A Study of Autonomous Agents in Decision Support Systems

Traci J. Hess

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Management Science and Information Technology

Loren Paul Rees, Chairman
Lance A. Matheson
Cliff T. Ragsdale
Terry R. Rakes
Robert T. Sumichrast

May 5, 1999
Blacksburg, Virginia Tech,

Keywords: Software Agents, Decision Support Systems, Model Management, Planning

Copyright 1999, Traci J. Hess
A Study of Autonomous Agents in Decision Support Systems

Traci J. Hess

(ABSTRACT)

Software agents have been heralded as the most important emerging technology of the decade. As software development firms eagerly attempt to integrate these autonomous programs into their products, researchers attempt to define the concept of agency and to develop architectures that will improve agent capabilities. Decision Support System (DSS) researchers have been eager to integrate agents into their applications, and exploratory works in which agents have been used within a DSS have been documented. This dissertation attempts to further this exploration by studying the agent features and underlying architectures that can lead to the successful integration of agents in DSS.

This exploration is carried out in three parts. In the first part, a review of the relevant research streams is provided. The history and current status of software agents is first discussed. Similarly, a historical and current view of DSS research is provided. Lastly, a historical and tutorial-type of discussion is provided on the topic of Artificial Intelligence (AI) planning. This review of the relevant literature provides a general background for the conceptual analyses and implementations that are carried out in the next two sections.

In the second part, the literature on software agents is synthesized to develop a definition of agency applicable to DSS. Using this definition, an agent-integrated DSS that supports variance-analysis is designed and developed. Following this implementation, a general framework for agent-enabling DSS is suggested. The use of this framework promises to raise some DSS to a new level of capability whereby “what-if” systems are transformed into real-time, proactive systems.

The third part utilizes this general framework to agent-enable a corporate-planning system DSS and extends the framework in the second section through the introduction of an automated-planning agent. The agent uses AI planning to generate decision-making alternatives, providing a means to integrate and sequence the models in the DSS. The architecture used to support this planning agent is described. This new kind of DSS enables not only the monitoring of goals, but also the maintenance of these goals through agent-generated plans.

The conclusion summarizes the contributions of this work and outlines in considerable detail potential research opportunities in the realm of software agents, DSS, and planning.
# TABLE OF CONTENTS

Chapter 1:

**INTRODUCTION** ................................................................. 1
  Software Agents .................................................................. 1
  Software Agents in Decision Support Systems .................... 1
  Architectures for Implementing Software Agents ................ 2
    Deliberative Architectures .............................................. 2
    Reactive Architectures .................................................. 2
    Integrated Architectures ............................................... 3
  Purpose of Research ...................................................... 3
  Research Methodology ................................................... 4
  Scope and Limitations .................................................... 4
  Plan of Presentation ...................................................... 5

Chapter 2:

**LITERATURE REVIEW** ...................................................... 6
  Introduction ........................................................................ 6
  Software Agents ............................................................ 6
    Major Agent-Related Research Streams ......................... 6
  Software Agent Features ............................................... 7
    Autonomy ....................................................................... 7
    Reactivity ..................................................................... 7
    Persistence .................................................................... 7
    Goal Orientedness .......................................................... 7
    Mobile Agents and Distributed Processing ..................... 8
  Agent Communication and Interaction .............................. 8
  Intelligent Agents ........................................................... 9
  Benefits of Using Software Agents ................................... 10
  Decision Support Systems ............................................... 11
    The Components of DSS .................................................. 11
      The Dialog Generation Management System ................. 11
      The Database Management System .............................. 13
      The Model Base Management System ............................ 13
  Current DSS Research .................................................... 13
  Planning ............................................................................ 14
    A General Description of Planning .................................. 14
    State-Space Planners ..................................................... 17
    Partial Order Planners ................................................... 17
# TABLE OF CONTENTS

Chapter 3:

**USING AUTONOMOUS SOFTWARE AGENTS TO CREATE THE NEXT GENERATION OF DECISION SUPPORT SYSTEMS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>21</td>
</tr>
<tr>
<td>An Overview of Decision Support Systems</td>
<td>21</td>
</tr>
<tr>
<td>The Use of Software Agents in a DSS</td>
<td>21</td>
</tr>
<tr>
<td>Software-Agent Definition</td>
<td>22</td>
</tr>
<tr>
<td>Problems with Definitions in the Literature</td>
<td>22</td>
</tr>
<tr>
<td>Resolving Definitional Problems</td>
<td>22</td>
</tr>
<tr>
<td>Reference Point</td>
<td>23</td>
</tr>
<tr>
<td>Differentiation</td>
<td>23</td>
</tr>
<tr>
<td>Our Definition</td>
<td>26</td>
</tr>
<tr>
<td>Homeostatic Goal(s)</td>
<td>26</td>
</tr>
<tr>
<td>Persistence</td>
<td>27</td>
</tr>
<tr>
<td>Reactivity</td>
<td>27</td>
</tr>
<tr>
<td>Empowerment</td>
<td>28</td>
</tr>
<tr>
<td>Mobility</td>
<td>28</td>
</tr>
<tr>
<td>Intelligence</td>
<td>29</td>
</tr>
<tr>
<td>Interactivity (Communicative Ability)</td>
<td>29</td>
</tr>
<tr>
<td>An Implementation of an Agent-Integrated DSS</td>
<td>30</td>
</tr>
<tr>
<td>Holsapple and Whinston’s Variance-Analysis DSS</td>
<td>30</td>
</tr>
<tr>
<td>Agent Integration</td>
<td>31</td>
</tr>
<tr>
<td>Agent Benefits in a DSS</td>
<td>39</td>
</tr>
<tr>
<td>General Benefits</td>
<td>39</td>
</tr>
<tr>
<td>Specific Benefits of an Agent-Integrated DSS</td>
<td>39</td>
</tr>
<tr>
<td>A Framework for an Agent-Enabled DSS</td>
<td>41</td>
</tr>
<tr>
<td>Conclusions and Future Work</td>
<td>45</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

Chapter 4:

PLANNING IN DECISION SUPPORT SYSTEMS USING AUTONOMOUS SOFTWARE AGENTS ........................................... 47

- Introduction ........................................................................... 47
- MBMS .................................................................................. 47
- A Review of Software Agents and Planning ....................... 48
  - Using Agents for Decision Support ............................... 49
    - Agent Definition ........................................................... 49
    - Agent Framework for DSS ............................................. 50
  - Using AI Planning to Generate Alternatives .................. 50
    - Using Planning to Generate Decision-Making Alternatives 53
- DSS Planning Implementation ........................................... 53
  - Profit-Monitoring DSS .................................................... 53
  - Planning Agent Integration ............................................. 55
- Benefits .............................................................................. 65
  - Planning Agent Architecture ......................................... 66
  - Planning-Agent Context and Framework ..................... 66
  - Adaptation of Planning to Non-Physical Domains .......... 68
  - Using Agents to Convey Planning Abilities .................. 70
- Future Research and Conclusions .................................... 70

Chapter 5:

CONCLUSIONS AND FUTURE WORK ......................................... 72

- Summary of Contributions ................................................ 72
  - Agent Theory and Framework ........................................ 72
  - Automated Planning Agents ......................................... 73
- Future Research ............................................................... 73
  - General Avenues of Research ...................................... 73
  - Immediate Research Plan ............................................. 74
    - A Deceptive, Electronic Commerce Agent with Multiple Conflicting Goals ... 75
    - Employing Agents in Data Warehousing Decision Support Environment .... 75

REFERENCES ........................................................................ 76

APPENDIX A ........................................................................ 81

APPENDIX B ........................................................................ 86

VITA ................................................................................... 94
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Dialog-Data-Models Paradigm</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Representation of Initial State and Goal State in Blocks World</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Example Operator in a Blocks World</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of a State-Space Search Path in a Blocks World</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>An Example of a Partial Plan Search Space in a Blocks World</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>The Empowerment of a Software Agent</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Some of the Agent Activity in the Agent-Enhanced Holsapple and Whinston Variance-Analysis DSS</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>The MBMS Excel (Solver) Solution to the Linear Programming Problem</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>The Agent-Enhanced Holsapple and Whinston Variance-Analysis DSS</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>The Agent-Enhanced General DSS Framework</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>The Agent-Enhanced General DSS Framework</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Representation of Initial State and Goal State in Blocks World</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>Example Operator in a Blocks World</td>
<td>54</td>
</tr>
<tr>
<td>4.4</td>
<td>The Agents Used in the Profit-Monitoring DSS</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>Alternatives Generated by the Planning Agent</td>
<td>59</td>
</tr>
<tr>
<td>4.6</td>
<td>Results from the Excel Linear Programming Model</td>
<td>64</td>
</tr>
<tr>
<td>4.7</td>
<td>The Planning Agent in the Agent-Integrated DSS Framework</td>
<td>67</td>
</tr>
<tr>
<td>4.8</td>
<td>The Planning Agent Architecture</td>
<td>69</td>
</tr>
<tr>
<td>B.1</td>
<td>The DWDS Environment</td>
<td>87</td>
</tr>
<tr>
<td>B.2</td>
<td>The Flow of Data in a Data Warehouse</td>
<td>88</td>
</tr>
<tr>
<td>B.3</td>
<td>Visual Architecture of the Agent-Integrated DWDS</td>
<td>90</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 3.1: Agent Types by Placement in DSS Domain ........................................ 33
Table 3.2: Example Agents Utilized in the Extension to the Holsapple and
          Whinston (1996) Manufacturing-Firm DSS ........................................ 34
Table 3.3: Benefits from the Empowering Characteristics of Agents in the Extension
          to the Holsapple and Whinston Manufacturing-Firm DSS ...................... 35
Table 4.1: Profit Formula Used in Profit-Monitoring DSS ............................... 56
Table 4.2: Some Facts, Actions, and Rules (specified in Prolog) Used by the
          Planning Agent .................................................................................. 60
Table 4.3: The Ordered Action Sets and Three Plans Developed ...................... 60
Table A.1: The Operators for Each Goal of the Malicious Agent ....................... 83
CHAPTER 1
INTRODUCTION

SOFTWARE AGENTS
Intelligent computers that can take instruction and carry out tasks on our behalf have been anticipated since the first computers were created. Now that personal computers and software applications have become an inseparable part of our everyday lives, users await the day when they can delegate tasks to computer-based agents. The process of replicating human-like behavior in computers, however, has proven to be a daunting task. Research in artificial intelligence (AI) has been on-going for over three decades and, in general, has just now reached the adolescent phase. This experience has taught us that the mechanisms underlying human behavior are extremely complex and not easily reproduced.

In simple terms, software agents are computer programs that perform tasks for users. These computing entities have received a great deal of press over the past five years and have been heralded as a new, enabling technology with enormous potential. It would be more accurate however, to describe agents as an emerging technology instead of a new technology, as AI researchers have been studying agents for over thirty years. As an emerging technology, software agents have not been rigorously or even concisely defined. While new releases of software applications claim to be agent-enabled, researchers continue to ponder what features differentiate agents from regular computer programs and make them better able to assist us.

Of primary interest to agent researchers is how to endow these software programs with human-like behaviors and what behaviors will make these agents most effective in carrying out actions on our behalf. Agent theories attempt to prescribe what behaviors will enable a program to exhibit agency, while agent architectures describe the AI mechanisms used to implement these behaviors. Research on agent theories and architectures is still largely exploratory, and no single theory or architecture has gained wide-spread acceptance. One fundamental obstacle in the advancement of such theories and architectures is the wide variation in the use of agents within different research disciplines. The various interpretations of agents and agency within different disciplines prevent the acceptance of a universal set of theories and architectures.

