, M. R. Deals among rational agents. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 91–99, Los Angeles, California, August 1985.

[Sandholm, 1993] Sandholm, T. An implementation of the contract net protocol based on marginal cost calculations. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 256–262, Washington, D.C., July 1993.

[Sathi and Fox, 1989] Sathi, A. and Fox, M. S. Constraint-directed negotiation of resource reallocations. In Gasser, L. and Huhns, M., editors, *Distributed Artificial Intelligence, Volume 2*, chapter 8, pages 163–193. Pitman Publishing, London, 1989.

[Sathi *et al.*, 1986] Sathi, A., Morton, T. E., and Roth, S. F. Callisto: An intelligent project management system. *AI Magazine*, 7(5):34–52, Winter 1986.

[Sen and Durfee, 1992] Sen, S. and Durfee, E. A formal analysis of communication and commitment in distributed meeting scheduling. In *Proceedings of the 10th Workshop on Distributed Artificial Intelligence*, Glen Arbor, Michigan, February 1992.

[Simon, 1969] Simon, H. A. *The Sciences of the Artificial.* The M.I.T. Press, 1969.

[Skinner, 1992] Skinner, J. M. *A Synergistic Approach to Reasoning.* PhD thesis, Dept. of Computer Science, University of New Mexico, Albuquerque, NM, May 1992.

[Sommerville, 1989] Sommerville, I. *Software Engineering, Third Edition.* Addison Wesley, 1989.

[Sriram *et al.*, 1991] Sriram, D., Logcher, R., and Fukuda, S., editors. *Computer-Aided Cooperative Product Development.* Lecture Notes in Computer Science, No. 492. Springer-Verlag, 1991.

[Sriram *et al.*, 1992] Sriram, D., Logcher, R., Groleau, N., and Cherneff, J. DICE: An object oriented programming environment for communication, coordination and control in computer aided engineering. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design (Volume III).* Academic Press, 1992.

[Sycara *et al.*, 1991] Sycara, K., Roth, S., Sadeh, N., and Fox, M. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1446–1461, November/December 1991.

[Sycara, 1985] Sycara, K. Arguments of persuasion in labour mediation. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 294–296, Los Angeles, California, 1985.

[Sycara, 1987] Sycara, E. P. *Resolving Adversarial Conflicts: An Approach Integrating Case-Based and Analytic Methods.* PhD thesis, Georgia Institute of Technology, Atlanta, Georgia, June 1987. Also published as Technical Report GIT-ICS-87/26.

[Sycara, 1988] Sycara, K. Resolving goal conflicts via negotiation. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 245–250, Saint Paul, Minnesota, August 1988.

[von Martial, 1992] von Martial, F. *Coordinating Plans of Autonomous Agents.* Lecture Notes in Artificial Intelligence, Springer-Verlag, 1992.

[Werkman, 1992] Werkman, K. Multiple agent cooperative design evaluation using negotiation. In Gero, J., editor, *Artificial Intelligence in Design '92*, pages 161–180. Kluwer Academic Publishers, 1992.

[Wiederhold, 1992] Wiederhold, G. Intelligent integration of diverse information. In *Proceedings of the ISMM International Conference on Information and Knowledge Management*, pages 1–7, Baltimore, MD, November 1992.

[Winner, 1988] Winner, R. The role of concurrent engineering in weapons system acquisition. Technical Report IDA-R-338, Institute for Defense Analysis, Alexandria, VA, December 1988.

[Yokoo *et al.*, 1992] Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992.

[Zlotkin and Rosenschein, 1989] Zlotkin, G. and Rosenschein, J. S. Negotiation and task sharing among autonomous agents in cooperative domains. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 912–917, Detroit, MI, August 1989.

[Zlotkin and Rosenschein, 1990] Zlotkin, G. and Rosenschein, J. S. Negotiation and conflict resolution in non-cooperative domains. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 100–105, Boston, Massachusetts, July 1990.

[Zlotkin and Rosenschein, 1991] Zlotkin, G. and Rosenschein, J. S. Cooperation and conflict resolution via negotiation among autonomous agents in noncooperative domains. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1317–1324, November/December 1991.