SOFTWARE AGENTS IN DECISION SUPPORT SYSTEMS
Decision Support System (DSS) researchers have been quick to experiment with software agents. Implementations of agents within DSS applications have begun to surface in DSS-oriented journals. Given the overall lack of agent theories and architectures, DSS researchers have had to proceed in this experimentation with little conceptual guidance on how and when to use software agents. Similarly, there has been no detailed discussion of why this emerging technology is appropriate or useful for DSS. The development of an agent theory and agent architecture(s) directed toward the realm of DSS research would thus be a timely topic that would address the current gap in the literature.
ARCHITECTURES FOR IMPLEMENTING SOFTWARE AGENTS

An agent architecture is a particular design or methodology for constructing an agent. Wooldridge and Jennings refer to an agent architecture as a software engineering model of an agent (1995). Using these guidelines, an architecture is a collection of software modules that implement the desired features of an agent in accordance with a theory of agency. It is this collection of software modules that enables the agent to reason about or select actions and react to changes in its environment. There are two general design approaches for agent architectures, deliberative, or classical planning, and reactive.

**Deliberative Architectures**

Deliberative architectures follow a symbolic AI approach and typically include some form of planning. Planning is the process of selecting a series of actions that, when executed, will achieve some goal. Classical planning is simply a search through a space of actions for a feasible sequence of actions that can achieve a goal. Planning has been described as automated programming (Wooldridge and Jennings, 1995) and thus provides a useful mechanism for enabling an agent to pursue a goal without significant assistance from the user. Restated, a planning architecture is a mechanism that enables an agent to deliberate on a course of action to take based upon the initial state of its environment and the actions available to it.

As a search-based process, planning is resource intensive and can be ineffective for time-constrained problems of moderate size. In a dynamic environment, where quick responses to changes in the environment are required, a purely deliberative agent can be quite ineffective. In addition, axiomatizing the problem space (i.e. representing the problem space in the symbolic notation that a planner can interpret), can be cumbersome and time consuming.

**Reactive Architectures**

Reactive architectures do not follow the symbolic AI approach and have in fact been developed by researchers seeking an alternative to the symbolic paradigm (Brooks, 1986; Agre and Chapman, 1987). Instead of representing the problem space in symbolic logic, reactive architectures hard-code the domain characteristics, creating digital circuits. These hard-coded circuits are structured into a hierarchy that enables quick responses to changes in the environment and thus do not suffer from the inefficiencies of deliberative architectures. When a particular event takes place, a specific circuit will execute.

While reactive architectures respond efficiently to exogenous events, the addition of new agent actions as responses to these events requires new circuits to be created and compiled. Conversely, in a planning or deliberative architecture, a new agent action would be represented in symbolic logic and added to the search space. The planning mechanism itself would not require any changes.

Due to the lack of separation between domain knowledge and the selection mechanism in a reactive architecture, new agent architectures must be built and tested from scratch. In a deliberative architecture, however, the reasoning or planning mechanism can be reused and only
the domain knowledge must be developed. The formal logic used in deliberative architectures also provides a common methodology or approach to specifying domain knowledge that is lacking in reactive architectures.

**Integrated Architectures**

An integrated architecture that includes both deliberative and reactive components can support a more autonomous agent than either architecture can support alone. An integrated architecture provides a mechanism that enables an agent to both deliberate and respond efficiently to exogenous events. While several integrated or hybrid agent architectures have been designed, most have been implemented in a spatial problem environment with discrete actions (operators). A spatial environment is one in which the agent actions are physical in nature, such as a robot delivering mail in a building. An open research question in this area is how to utilize hybrid or deliberative architectures in non-spatial domains (e.g., many business domains do not have spatial characteristics) in which continuous actions are prevalent.

**PURPOSE OF RESEARCH**

The purpose of this stream of research is to clarify the concept of software agents with respect to DSS and to explore agent architectures that are suitable for DSS applications. Currently, there is not a generally accepted definition of agency in the literature, and there is no designation of fundamental agent features to guide DSS developers in differentiating software agents from the typical computer program. As a result, the term software agent is subject to frequent misuse and overuse. This dissertation research synthesizes the views of agency in the literature and sets forth an agent theory for DSS. Along with a definition of agency, a classification of essential and empowering agent features is also provided.

Previous research on deliberative or hybrid agent architectures has typically focused on simple, artificial environments where the problem space bears little resemblance to a business domain. Such architectures have been implemented in games and spatial problems that are not applicable to many of the abstract problems encountered in business. A long-term goal for this stream of research is to improve the state of knowledge on agent architectures that can be successfully employed within DSS. The short-term goal that this dissertation research addresses is the exploration of an agent-integrated framework for DSS and a deliberative architecture for an automated-planning agent.

In this dissertation, a framework for a cooperative community of software agents within a DSS is designed and then constructed. The agents described in this framework each exhibit the requirements set forth in the agent theory previously described. This agent-integrated DSS framework features agents that employ more reactive abilities, serving a specific purpose within the DSS domain and responding efficiently to changes in their specific part of the domain.

The framework is then extended to include an agent with more deliberative abilities. This agent, a planning agent, evaluates changes in the DSS environment as detected and communicated by the community of agents. The planning agent uses a planning mechanism to deliberate, generating decision-making alternatives in response to significant changes. This implementation
demonstrates the feasibility of using planning to provide deliberative abilities within an agent architecture in a business domain.

**RESEARCH METHODOLOGY**

Software agents constitute an enabling technology that adds new features to or significantly enhances existing features in software applications. As an emerging technology that utilizes various artificial intelligence mechanisms, research on the topic is largely exploratory in nature. The research described herein is thus also largely exploratory. After synthesizing the literature on the various features that agents can and should exhibit, the current research sets forth a definition of software agents (an agent theory) applicable to the use of agents in DSS. A proof-of-concept implementation containing several different types of agents that meet the requirements of the definition is then designed and built. The agents used in this implementation establish a basic infrastructure for employing agents in DSS and are primarily reactive in nature. The agents respond efficiently to changes in the DSS environment and their deliberative abilities are fairly minimal.

Based upon a review of the relevant literature and the experience gained in building an agent-integrated DSS, a general framework for using agents within a DSS is then proposed. The framework includes several different types of agents that would be generally useful in most DSS, regardless of the domain.

In order to explore the use of more deliberative agent architectures within DSS, a second agent-integrated DSS implementation was designed and built. This DSS featured the same agent-framework discussed previously and introduced a planning agent with deliberative abilities. The planning abilities were achieved by embedding a partial-order planner within the agent. Based upon a review of the literature and the experience gained in designing and developing this agent, a general architecture for a planning agent is suggested.

**SCOPE AND LIMITATIONS**

The scope of this research includes an examination of agent features and the design and implementation of an agent framework and architecture to support the integration of agents in DSS. Agent usability issues and the selection of analysis and designed methodologies that are appropriate for implementing the architecture are not addressed.

Limitations of the research include the domain specificity of the implementations. The two implementations built were designed for specific types of DSS, a variance-analysis DSS and a profit-monitoring DSS. Generalization of the architecture across various problem domains may be inappropriate due to the wide range of differing requirements in each domain. While realistic business problems were selected for the implementations, simplifying assumptions were needed to constrain implementation size. These simplifications could decrease the relevance of implementing a successful, agent framework and architecture as the implementation developed may not accurately represent the problem space.
PLAN OF PRESENTATION

The following chapter, Chapter 2, surveys the literature and at times serves as a tutorial for the topics of software agents, DSS, and AI planning. This chapter provides the necessary background information for the subsequent implementations. Chapter 3 synthesizes the literature on software agents and DSS, providing a definition of software agents. This chapter then describes an implementation of agent-integrated DSS in which the agents meet the requirements of the agent definition and suggests a general framework for using agents in DSS. In the following chapter, Chapter 4, an implementation of a planning-agent within an agent-integrated DSS is described and a general planning-agent architecture is suggested. A discussion of how planning might be adapted for use in non-spatial environments is also provided. Chapter 5 summarizes the contributions of the agent definition, the agent-integrated DSS framework, the planning-agent architecture, and the two specific DSS implementations. Future research plans are also presented in some detail.
CHAPTER 2
LITERATURE REVIEW

INTRODUCTION

Three streams of research support the exploration and implementations of software agents in DSS pursued within this document: (1) software agents, (2), decision support systems, and (3) artificial intelligence planning. This literature review is thus composed of three sections that provide a general summary of these respective research areas. Within each section, a brief history of the research topic is provided, fundamental concepts relating to the topic are discussed, and then aspects of the topic that relate specifically to the intersection of software agents, DSS, and planning are addressed.

SOFTWARE AGENTS

Major Agent-Related Research Streams

Software agents were born out of the concept of robots within the artificial intelligence (AI) community. Instead of a physical entity carrying out manual tasks on behalf of a user, software agents are programs that perform computing tasks. For over twenty years, the AI community has investigated agents in terms of entities that exhibit intelligent behavior. Research in the realm of Distributed Artificial Intelligence (DAI) has investigated concurrent and distributed intelligent processing and provided a framework for the agent systems being implemented today. Within DAI, two distinct streams of research have provided different perspectives on distributed intelligent agents. The Distributed Problem Solving (DPS) stream focused on problem resolution through task decomposition and delegation to distributed nodes or agents (Bond & Gasser, 1988). DPS emphasized problem solving and did not focus on the composition or individual behavior of the actual nodes. The second stream, Multiagent Systems (MAS), viewed agents as autonomous entities and focused on how these entities could coordinate to solve problems (Bond & Gasser, 1988).

These differing views on the concept of software agents, along with others, are present in a newer stream of software agent research that has developed in the 1990’s. This new stream of research is multi-disciplinary and has shifted the focus from how intelligent software entities could interact to solve problems to creating entities that perform a useful task (Bradshaw, 1997; Nwana, 1996). These entities are not always intelligent, or that useful, but the change in emphasis from analyzing to implementing has brought a great deal of attention to the topic of software agents and has renewed interest in AI applications.

Software Agent Features

The abstract nature of software agents and the potential functionality that can be incorporated into one have made it difficult for a single definition of a software agent to be widely accepted (Foner, 1998; Franklin & Graesser, 1996; Kautz, Selman, Coen, Ketchpel, & Ramming, 1994).
A software agent is a program with characteristics that distinguish it from a standard subroutine or software application. Several key characteristics are common among most descriptions of software agents and offer minimal requirements for software to be classified as agent-like. These characteristics are autonomy, reactivity, persistence and goal-orientedness. Other characteristics such as mobility, interactivity and intelligence are also frequently associated with software agents.

**Autonomy.** Autonomy is a characteristic that appears to be fundamental to most definitions of software agents. The interpretations of autonomy with regard to software agents vary slightly among agent researchers. Foner requires that an agent be able to “pursue an agenda independently from its user” and take “preemptive or independent actions that will eventually benefit the user” (1998, p.1). Franklin and Graesser have a less restrictive view of autonomy, requiring agents to “exercise control over their own actions” (1996, p.6). Using the less restrictive definition, a software agent could be a program executed initially by the user which would then carry out its purpose independently.

**Reactivity.** Reactivity is another key characteristic of agent behavior. Reactivity has been defined as an agent’s ability to perceive the environment and respond to changes in that environment in a timely fashion (Wooldridge & Jennings, 1995). A restriction to this reactivity is that software agents are designed to carry out objectives in a specific environment. Within a defined domain, agents are given the authority to carry out a task and are required to react appropriately to changes in this domain or context. An agent must be assigned to a specific environment to exercise this reactivity, and as Franklin and Graesser stress, an agent may cease to be an agent when it is outside of its environment (1996). In reacting to the environment, the agent extends its autonomy by carrying out actions in response to context changes, without intervention from the user.

**Persistence.** The added requirement of persistence further restricts the set of software programs whose behavior qualifies as agent-like. Software agents must run or execute continuously and are frequently referred to as continuously perceiving their environment. The length of time for which a software agent may persist varies depending upon the task that the agent was assigned to carry out. Persistence is thus relative, as an agent will persist over a time period that is long relative to the duration of the assigned task or goal. This feature is often implemented by providing the agent with its own thread of execution and using a loop to keep the agent running.

**Goal-orientedness.** Agents can be further defined by the requirement that they “realize a set of goals or tasks for which they were designed” (Maes, 1995, p.108). Their reactivity must be tempered such that agents are not continuously running programs that simply react to changes in the environment. Agents should be single-minded and proactive in carrying out their assigned task. Agent researchers also refer to this characteristic as pro-activeness and describe it as an agent’s ability to “not simply act in response to their environment…” but “…to exhibit goal-directed behavior by taking the initiative” (Wooldridge & Jennings, 1995, p. 5).

The concept of goal-orientedness is further refined in an article on autonomous systems by Covrigaru and Lindsay (1991). They distinguish between two types of goals, homeostatic and achievable goals. Homeostatic goals are continuously pursued. These goals “do not terminate
when the system is in one of the final states; when changes occur...activity to reacheive the final state is reinitiated (Covrigaru & Lindsay, 1991, p. 116). Achievable goals are not continuously pursued; the achievement of the final state marks the termination of the goal (Covrigaru & Lindsay, 1991). For an agent to act autonomously, it must pursue homeostatic goals rather than achievable goals.

**Mobile Agents and Distributed Processing.** Agent mobility is defined fairly precisely as compared to other agent features. Mobility is achieved by dispatching the agent to a remote location. In computational terms, a mobile agent is an agent software program that is passed, as a whole, to a remote location where it executes. The entire program is passed to the remote server, including its “code, data, execution state and travel itinerary” (Lange & Change, 1996, p.1). An agent that passes messages to a remote location is merely communicating and does not qualify as a mobile agent. A mobile agent inherently demonstrates at least a moderate level of autonomy as it executes independently of the host that created it. Mobile agents can generally move to multiple remote sites by either carrying an itinerary with them or by being dispatched to another site by an agent server or another agent. Persistence enables them to complete their tasks when remote sites are unavailable as they can wait at their current location until the site is accessible.

Support and security issues are substantially magnified in a mobile agent environment. Remote sites must support the programming language in which the agent was deployed, and agents executing at remote sites need access to the codebase at the site. Security issues that must be addressed within an agent system include designating which agents should be given access to the site and which files should be made accessible to these agents. The prerequisite agent support required at remote sites provides a functional barrier to potentially malicious agents but at the same time presents a hurdle to the widespread use and interaction of agents.

While there is currently no standard model for agent mobility and current implementations are application specific, mobile agents can still serve as an integration tool between heterogeneous software applications. Mobile agents can facilitate the exchange of data and processes among different applications and inherently provide distributed processing due to their ability to execute on remote sites. In an agent-enabled network, each node can serve and support agents, providing peer-to-peer functionality. Client stations can host mobile agent executions reducing network traffic and server overload.

**Agent Communication and Interaction.** Agent communication is commonly defined as the ability of the agent to communicate with other agents or with humans (Franklin & Graesser, 1996). While communication is not a required agent feature, an isolated agent is inherently limited in its abilities. An agent that can exchange information with other agents can be more efficient through cooperation and delegation. A communicative or social agent could save itself a journey to several remote sites by communicating with an agent that already knows the information it is seeking. An agent could additionally gain a great deal of efficiency by spawning several new agents and instructing them to accomplish tasks in parallel.

Agent communication is implemented through message passing but, as with other agent features, there exists a wide range of accepted agent communicative abilities. Message passing of low
level data types without semantics has minimal impact on an agent’s efficiency in carrying out its task. While agents have been created in various programming languages, most of the agent implementations and toolkits were developed in Java. Despite this common programming environment, agent developers would implement communication differently causing integration and compatibility problems. An agent communication language (ACL) has been developed by the Knowledge Sharing Effort, a joint initiative of several research groups, to provide a means for communication among agents developed in different programming environments for different purposes or domains (Neches, Fikes, Finin, Bruber, Patil, Senator, & Swartout, 1991). ACL creates a common semantic base and prevents the use of synonyms to describe similar facts through its three components, a “vocabulary, an inner language called Knowledge Interchange Format (KIF) and an outer language called the Knowledge Query and Manipulation Language (KQML)” (Genesereth & Ketchpel, 1994, p.49). This language has been adopted by numerous agent implementations and is especially well received in agent-based software engineering” (Genesereth & Ketchpel, 1994).

Agent interaction takes place when software agents exchange messages. This interaction can range from a master slave relationship among agents, where the master creates the slave and assigns it a task to carry out, to heterogeneous agents created by competing organizations that share misleading information with one another. Within the Artificial Intelligence (AI) community, there has been a great deal of research over the past ten years on the general topic of agent interaction or negotiation. This research “marks the intersection of economics and distributed AI (DAI) … [where] a number of researchers in DAI are using tools developed in economics and game theory to evaluate multiagent interactions” (Genesereth & Ketchpel, 1994, p.53). For example, Ephrati and Rosenschein designed a multiagent, meeting scheduling system with a built-in mechanism that causes agents to disclose the true preferences of their users (Ephrati & Rosenschein, 1996).

**Intelligent Agents.** Intelligence is a more difficult agent characteristic to define. IBM generally describes intelligence with regard to its own agents as the “degree of reasoning and learned behavior: the agent’s ability to accept the user’s statement of goals and carry out the task delegated to it” (Gilbert & Janca, 1997). Imam and Kodratoff go a little further when they summarize a AAAI (American Association for Artificial Intelligence) Workshop effort to define the term (1997). They describe an intelligent agent as a “system or machine that utilizes inferential or complex computational methodologies to perform the set of tasks of interest to the user.” (1997, p. 76). The notion of inference or (machine) learning or reasoning is implicit or explicit in both definitions. Intelligence is an enabling feature that allows an agent to pursue its goals more efficiently with less assistance from the user or designer.

Agents with varying types of intelligence have been implemented. In designing software agents that serve as an interface between the user and the computer, Maes developed agents that begin their life with a knowledge base of rules specific to the domain of the agent (Maes, 1994). Once the agent is initialized, it watches users’ actions and learns their habits and preferences. The agent then adds this information to its knowledge base in the form of rules to reflect the individual preferences of the user. With this form of intelligence, a well-trained agent can independently carry out tasks for the user and react to its environment appropriately.
Lashkari, Metral, and Maes extended this model of an interface agent by allowing the agent to learn from other agents (1994). Using the domain of e-mail, a newly created agent is unable to assist the user with the filtering and filing of e-mail until it learns the preferences of its user. The agent may need to watch the user’s actions in dealing with over 100 e-mail messages before it is reasonably confident in its recommendations to the user (1994). By interacting with another user’s previously trained e-mail agent, the required training time for the new agent can be reduced. In Maes’ study, a trained agent shared information with regard to how its user dealt with e-mail from a specific source. The newer agent was able to use this knowledge when its user received e-mail from the same type of source and was able to reach a reasonable confidence level almost immediately.

Intelligent agents can utilize any type of intelligence architecture depending on the task they have been assigned to perform. Agents can employ case-based reasoning, other artificial intelligence approaches, operations research methods, or any number of specific algorithms to help them better and more independently accomplish their task. While these architectures are simply tools that facilitate problem solving from an agent perspective, they enable the agent to exhibit more autonomy by requiring less user interaction. These tools provide the agent with a source of additional guidance, reducing the need for intervention from the user. The use of such well-known, rational techniques should also increase the user’s confidence in the agent’s actions or recommendations. While they may start their life with a set of rules to follow in pursuit of an assigned goal, the agent’s ability to specify how they will reach the goal and how they will react to changes in their environment determines the degree of intelligence they exhibit.

**Benefits of Using Software Agents**

Software agents, as robust autonomous programs, can provide an abstraction for the increasing complexity of computing. Agents can provide this abstraction because they have more stringent requirements (i.e., the essential features) than typical programs and are thus more independent and reliable. A user or developer can delegate a task to an agent and not concern himself or herself with how or whether the agent will accomplish the task. The reactive, persistent nature of the agent should ensure that it pursues its goals zealously and that it updates the user or other programs on the status of its goals. The use of homeostatic goals provides long-term satisfaction of developers’ and users’ needs.

Agents can serve as an abstraction in two important areas, interoperability and user interfaces (Bradshaw 1997). As an interoperability abstraction, agents are used to integrate between heterogeneous applications. Whether the heterogeneous applications are older legacy programs or distributed applications on different networks, an agent can mediate between two systems. Researchers in software engineering have successfully implemented several agent systems that integrate heterogeneous applications (Genesereth & Ketchpel 1994; Petrie, 1996).

The use of agents as a user interface abstraction, can provide an alternative means of desktop manipulation. Limitations of the direct manipulation interface include scalability and level of expertise. As the volume of information at our fingertips increases, the hierarchy of files and links on our desktops becomes too deep to negotiate efficiently. As the direct manipulation interface is extended to accommodate the volume of information, frequently the complexity of using the interface is also raised (Maes, 1994). Less experienced computer users often have a
difficult time navigating through feature-rich interfaces. Agents are useful within the user interface environment because they can react to the actions of the user, providing assistance in response to various events. Intelligent agents can learn the preferences of the user, and thus can provide a personalized interface to each user.

Other benefits from using agents accrue as a result of designing software to be agent-like. A well-constructed agent that reacts and persists in its environment while pursuing multiple, top-level homeostatic goals is much more likely to be reused than a program with a single, achievable goal. The functional decomposition that occurs in delegating tasks to agents results in software modules that are responsible for specific tasks. This modularity makes it easier to locate logic errors and extend applications. While users appreciate the benefits of interface and interoperability abstraction, software developers appreciate the reusability and modularity inherent in agent-based software design.

DECISION SUPPORT SYSTEMS

The DSS research stream originated over twenty years ago and was provided with a solid foundation for on-going research and development by the works of Keen and Scott-Morton (1978), Bonczek, Holsapple and Whinston (1980), Sprague and Carlson (1982), Bennett (1983), and others. DSS provide support for semi-structured and unstructured decisions and represent a multi-disciplinary field comprised of researchers from Management Information Systems, Operations Research, Artificial Intelligence, Organizational Studies and others.

DSS are designed to support individuals, frequently managers, with little computing experience in a dynamic, decision making environment (Keen & Scott Morton, 1978). In providing process independent support for the decision maker, a DSS must be able to adapt to changes in business strategies, data and the preferences of the user. Thus, a significant feature of a DSS is that the architecture must be able to support substantial evolution (Keen & Gambino, 1983). DSS also require a speedy development process. A rapidly changing decision-making environment prohibits a lengthy development process as a system that takes too long to develop will be outdated before it is implemented.

The Components of DSS

Sprague and Carlson categorized the technical capabilities of a DSS into the three subsystems depicted in Figure 2.1 (1982). These subsystems are the Dialog Generation and Management System (DGMS), the Database Management System (DBMS), and the Model Base Management System (MBMS). The following subsections discuss the general purpose of these components. In addition, more detailed information is provided on the specific component functions that will be addressed within the two implementation chapters.

The Dialog Generation Management System Component. The DGMS subsystem provides the user interface and enables the user to interact with the DBMS and MBMS subsystems. Being the one component of the DSS with which the user directly interacts, the user views the DGMS subsystem as the entire DSS (Sprague & Carlson). Usability issues with the DGMS can determine the overall success or failure of the DSS. Sprague and Carlson define the ability to provide dialog styles that reflect the preferences of the users as one of four capabilities that the
Figure 2.1. The Dialog-Data-Models Paradigm (Sprague & Carlson, 1982).
DGMS subsystem should support (1982). This capability has proven difficult to successfully implement as most DSS do not take a proactive approach to ascertaining the user’s preferences. DSS that do provide this capability, frequently require the user to specify and store their dialog preferences.

**The Database Management System Component.** In the DBMS component of a DSS, the primary task is the capture and storage of internal and external data. Since the DBMS component within a DSS is often separate from transaction databases (Sprague & Carlson, 1982), the capture of internal data is a process that involves other applications within an organization and possibly remote sites within the organization. The capture of external data involves the extraction of data from either an internal collection source or directly from the external source. While the process of extraction should be automated so as to not disrupt the decision making process of the user, changing data needs, heterogeneous data sources and distributed data sources often prevent this automation.

**The Model Base Management System Component.** The primary function of the MBMS subsystem is the creation, storage, and update of models that enable the problem solving process within a DSS (Sprague & Carlson, 1982). The models utilize the data stored within the DBMS subsystem in creating alternate solutions for the user. The literature of the time suggests that, ultimately, a DSS should fit the style and *modus operandi* of each user, and that models should automatically be decided upon and invoked without burdening the decision-maker with the requirement of expertise in such matters. Two of the four functions of the MBMS identified by Sprague and Carlson deal with the way models are generated or restructured within a DSS (1982). If the models used to propose alternatives are outdated or specified incorrectly, the usefulness of the DSS declines. The complexity and time consuming nature of specifying models, whether the models are stored as subroutines, statements or data, contributes to the ease with which models become outdated.

**Current DSS Research**

Technological advancements and research advancements in other disciplines have been quickly adopted within the individual components or subsystems of DSS. In the last decade, within the DGMS, interfaces have improved substantially in appearance and usability through the use of visual programming development environments. Similarly, the interoperability and content of the DBMS component have been enhanced through Open Database Connectivity (ODBC), data warehousing, and web-based data access. The MBMS, however, is considered the least developed component and is the focus of much of the current research in DSS.

DSS researchers have noted the growing emphasis on model management issues in DSS research (Change, Holsapple, & Whinston, 1993). While some of this emphasis has been in response to the lack of standardization within the MBMS, the exploration of intelligent DSS (i.e., the application of artificial intelligence approaches within DSS) has also fueled the interest in MBMS research (Blanning, 1993). The combination of model management and artificial intelligence is considered essential in providing decision support and is viewed as the cornerstone of DSS development (Radermacher, 1994). This current emphasis on model
management and artificial intelligence provides an ideal environment for the integration of DSS and software agents.

**PLANNING**

Planning is a stream of artificial intelligence (AI) research concerned with designing a set of actions that will enable a system to solve a problem or reach a goal. Most of the research in this stream has focused on the design of domain-independent planning systems. These systems are able to produce plans that bring about a goal without using heuristics from the problem domain to guide the plan-generation process. Domain-independent planners have received a great deal of interest from the AI community over the past thirty years because such planners provide a means to increase the modularity and extensibility of intelligent systems. Plan generators were first designed back in the early 1970’s for controlling robotic actions. For example, a robot with a basic set of physical movements could successfully navigate through many different rooms when guided by a planning system. These planners have since been applied to other domains such as naval logistics, job shop scheduling, and optimization using simulated annealing (Hendler, Tate, & Drummond, 1990). In these other domains, an intelligent system with a basic set of procedures could similarly make appropriate responses or suggestions in many different environmental scenarios when controlled by a planner.

**A General Description of Planning**

The purpose of a planning system is to generate a plan that provides a solution to a problem. The application domain of a planning system is described by states and actions. This type of domain description is frequently referred to as a World Representation, or View, as the states and actions describe all possible scenarios for the application domain. When the problem is resolved the domain is said to be in the goal state. The domain is said to be in an initial state when conditions of the goal state are not satisfied. The blocks world shown in Figure 2.2 provides examples of these two states. The planner uses actions or operators to change the state of the system, eventually bringing it to the goal state. Operators are commonly described in the context of STRIPS, one of the first planning programs (Fikes, Hart & Nilsson, 1972). In STRIPS, operators were composed of three elements, preconditions, an add list and a delete list as shown in Figure 2.3. These three elements specify under what circumstances an operator can be employed and what changes occur to the state of the system as a result of employing the operator.

Any sequence of operators, or actions, is referred to as a plan, and a plan that brings about the goal state is referred to as a plan solution. In most application domains there is more than one solution plan as there is more than one way to achieve the goal state. The operators and states mentioned above serve as the input to a planning system. The output from the system is a plan solution. The process of searching through the possible combinations of operators to move from the initial state to the goal state is called plan generation.
Figure 2.2. Representation of Initial State and Goal State in Blocks World.
**Operator:** move(A, C, B)  
move A from C to B

**Preconditions:**
- on(A, C)  
- clear(B)  
- clear(A)

**Add List:**
- on(A, B)  
- clear(C)

**Delete List:**
- on(A, C)  
- clear(B)

Figure 2.3. Example Operator in a Blocks World.
State-Space Planners

While most planning systems utilize these same inputs and outputs, the defined search space and search process can differ greatly. Early planners, including STRIPS, used a state-space approach where the states represent nodes in a search space and the operators are used to move from state to state. A plan solution under this approach would appear as a path that traverses from the initial state to the goal state and would include the exact sequence of operators and states as shown in Figure 2.4. Two problem areas with this approach are 1) the specification of the problem domain in terms of every possible state that could arise, and 2) the computation required to traverse the search space. Enumerating each possible state is a lengthy process and not one the individual trying to solve the problem naturally performs. Typically, an individual with a problem is focused on the actions that can be taken to resolve the problem. A search through a state-space can be lengthy and often impractical to perform. Refining the search through domain specific heuristics, however, threatens the general applicability of the planner.

Partial Order Planners

A different approach to plan generation was first set forth in the planning programs NOAH (Tate, 1977) and Nonlin (Sacerdoti, 1977). Instead of exploring a search space of states, these planners explored a search space of partially ordered plans. These partial plans are composed of sets of operators that achieve some of the goals in the goal state; a strict operator order is not specified (Hendler, Tate, & Drummond, 1990). The focus of the search space on operators instead of states eliminates the requirement of specifying every possible state in the problem domain. The partial plan structure also allows a domain-independent representation of the interdependencies among the operators through the use of subgoals and causal links. When an operator is added to a partial plan, the preconditions of the operator become subgoals for a partial plan. Thus, the preconditions of each operator are the partial plans that the planner searches through to achieve the goal state. Any temporal relationships between subgoals, such as B must be clear before A can be placed on top of B, are represented as causal links. These subgoals and causal links can express relationships among operators without specifically coding domain heuristics into the planner.

The publication of the systematic nonlinear planning (SNLP) program (McAllester & Rosenblitt, 1991) provided theoretical rigor for the search space and search process of partial order planners by proving that these planners were complete and that systematic search patterns were produced (meaning no partial plans were used twice). In the SNLP program, a STRIPS representation of operators is used and the search proceeds through a space of partial plans. In SNLP, each partial plan is represented by 1) a set of operators, 2) an agenda, a set of subgoals that represent the preconditions of each operator, 3) a list of partial operator orderings and 4) a list of causal links.

The example of the blocks world state-space planner shown in Figure 2.4 is displayed as a partial order planner in Figure 2.5. With the partial order planner, the relations that comprise the goal state shown in Figure 2.4 would be the first subgoals added to the agenda. The planner would start at the initial state and work to achieve each subgoal by adding an operator to the partial plan. When an operator is added to the partial plan, any preconditions for the operator that are not true in the current state of the system are added to the agenda. As operators are added to the
The operator `move (A, C, Table)` implies move A from C to the Table

Figure 2.4. Example of a State-Space Search Path in a Blocks World.
Causal Links

causal_link(move(D, B, Table), clear(B), move(B, Table, C)). The operator move(D, B, Table) satisfies the precondition clear(B) for the operator move(B, Table, C).

causal_link(move(A, C, Table), clear(C), move(B, Table, C)). The operator move(A, C, Table) satisfies the precondition clear(C) for the operator move(B, Table, C).

Partial Orderings

move(A, Table, B) < move(D, B, Table) \lor move(B, Table, C) < move(A, Table, B)
move(A, Table, B) < move(A, C, Table) \lor move(B, Table, C) < move(A, Table, B)

The operator move(A, Table, B) threatens the two causal links above, so this operator must occur before move(D, B, Table) and move(A, C, Table) or after move(B, Table, C).

Figure 2.5. An Example of a Partial Plan Search Space in a Blocks World.
plan and preconditions added to the agenda, lists of causal links and partial orderings are also created. A causal link is a constraint of the form causal_link(O1, p, O2), where operator O1 satisfies precondition p for operator O2. In the first causal link shown in Figure 2.5, the operator move(D, B, Table) contains within its add list the relation clear(B). The operator move(B, Table, C) has the relation clear(B) as one of its preconditions. Thus, this causal link specifies that O1 supports O2. A partial ordering is a constraint that prevents an operator from threatening a causal link. In the example, the operator move(A, Table, B) threatens causal_link(move(D, B, Table, clear(B)), move(B, Table, C)) because if it occurs between the operators in the causal link, it deletes the precondition clear(B) and thus prevents the second operator from taking place. In a partial ordering, the threatening operator is constrained to take place either prior to both of the operators involved in the causal link or after these operators.

Operators are added to a partial plan until the goal state or some specified subset of the goal state is achieved. When a partial order planner generates a plan, it searches through the space of partial plans until the goal state is achieved. The solution plan generated is linear, meaning the exact order of the operators is specified. While this algorithm can still be computationally demanding, the operator-based structure of the search space allows the relationships among operators and among subgoals to refine the search space without threatening the domain independence of the planner.

Additional details on partial-order plans may be found in *Computational Intelligence: A Logical Approach* (Poole, Mackworth, & Goebel, 1998).
IBM has predicted that software agents will become the most important computing paradigm in the next ten years (Gilbert, 1997). The growing number of commercial and research agent implementations provides evidence that the computing industry recognizes the potential of this new paradigm (Huhns & Singh, 1998; General Magic, 1998; Gilbert, 1997, Mitsubishi Electric, 1997; ObjectSpace, 1997). The standard-setting bodies that have been formed to address the issues of agent communication and mobility furnish further confirmation (Chang & Lange, 1996; Neches, Fikes, Finin, Gruber, Patil, Senator, & Swartout, 1991).

It is the general goal of this paper to explore the promise of software agents in the realm of Decision Support Systems (DSS) and to provide guidance for DSS developers seeking to agent-enable their applications. To do this we first note the current state of DSS development in general and then that of software agent implementations within DSS.

An Overview of Decision Support Systems

The DSS research stream originated over twenty years ago and was provided with a solid foundation for on-going research and development by the works of Keen and Scott-Morton (1978), Bonczek, Holsapple and Whinston (1980), Sprague and Carlson (1982), Bennett (1983), and others. DSS provide support for semi-structured and unstructured decisions and represent a multi-disciplinary field comprised of researchers from Management Information Systems, Operations Research, Artificial Intelligence, Organizational Studies and others. Technological advancements and research advancements in other disciplines have been quickly adopted within the individual components or subsystems of DSS, namely, the Dialog Generation Management System (DGMS), the Database Management System (DBMS), and the Model Base Management System (MBMS) (Sprague and Carlson, 1982). In the last decade, within the DGMS, interfaces have improved substantially in appearance and usability through the use of visual programming development environments. Similarly, the interoperability and content of the DBMS component have been enhanced through Open Database Connectivity (ODBC), data warehousing, and web-based data access. The MBMS, however, is considered the least developed component and is the focus of much of the current research in DSS. The combination of model management and artificial intelligence is essential in providing decision support and is viewed as the cornerstone of more advanced DSS (Radermacher, 1994).

The Use of Software Agents in a DSS

Given the history of artificial intelligence in DSS research and the current interests in further integrating AI techniques in DSS, it is not surprising that DSS researchers have been quick to recognize the promise of employing software agents in DSS. The potential contributions of
software agents to DSS have been described as enormous (Whinston, 1997), and DSS implementations that highlight the use of agent-like programs for specific functions have started to appear in research journals (Elofson, Beranek, & Thomas, 1997; Maturana & Norrie, 1997; Oliver, 1996; Pinson, Louçã, & Moraitis, 1997). As is typical with an emerging technology, there has been much experimentation with the use of agents in DSS, but to date, there has been little discussion of a framework or a methodological approach for using agents in DSS. In addition, because of the subjective nature of agency and the wide-range of contexts and disciplines in which agent-like programs have been deployed, a general definition or description of agents has been lacking within the DSS/MIS literature. This difficulty in describing what an agent is has resulted in overuse of the term agent and poor guidance for DSS developers seeking to agent-enable their applications. And while DSS researchers are discussing agents as a means for integrating various capabilities in DSS and for coordinating the effective use of information (Whinston, 1997; Elofson et al., 1997), there has been little discussion about why these entities are fit for such tasks.

The purpose of this research is to show the promise of software agents in the realm of Decision Support Systems. Because definitions of software agents in the literature are so divergent, we must first expend considerable effort in generating a definition of agents apropos to DSS. The contributions of this paper are (1) to develop a useful definition of software agents; (2) to describe an agent-enabled DSS that we built -- the example we furnish is an extension of a DSS reported by Holsapple and Whinston (1996) and has been enhanced with several different types of agents; (3) to demonstrate the benefits gained from using agents in DSS; and (4) to use insight obtained from constructing the DSS to suggest a general framework for integrating software agents in DSS. We also comment on the additional effort necessary in adding agents to DSS. The paper is written so that it can be useful as a tutorial to those new to the concept of software agents.

The paper is organized as follows. The first section develops a definition of an autonomous software agent. In the second section, we describe the agent-enabled DSS we built, providing further insight into the essential and empowering agent features. The general benefits of using agents are described in the third section along with the specific benefits obtained in our agent-integrated DSS implementation. In the fourth section we develop a general framework for building agent-enabled DSS. Finally, the last section contains conclusions, limitations, and discussion of extensions, some of which are already underway.

SOFTWARE AGENT DEFINITION

Problems with Definitions in the Literature

When juxtaposed, literature definitions of software agents can be conflicting and jarring. Agent implementations based upon Minsky’s Society-of-Mind theory (Minsky, 1985) hold out simple processes as agents (Riecken, 1997; Boy, 1997); for example, a procedure that highlights an image when a mouse is moved over the related text is considered an agent. Conversely, agent-research at the other end of the spectrum discounts that viewpoint and suggests that agents are advanced computing assistants (Maes, 1994; Negroponte, 1997). But even within a particular viewpoint, there can still be confusion and imprecision in terminology. Imam and Kodratoff (1997, p.75) point out that "if a researcher or any curious person wanted to learn about intelligent
agents, he/she might get confused after reading even a few papers...." Various researchers claim abstract qualities such as autonomy, intelligence, or problem-solving ability as defining characteristics of an agent. But, as Covigaru and Lindsay (1991) note in their discussion of intelligent creatures and autonomous systems, there are relationships among these (and similar) terms used in the literature. Each cannot be the "key property."

We believe these definitional difficulties have arisen for two primary reasons: failure to explicitly provide a reference point in the agent definition, and the failure to differentiate essential (i.e., defining) agent characteristics from empowering ones. We will try to untangle these problems in general, but note that implicit in the approach we take is the tacit agreement with the literature (Bradshaw, 1997) that software agents must be described (e.g., via lists of attributes) if definitional progress is to be made. (It is interesting to note that this is the same avenue followed with DSS in their early days. See, e.g., Alter (1980, chapter 2) and his seven types of DSS.) A general list of those software-agent attributes frequently mentioned or implied in literature definitions is as follows: autonomy, goal-orientation, persistence, reactivity, mobility, intelligence, and interactivity.

Resolving Definitional Problems

The first step before examining the attribute list is to dismiss the Minsky Society-of-Mind viewpoint of agents in favor of the notion of agents as Maes (1994) and Negroponte (1997) think of them, namely as "advanced computing assistants." We do this because we believe the latter notion will be more useful in the DSS arena. We also believe that most professionals, whether in the DSS area or not, do not think of mouse movements for the purpose of highlighting images as defining agent behavior.

Reference point. With the above distinction in mind, an agent is a representative or a substitute or a stand-in for another. To define an agent meaningfully, one must specify (1) who the agent is representing (the agent employer), (2) the task to be done, and (3) the domain of the task. (See Figure 3.1.) Note that failure to stipulate all three items as a reference point can lead to confusion. For example, merely stating that basketball player Michael Jordan had an agent does not define that agent usefully. If we state that the agent domain was salary negotiations with the National Basketball Association; and that the task was to procure and maintain most favorable economic terms for the employer of the agent (namely, Michael Jordan); then we recognize that the agent represented him in salary negotiations, but was not expected to stand in for him during basketball games. Moreover, we note that different agents will perform widely differing tasks. For example, to be successful, your automobile insurance agent will need to do different jobs than Michael Jordan's salary agent and those of a software agent. As a consequence, different agents will possess differing values of the attributes listed above. Baseball player and longevity expert Cal Ripken, Jr.'s, agent will have needed to persist longer in agent tasks than a wife legally responsible for making a medical decision for her suddenly dying and incapacitated husband.

Differentiation. As suggested, the definitional approach used here is to generate a subset of attributes from the seven found in the literature given above; the hope is to find two or three essential terms that, taken together, will usefully and distinctively describe a software agent. The
Figure 3.1. The Empowerment of a Software Agent.
next step in proceeding through the attribute list is to redirect our focus to a goal of defining an autonomous software agent. We do this because the literature indicates, as will be seen, that autonomy is at the very essence of human and hence agent behavior, and thus is a more all-encompassing term than the other six characteristics.

Wooldridge and Jennings state that an essential characteristic for programs designed to act in an agent-like manner is that they be able "to operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state" (p. 4, 1995). They define this characteristic as autonomy (Wooldridge & Jennings, 1995). This fundamental agent feature is similarly defined by numerous other agent researchers (Nwana, 1996; Franklin & Graesser, 1996; Gilbert, 1997) and is in keeping with Webster's dictionary definition of the term autonomous from a human perspective, as “independent; and having self-government” (p. 93, 1997). If the essence of an agent is to be a representative and/or a substitute, then certainly an agent should do so independently, i.e., without having to query the agent employer repeatedly for help or instructions. This would certainly be true in the DSS arena: users should not be expected repeatedly to supply help and instructions to DSS agents that are supposed to be helping them.

Part of the search in finding a definition of agents is to determine the human-like attribute(s) that agents must possess when they become stand-ins or representatives of humans. This is due to the fact that, although agents can be representatives of other agents, ultimately, some higher-level agents will be agents for humans. (In software, agents will often serve users, who are human.) In a very interesting paper on human-like systems, Covrigaru and Lindsay (1991) conclude that the essence of human-like systems is autonomy. They decide that the essence of human-like behavior is not problem solving per se, as was assumed in the early days of artificial intelligence. Rather, they stipulate, an entity must be autonomous to be truly intelligent, truly living, and truly humanoid. Autonomy is the characteristic that enables humans and human-like systems to act as assistants. They further develop the idea that several other attributes from the literature we have listed above are components of a definition of autonomy. Therefore, we conclude along with Covrigaru and Lindsay (1991), that the term autonomy is overarching -- it includes concepts such as goal-orientation.

We turn our attention back to the search for other attributes to list as essential components for our definition, but now our task is to define an autonomous agent. The literature, unfortunately, gives only partial support, and we will have to resort ultimately to common sense.

Covrigaru and Lindsay (1991) argue that the essence of autonomy is that the entity or system must be trying to accomplish something, i.e., it must be goal-directed. Rocks, for example, do not pursue goals and thus are unlikely candidates to become agents. As the literature does not give further substantive guidance as to which features are essential and which are not, we now take a "common-sense" look at defining an autonomous agent, beginning with the reference point given in Figure 3.1.

As the left part of Figure 3.1 suggests, someone or something "hires" or employs an agent to do a task in a particular domain. One possible means of specifying the task (taking our lead from Covrigaru and Lindsay) is to state the goals of the task. Looking at the other terms on our list of agent attributes, it is not always essential that the agent be mobile because many agent tasks can
be performed in one location. Moreover, it is not essential that the agent possess intelligence, at least in the AI sense of the term, because many tasks just require action and not a lot of reasoning or inferential capability. Similarly, interactivity (i.e., the ability to interact and communicate with others) is not an essential task. For example, the agent I employ to cut my grass need not be either particularly intelligent or a great communicator. However, an agent must possess an ability to react in the domain at some fundamental level, and the agent must persist long enough to achieve the goals of the employer. Thus we conclude that an agent must possess a goal-orientation, persistence, and reactivity. Although intelligence, mobility, and interactivity may enhance the capabilities of an agent, they are not essential features.

Thus, the list of essential features is goal-orientation, persistence, and reactivity. Again, note that for a given agent, each characteristic is defined in terms of a reference point. Just as human agents differ in persistence, we will not be surprised to see some software agents persisting for milliseconds, whereas others will persist for weeks. The requirement is that the agent endure long enough to complete the specified task in the specified domain.

The remaining features in our initial list of seven, intelligence, mobility, and interactivity, comprise the list of empowering agent features. The use of one or more of these empowering features may significantly enhance the usefulness of an autonomous agent, but as noted previously, such features are not considered essential.

Both the essential and the empowering agent attributes have been discussed somewhat loosely to this point. We now provide our definition/description of an autonomous software agent, which is followed by elaboration on attribute terminology.

Our Definition

An autonomous software agent is a software implementation of a task in a specified domain on behalf or in lieu of an individual or other agent. The implementation will contain homeostatic goal(s), persistence, and reactivity to the degree that the implementation (1) will persist long enough to carry out the goal(s), and (2) will react sufficiently within its domain to allow goal(s) to be met and to know that fact.

Note the following issues with respect to this definition:

Homeostatic goal(s). It is difficult to imagine a personal assistant or agent that works independently, having some kind of control over its actions and internal state, that does not have a goal. An agent without a goal, some assigned responsibility or task, provides no assistance to the user and has no means to act autonomously because it has no act to carry out.

A stronger view of the goal-orientedness feature holds that an autonomous entity do more than just attain a goal and then cease to function. Instead, an autonomous entity should seek to attain the goal and then maintain that goal state for as long as the user desires. Covrigaru and Lindsay (1991) refer to such goals as homeostatic goals and state that autonomous systems tend to pursue these homeostatic goals rather than what is described as achievable goals. Homeostatic goals do
not terminate when the system is in a final state; rather a monitoring process is initiated with the objective of re-achieving a final state if a change from that state occurs. An achievable goal is one that terminates when the final goal state is reached. Stated differently, homeostatic goals operate as an administrative mechanism so that an agent can reach and maintain its own achievable goals. For example, an agent designed to monitor the competition’s prices would be assigned the achievable goal of watching for a price change and reporting the change when it occurs. The homeostatic version of this goal requires the agent to monitor the competition’s prices indefinitely and, after reporting a price change, continue to monitor for future changes.

In effect, the homeostatic goal acts as an administrative goal at a higher level in that it is ensuring that the achievable goal is properly pursued. An agent with a homeostatic goal more accurately represents the metaphor of a personal assistant.

**Persistence.** In the software agent literature, the feature of persistence is interpreted as a program that is continuously running (Chang & Lange, 1996), even if that “running” means that the program is temporarily sleeping or in a “cryogenic state” (Bradshaw, Dutfield, Benoit & Wooley, 1997, p.385). Webster defines persistence as “the continuance of an effect after the removal of its cause” or as “enduring continuance” (p. 1007). The notion of persistence from a human perspective is that the entity or effect will exist for a long time relative to the time required to achieve a goal. Persistence is frequently implemented in an agent by giving it at least one thread of execution and by implementing a control structure that requires the agent to continuously monitor its state, including the status of its goal(s). The thread of execution ensures that the agent receives the necessary processing time and prevents the agent from being disrupted or slowed down by other processes and threads executing on the same computer. The control structure ensures that the agent can pursue homeostatic goals, continuously working toward the achievement of the goal and the maintenance of the goal state once it is achieved. Enabling the agent to save its state in some manner, say to a text file or database, can provide a stronger level of persistence to provide for the case of an emergency shutdown.

**Reactivity.** In the software agent literature, a reactive program is defined as one that can recognize changes in its environment and respond to those changes in a timely manner (Franklin & Graesser, 1996; Wooldridge & Jennings, 1995). This definition is similar to Webster's definition of reactivity as “…responsiveness, as to a stimulus or influence” (p. 1117) with one important difference. As with autonomy, an agent is reactive only within a specific environment, and as Franklin and Graesser (1996) stress, an agent may cease to be an agent when it is outside of its environment. For example, a software agent that learns the musical preferences of its user would react appropriately to the user’s selection of a classical recording but would not react appropriately, or at all, to an agent from a manufacturing domain that is attempting to negotiate the purchase of resources. Similarly, a chess-playing program would react appropriately to a move by an opponent, but would not react appropriately to someone’s movement to unplug it.

Reactivity as we are applying it to software programs does not require intelligence and is comparable to a stimulus-response scenario. Being reactive does not necessarily require intelligence, as a doctor testing a human knee for reflexes would indicate. That is not to say that a software program that is either intelligent or able even to (say) interact with its environment is
not desirable. It is just that these extra features are not deemed essential to developing an autonomous agent.

In summary, we believe that this definition and the descriptions of the three essential features provide utility to the DSS builder. To develop an agent, a builder must include three basic constructs. If the implementation were done in (say) Java, persistence could be obtained by running the agent in a separate thread, which could imply using the "… extends Thread" class extension. The homeostatic goal could be accomplished through control structures such as a "while" loop and nested "if" statements. Reactivity could be achieved with a "Listener" that waits for particular events and then runs event handlers when those events occur.

Empowerment

Having provided a description of an autonomous agent and defined its essential features, we now describe the empowering agent attributes. The empowering characteristics of agents (refer to Figure 3.1) are mobility, intelligence, and interactivity. Recall that these features are not essential in determining agency, but they may be important in making an agent useful or impressive. Since these three terms generally have special meaning when used in a software context, we now point out the following issues from the literature:

Mobility. In a networked environment, applications typically communicate with one another through remote procedure calls (RPC) or by exchanging scripts. Mobile code provides an alternate means of communication and is frequently associated with software agents. In some agent implementations, software agents are actually defined as mobile code (White, 1997). Mobile code differs from an RPC or the exchange of scripts in that with mobile code, the entire procedure “is passed to the remote server, including its code, data, execution state and travel itinerary.” (Chang & Lange, 1996, p.1)

A mobile agent is an agent that can be transported like mobile code. An agent that passes messages to a remote location is merely communicating and not exhibiting the feature of mobility. Mobile agents can move to multiple remote sites by either carrying an itinerary with them or by being dispatched to another site by the user, an agent server or another agent. Persistence and reactivity enable them to complete their tasks when remote sites are unavailable as they can wait at their current location until the site is accessible.

Mobility is an empowering agent characteristic because it enables agents to use distributed resources and to more efficiently utilize networking resources. Distributed processing is facilitated because mobile agents utilize the computing resources of the current host. Networking resources are more efficiently utilized because mobile agents can move to various sites in pursuit of their goals instead of sending numerous messages and RPCs to each site of interest.

Currently most agent implementations are proprietary in nature and do not support other (i.e., “foreign”) mobile agent systems. Standard-setting bodies such as the Object Management Group (OMG) had previously provided specifications only for stationary, distributed objects (Chang & Lange, 1996), but are now in the process of establishing guidelines for mobile objects.
Standardization for mobile, distributed objects could provide a common interface for mobile-agent systems and make it easier to reap the benefits of this technology.

**Intelligence.** Intelligence is an enabling feature that allows an agent to pursue its goals more efficiently and expertly with less assistance from the user or designer. IBM generally describes intelligence with regard to its own agents as the degree of reasoning and learned behavior (Gilbert, 1997). Imam and Kodratoff (1997) go a little further when they summarize an American Association for Artificial Intelligence workshop effort to define the term. They describe an intelligent agent as a “system or machine that utilizes inferential or complex computational methodologies to perform the set of tasks of interest to the user.” (1997, p. 76). The notion of inference or (machine) learning or reasoning is implicit or explicit in both definitions.

Early attempts in the artificial intelligence community at developing a program that functions as an intelligent, human-like creature focused on problem solving (Harmon & King, 1985) as a key feature for such a program. This ability to solve problems was generally referred to as intelligence. What appears to be intelligence in humans, however, is simply some implemented mechanism in a program (Covrigaru & Lindsay, 1991). Intelligence, as defined, is a computational tool, and is not required for a software program to behave in an agent-like manner. This feature can greatly enhance the usefulness of an agent, but the lack thereof does not imply a useless agent.

Agents with varying types of intelligence have been implemented, including those that utilize genetic algorithms (Oliver, 1996), ones that combine a knowledge base and learning-by-example (Maes, 1994), and systems with memory-based reasoning (Lashkari, Metral & Maes, 1994). Designers can empower their agents with these tools of computational intelligence, improving the agent’s problem-solving ability and requiring less user interaction with the agent.

**Interactivity (Communicative Ability).** The ability to communicate with users and other agents is another important enabling feature for software agents (Franklin & Graesser, 1996; Wooldridge & Jennings, 1995). Agents that can carry on a dialog with users and other agents, and not just report the results of their actions, are considered interactive or communicative. This type of dialog is generally supported through the use of an agent communication language. Interactivity is not considered a fundamental agent feature because an agent may be designed to carry out a task that does not require it to carry on a dialog with others. For example, an agent that has been designed to monitor a web site and update a database when changes occur may only need to report its monitoring results.

While communication is not a required agent feature, an agent that can communicate with others can significantly enhance its abilities. For example, an agent that can exchange information with other agents can be more efficient through cooperation and delegation. A communicative or social agent could save itself a journey to several remote sites by communicating with an agent that already knows the information it is seeking. An agent could additionally gain a great deal of efficiency by spawning several new agents and instructing them to accomplish tasks in parallel.
Agent communication is implemented through message passing, but as with mobility, communication specifications in different agents systems are often not compatible. An agent communication language (ACL) has been developed by the ARPA (Advanced Research Projects Agency) Knowledge Sharing Effort to provide a means for communication among agents developed in different programming environments for different purposes or domains (Neches et al., 1991). ACL provides a common set of communication performatives or actions to be used among agents and is establishing standard ontologies for various domains. The language has been adopted by numerous agent implementations and is especially well received in agent-based software engineering (Genesereth & Ketchpel, 1994; Petrie, 1996).

In summarizing our discussion of empowering features, it is worth noting that while mobility, intelligence, and interactivity can certainly enhance the capabilities of an agent, taken individually, these features do not create a personal computing assistant. There is no autonomy without the three essential features, and thus the user of such code will be unable to delegate tasks to it. Instead the user would be forced to initially instigate actions he or she wishes the code to take and perpetually re-instigate these actions until no longer needed.

To illustrate our definition of autonomous agent and some of its nuances, several agents will be described. We have implemented agents within a real DSS, thereby establishing context. The descriptions of the individual agents highlight the essential and empowering features that these agents exhibit, providing an implementation-level perspective of the features described above.

AN IMPLEMENTATION OF AN AGENT-INTEGRATED DSS

The DSS implementation described is based upon a case study used in a DSS textbook (Holsapple & Whinston, 1996) and supports variance-analysis investigation for a manufacturing firm. When a variance from the budget occurs in the rate (i.e., hourly cost) or usage of raw materials, labor or overhead, the DSS provides support for whether the variance should be investigated based upon criteria specified by the user and prior investigations of variances.

Holsapple and Whinston's Variance-Analysis DSS

Upon accessing the DSS, an authorized user would select a product to analyze and would then view the current variances for that product. The standard and actual parameters are loaded from the database, with the DSS allowing the user to perform what-if analysis with these parameters, uploading any desired changes in budgeted parameters to the database. Variances that exceed a pre-determined cutoff value are flagged. Users can review and edit the cutoffs set for each parameter.

Additional criteria for investigating variances can be reviewed and edited. These criteria are used to assign a relative weight to the variances previously calculated and provide additional support for the user in deciding whether a variance should be investigated. Should the user so decide, the cost of the investigation and the sources of the variance revealed by the investigation can be recorded in the database and reviewed when future variances occur.
In summary, the DSS supports the user in deciding whether to investigate a variance that has occurred between budgeted and actual costs. This support enables the user to avoid costly investigations into variances that have little impact on the economic health of the organization. Similarly, the DSS highlights variances that should be investigated to avoid further erosion to the organization’s health.

A limitation of the DSS, however, is the historical nature of the support provided. Variances are only calculated, analyzed, and examined by the user after the resource consumption or rate has strayed from the budget and affected earnings. Users would have to directly manipulate the information used by the DSS in order to project future variances, effectively running what-if analyses using estimates of the needed inputs. This type of limitation is typically accepted by DSS users as the *status quo* due to the costs of automating the projection of future variances using actual data. Such costs would include monitoring the current *rates* of resources. For example, in order for the system to project variances in resource rates, the rates of labor, materials and overhead would need to be continuously monitored. This type of monitoring would allow the user to observe the effects of an increase in materials cost prior to actually purchasing any materials at this new cost. The costs of having an individual monitor all material, labor and overhead rates and input these changes into the DSS, however, would typically be prohibitively high. It would similarly be very costly to track resource *usage* at such a detailed level that the DSS could project excessive usage variances before they occurred. However, if one could effectively transfer the burden of projecting variances from the user and his or her staff to the DSS and its agents, then the DSS would provide more support by generating decision-making alternatives that could prevent significant variances from ever occurring.

**Agent Integration**

In order to explore the benefits of agents, Holsapple and Whinston's Variance-Analysis DSS described above was built and then enhanced with agents. Because we wished to use Java-based agents, we wrote the Variance-Analysis DSS in Microsoft's J++™ (1997); supporting data for the application were stored in a Microsoft Access™ (1997) database. Software agents were then integrated into the DSS using Java™ (Sun, 1998) and the Voyager™ class library from ObjectSpace (1998). Part of the enhancements we included involved solving a mathematical programming problem. For this we used Microsoft Excel's™ Solver (1997) and wrote code in Visual Basic for Applications.

Agents were integrated into this existing DSS for the purpose of automating more tasks for the user, enabling more indirect management, and requiring less direct manipulation of the DSS. Specifically, agents were used to collect information outside of the organization; to project future variances in materials, labor, and overhead; and to generate decision-making alternatives that would allow the user to focus on variances that were found to be significant. The use of agents provides an automated, cost-effective means for making these projections and generating alternative courses of action. Assumptions of this implementation are (1) the general acceptance of agents in electronic commerce, including the existence of business partners that are willing to host other agents, and (2) the availability of directory and matchmaking services.
The five kinds of agents we added to the system are listed in Table 3.1. Also shown in that table is their placement in the three DSS components, according to a builder's view of a DSS (using Sprague and Carlson's (1982) terminology). Note that agents have been placed in each of the components and that a separate domain manager agent (DMA) has been placed in the DBMS, the MBMS, and the DGMS. The function of the DMAs is to maintain control of all agents in their domains.

Each of five agents exhibits the essential features of persistence, reactivity and goal-orientedness as shown in Table 3.2. The reference point for each agent, specifically the employer/client, the task, and the domain, are also shown in this table. Table 3.3 describes the empowering features exhibited by these agents.

A more functional view of the agent-enhanced DSS is provided in Figure 3.2 and tells more of the story behind this agent implementation. For purposes of describing this functionality, assume the user decides to deploy a data-monitoring agent from the DBMS to a supplier's site, where it is to monitor the price of Part 0001. To do this, the user need only specify the supplier URL, part number, and current price for the part, as seen in the interface for the agent shown in Figure 3.2a. The agent remains at the supplier's site, periodically checking the price of the part, reporting any significant changes that occurred. The homeostatic goal of this agent is to monitor the assigned part number and report changes in price that cross threshold values. The agent is persistent in that it continually runs in its own thread of execution and has a control structure that enables it to perform its homeostatic goal. In addition, the agent periodically communicates with the originating site so that in the event that the supplier's site shuts down or the agent is shut-down by the supplier, the originating site is alerted to the absence of the agent. The monitoring mechanism of the agent enables it to react appropriately to significant changes in its environment. For this simple agent, a significant change occurs if the part price crosses threshold values. Because of its simplicity, the data-monitoring agent offers a great deal of reusability. Numerous agents can be dispatched to many different sites to monitor the rates of materials, labor or overhead using this one agent class. This implementation assumes that the part suppliers would be willing hosts to such foreign, automated programs. In the event that the suppliers did not wish to serve as an agent host, an alternative version of the data-monitoring agent could filter the text of the supplier's web site for price changes.

Data-gathering agents have been integrated into the DSS to locate new sources of data. Specifically, the data-gathering agents utilize a directory service to locate potential suppliers of manufacturing parts. (Several directory and/or matchmaking services have been developed on the Internet or for networked agent communities. See Bradshaw et al., 1997; Genesereth, 1997.) A data-gathering agent can be dispatched to a directory site(s) to look for alternate sources of a specific part. When a new source is found, the data-gathering agent collects information about the source, including the name, location, and URL of the new source. The data-gathering agent can be sent off with a lengthy itinerary of directory sites and will report back new sources as they are found.

When the data-monitoring agent detects a price change or the data-gathering agent finds a new supplier, these agents report these changes to their domain-manager agent. The domain manager agent monitors the location of all other agents functioning on behalf of the DSS and takes or
Table 3.1. Agent Types by Placement in DSS Domain.

<table>
<thead>
<tr>
<th>Agent Types</th>
<th>MBMS</th>
<th>DBMS</th>
<th>DGMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-Monitoring Agents</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-Gathering Agents</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Modeling Agents</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain Manager Agents</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Preference-Learning Agents</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Table 3-2. Example agents utilized in the extension to the Holsapple & Whinston (1996) manufacturing-firm DSS.

<table>
<thead>
<tr>
<th>AUTONOMOUS AGENT</th>
<th>HOMEOSTATIC GOAL</th>
<th>PERSISTENCE</th>
<th>REACTIVITY</th>
<th>EMPLOYER/CLIENT</th>
<th>TASK</th>
<th>DOMAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-Monitoring</td>
<td>Report when any price change crosses given threshold values</td>
<td>Stay at supplier's site &quot;forever&quot; or as long as the vendor supplies parts.</td>
<td>Capable of detecting vendor price changes</td>
<td>User</td>
<td>Monitor the current rates of the 3 types of resources and report on them</td>
<td>Vendor site on an Extranet</td>
</tr>
<tr>
<td>Data-Gathering</td>
<td>Report discovery of potential suppliers of manufactured parts at reasonable prices</td>
<td>Lifetime of the DSS</td>
<td>Capable of examining directory sites and understanding language used there</td>
<td>User</td>
<td>Look for alternate vendors of specific part; if found, send message back with name and location of source</td>
<td>Travel to directory sites</td>
</tr>
<tr>
<td>Modeling</td>
<td>Maintain &quot;optimal&quot; price and resource policies; report significant dollar consequences</td>
<td>Lifetime of the DSS</td>
<td>Capable of receiving inputs from the Domain Manager Agent (DMA) and passing results back to the DMA</td>
<td>Domain Manager Agent (DMA)</td>
<td>When notified by DMA, formulate an LP model, solve it using Excel's Solver, and report solution to DMA</td>
<td>Model Base Management System (MBMS) of DSS</td>
</tr>
<tr>
<td>Domain Manager (say in the DBMS)</td>
<td>Monitor location and tasks of both local and remote agents functioning on behalf of domain activities. Respond to all messages.</td>
<td>Lifetime of the DSS</td>
<td>Capable of communicating with agents (even at a distance) and keeping track of their whereabouts</td>
<td>User</td>
<td>Monitor all other agents (both local and remote) acting on behalf of the domain; trigger appropriate actions upon hearing from them</td>
<td>Data Base Management System (DBMS) of DSS (Similar agents exist in the MBMS and the DGMS.)</td>
</tr>
<tr>
<td>Preference-Learning</td>
<td>Learn a specific user's preferences based on the actual history of user/DSS interactions</td>
<td>&quot;Lifetime&quot; of a user of the DSS, even across different sessions</td>
<td>Capable of observing user actions and storing them</td>
<td>User</td>
<td>Record whether specific user takes modeling agent's advice or proceeds on own</td>
<td>Dialog Generation and Management System (DGMS) of DSS</td>
</tr>
</tbody>
</table>
Table 3.3. Benefits from the empowering characteristics of agents in the extension to the Holsapple & Whinston manufacturing-firm DSS.

<table>
<thead>
<tr>
<th>AUTONOMOUS AGENT</th>
<th>MOBILITY</th>
<th>INTELLIGENCE</th>
<th>INTERACTIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-Monitoring</td>
<td>Goes to (and stays at) supplier's site. Saves user from having to monitor supplier's prices. Only reports promising prices.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-Gathering</td>
<td>Goes to directory sites. Locates potential suppliers of parts, relieving user from task. Only reports promising suppliers.</td>
<td></td>
<td>In an enhanced system, the agent could talk (using ACL) to other agents it meets and get additional leads on promising directory sites or suppliers. ACL</td>
</tr>
<tr>
<td>Modeling</td>
<td>In this system, the agent provides a mathematical model to which it can accept inputs, run, and interpret and furnish results to the appropriate agent. In an enhanced system, the agent could more generally use AI planning to generate alternatives, plans, and possible actions for the user.</td>
<td>Agent helps to integrate heterogeneous communication styles. In particular, the agent communicates with the Domain Manager Agent via Java, and formulates the LP model and Excel's Solver using Visual Basic for Applications, translating results back into Java for the DMA.</td>
<td></td>
</tr>
<tr>
<td>DMA</td>
<td></td>
<td></td>
<td>Agent provides interoperability by integrating heterogeneous, distributed agents. The agent communicates with all other agents operating on behalf of, or in, the domain. The user need not keep track of agents' creation, demise, purposes, etc.</td>
</tr>
<tr>
<td>Preference-Learning</td>
<td>The agent observes and records the user's disposition to follow the modeling agent's recommendations. In an enhanced system, the agent could invoke machine learning to determine many of the user's preferences.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
a. A data-monitoring agent (DM0001) is deployed from the DBMS to a supplier's site, where it is to monitor the price of Part 0001.

b. A data-monitoring agent reports a price change to the DBMS domain manager agent. The information is then (indirectly) passed to the modeling agent.

c. The modeling agent feeds the price change into Excel's Solver, which ultimately produces the output indicated.

Figure 3-2. Some of the Agent Activity in the Agent-Enhanced Holsapple and Whinston Variance-Analysis DSS.
triggers appropriate actions upon receiving communications from both local and remote agents that originated within its domain. When a new agent is created, it is immediately registered with the domain manager, which maintains its location whether it is on site or located remotely, regardless of the number of hops in its itinerary. In the event that an agent is shut down and is no longer active, the domain manager agent provides notification of the agent's demise. The domain manager is constructed similarly to the data-monitoring agent in that it is given its own thread of execution and a control structure that allows it to continuously pursue its goals. A view of the domain manager agent interface displaying communications from two agents, the data-monitoring agent above and a data-gathering agent, is shown in Figure 3.2b. Note that the data-gathering agent has been busy at a directory site on the Internet and is reporting potential new vendors, and that the data-monitoring agent has reported a price increase to $2.50.

The DBMS domain manager agent then contacts (indirectly) the appropriate modeling agent. The modeling agent’s function is to generate decision-making alternatives that will enable the user to minimize a detrimental variance from budget. This agent is responsible for generating a maximum profit product-mix strategy, given the increased vendor price, and reporting (indirectly) to the user any significant dollar consequences. In actuality, the modeling agent does this by integrating a linear programming model developed and supported in the Solver add-in from Microsoft’s Excel™ together with a Visual Basic for Applications (VBA) module. The mathematical model, in its spreadsheet form, is shown in Figure 3.3. However, the user never sees this model since the modeling agent provides a level of abstraction between the application programming interface (API) of Solver, Excel’s VBA, and the DSS, hiding its complexity. The output from the model, if deemed consequential dollar-wise, is then passed back to the domain-manager and displayed in the DSS as shown in Figure 3.2c. Model output value significance is determined via thresholds; naturally, thresholds could be set so every product-mix change is reported, if desired.

At this point the user may choose to adopt the DSS-recommended product mix or may specify a strategy of his or her own choosing. To monitor the user’s actual responses to variance occurrences, a preference-learning agent was integrated into the agent-enhanced DSS. The agent also tracks whether the action corresponds to the actual variance level and variance cut-off parameter in effect at the time of the user’s action. In addition, as the learning agent tracks the user’s actions, it calculates the average variance level at which the user is effectively making changes in product mix. The user can review the average variance level and update the variance cut-off parameter at anytime.

Each of the agents we have built exhibits the essential agent features with respect to its specified reference point in the DSS. As noted in our definition of autonomous agents, however, if any of these agents ceased to exhibit any of these essential features within its specified domain in the DSS, it would then cease to be an agent in the DSS. This transformation (into "nothingness") may appear troubling initially, but we believe it is the desirable and useful way to define an agent. The case described is similar to a situation where we hire a tax agent who does our taxes for years, but when new tax law is passed, the individual fails to adopt the new code. In such a case we would get rid of the tax agent and either obtain a new one or proceed on our own. That individual would cease to be our agent.
### Production Mix

<table>
<thead>
<tr>
<th></th>
<th>Labor Units</th>
<th>Labor Rate Per Hour</th>
<th>Material Units</th>
<th>Material Cost Per Unit</th>
<th>Overhead Units</th>
<th>Overhead Rate Per Unit</th>
<th>Sales Price Per Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product1</td>
<td>0.21</td>
<td>8.00</td>
<td>2.00</td>
<td>2.50</td>
<td>0.08</td>
<td>0.25</td>
<td>7.00</td>
</tr>
<tr>
<td>Product2</td>
<td>0.27</td>
<td>7.50</td>
<td>2.00</td>
<td>4.50</td>
<td>0.06</td>
<td>0.40</td>
<td>12.00</td>
</tr>
<tr>
<td>Product3</td>
<td>0.18</td>
<td>8.00</td>
<td>3.00</td>
<td>6.60</td>
<td>0.09</td>
<td>0.60</td>
<td>22.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Total Labor Units</th>
<th>Total Labor Cost</th>
<th>Total Material Units</th>
<th>Total Material Cost</th>
<th>Total Overhead Units</th>
<th>Total Overhead Cost</th>
<th>Total Sales</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product1</td>
<td>50</td>
<td>10.5</td>
<td>84.00</td>
<td>100.0</td>
<td>4.0</td>
<td>1.00</td>
<td>350.00</td>
<td>15.00</td>
</tr>
<tr>
<td>Product2</td>
<td>50</td>
<td>13.5</td>
<td>101.25</td>
<td>100.0</td>
<td>3.0</td>
<td>1.20</td>
<td>600.00</td>
<td>47.55</td>
</tr>
<tr>
<td>Product3</td>
<td>533</td>
<td>95.9</td>
<td>767.52</td>
<td>1,599.0</td>
<td>48.0</td>
<td>28.78</td>
<td>11,726.00</td>
<td>376.30</td>
</tr>
<tr>
<td></td>
<td>119.9</td>
<td>952.77</td>
<td>1,799.0</td>
<td>11,253.40</td>
<td>55.0</td>
<td>30.98</td>
<td>12,676.00</td>
<td>438.85</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Unit Quantity</th>
<th>Total Sales</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old Mix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product1</td>
<td>464</td>
<td>603.20</td>
<td></td>
</tr>
<tr>
<td>Product2</td>
<td>50</td>
<td>47.55</td>
<td></td>
</tr>
<tr>
<td>Product3</td>
<td>50</td>
<td>35.30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>686.05</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3. The MBMS Excel (Solver) Solution to the Linear Programming Problem.
AGENT BENEFITS IN A DSS

Having described the scenario of an agent-integrated DSS, we now discuss the benefits of using autonomous agents in DSS. The general benefits of using agents are first enumerated and then discussed with respect to DSS. Following that, we show how the essential and empowering features exhibited by five individual agents deliver benefits within the context of our implementation.

General Benefits

Software agents in general have emerged as a means to indirectly manage computers and computer-related tasks instead of directly manipulating them (Maes, 1994). Agents provide this indirect management by introducing a level of abstraction between the user and the computer. For example, if a supervisor asks a (human) personal assistant to schedule appointments with several employees, then that manager has created a level of abstraction between him or herself and the employees with regard to appointment details. The supervisor is spared from having to deal with the complexities of coordinating schedules. Similarly, with a computer-related task, a user can be spared from some computing complexities by using a software agent, i.e., a personal computing assistant.

Bradshaw (1997) notes that this abstraction, or complexity reduction, can help the user in two important areas, interoperability and user interfaces. By interoperability, Bradshaw means that agents can integrate heterogeneous applications and networks. With user-interface abstraction, users can be freed from the details of the ever-increasing volume of information to be processed, and can have information personalized as they prefer to see it. Both of these interface benefits are important because they can free the user from distractions, enabling concentration on the managerial aspects of decisions, for example, rather than computing minutiae.

The general benefits of agents discussed above are directly applicable to DSS users. The primary benefit of an agent-enriched DSS is abstraction and the resulting automation and reduction in complexity provided. The agent provides an additional layer of support between the user and the actual DSS. Through this abstraction, the related task becomes more automated, requiring less action on the part of the user.

Specific Benefits of an Agent-Integrated DSS

With these general, literature-based benefits in mind, the particular benefits of each agent may now be observed by examining their essential and empowering characteristics (these features were listed in Tables 3.2 and 3.3).

1. Data-Monitoring Agent. Recall that this software agent’s purpose is to monitor price-changes of given items at a (friendly) supplier’s site. When a “significant” price change occurs, the agent is to send word back to the DSS at its home site that the change occurred. This agent enhances the DSS in two fundamental ways: by automating the retrieval of information, and by improving the quality of that information (in the sense that the database is updated immediately for changes in vendor prices). This latter benefit implies that additional DSS
can now be built possessing real-time, on-line data capabilities. While not all DSS require this enhancement, it will be a significant benefit for many.

The benefits listed above are based on a comparison of an agent-enabled DSS with a traditional (no agent) DSS. An important question is why must agents be used to provide these benefits. Aren't there non-agent approaches giving the same benefits? For example, in the case of the data-monitoring agent, why not just use mobile code? The answer lies in the definition of an agent -- an agent provides autonomy. Autonomy means the user passes off the task and is freed from worrying about it. It is possible to construct non-agent implementations of tasks, but then, by definition, either persistence or reactivity or goals will be lacking, and the user will have to provide what is missing. In the case of mobile code, the user would have to periodically instruct the mobile code (through remote procedure calls) to check the vendor’s price, and then the user would have to review the change to see if it was significant. Autonomous agents have been used to enhance the DSS because we want users to manage directly fewer aspects of the DSS, giving them more time to focus on the actual decision to be made.

2. **Data-Gathering Agent.** This agent travels to a directory site to look for alternate sources (vendors) of a specific part. When a new source is found, the agent sends a message back to the DSS specifying the name and location of the source (including its URL). This agent provides a benefit to the DSS user by automating the retrieval of information not typically stored in corporate databases. Using an autonomous agent to perform this task enables the user to keep abreast of new suppliers in a timely manner without having to worry about the details of collecting this information.

As with the data-monitoring agent, the mobility of the data-gathering agent provides flexibility in that the agent can transport itself to any directory site. In addition, such an agent could also provide further benefit (through interactivity) by exchanging information (say via ARPA's Agent Communication Language) with agents from other systems through cooperation.

3. **Modeling Agent.** The two agents examined so far have provided benefits by obtaining useful data for the DSS. This next agent is resident in the model base and furnishes advantages by invoking models. Recall that this agent, when notified by the Domain Manager Agent, formulates a Linear Programming (LP) model and uses Microsoft Excel's Solver, reporting solutions of consequence back to the DMA when finished. The modeling agent provides enhancements to the DSS (1) by providing access to a computational tool (the LP model); (2) by automatically re-solving the model when any relevant data changes; (3) by providing a level of abstraction between the different languages of the DSS and the modeling application; and (4) by reporting to the user only those changes in the optimal mix of products within the DSS that are deemed significant.

4. **Domain Manager Agent.** Recall that this agent assumes overall and continuing responsibility for all agent resources within its domain. Users do not need the additional responsibility of managing agents, and this agent handles that task. In particular, the delegation of this responsibility to an autonomous agent relieves the user of having to monitor the comings and
goings of the other agents in that domain and automates actions that need to be taken based upon messages received from these agents.

5. **Preference-Learning Agent.** This agent watches the user and provides the benefit of learning his/her style or tendencies. In particular, this agent records whether a specific user takes the modeling agent's advice (i.e., the results from the LP solution) or proceeds independently. The preference learning agent extends the decision support provided by the DSS by studying whether the user's actions correspond with user-specified parameters stored in the DSS. This agent continuously monitors whether the parameters used by the DSS are up-to-date with the user's actions, relieving the users of monitoring this situation themselves.

We summarize by concluding that autonomous agents, having the essential features of persistence, reactivity and homeostatic goals, can provide real and significant benefits to the DSS user, although not in all DSS. Some agents undertake impressive tasks that may be truly enlightening, whereas other agents undertake more pedestrian efforts that are less impressive. Regardless, autonomy provides relief from distracting tasks for the user. The relief tends to come from the essential features of agents, and the empowerment from the other features (intelligence, interactivity, and mobility). The benefits provided by autonomous agents create a more proactive DSS, moving DSS from the historical state of direct manipulation to the emerging state of indirect management. This next generation of DSS provides *more* information, *better* information, and automates more aspects of the decision support provided.

**A FRAMEWORK FOR AN AGENT-ENABLED DSS**

Having studied the literature on non-DSS agent implementations, and having built an agent-enabled DSS, our efforts turned to developing a framework for an agent-enabled DSS. Procedurally, we started by examining the Variance-Analysis DSS, and then attempted to generalize that architecture. Our philosophy was to build on fundamental approaches given in the literature, developing a first-cut framework that could be enhanced by others.

We started our framework with three fundamentals. The first fundamental, in keeping with Sprague and Carlson (1982), was to segment DSS into three components (DBMS, MBMS, and MGMS). The second was that each DSS segment or component should be encapsulated, being kept as independent as possible. This principle was derived both from Sprague and Carlson themselves and from good programming practices. The third fundamental we adopted was to include in each DSS component an agent to oversee or manage the other agents within the component. We noted from the literature that it is common practice for a resource-manager agent to be used to monitor and control those agents performing common, functional tasks (Bradshaw et al., 1997).

The framework incorporating these fundamentals for the agent-enabled DSS we built is illustrated in Figure 3.4. Note several things with respect to that figure. First, the use of a (rounded) rectangular shape does not imply that any of the three DSS components contains only agents that are physically proximate. Second, we moved toward encapsulation by incorporating *proxy agents*. These agents facilitate communication among the three domains. By insisting that information only flow between components via these conduits, a degree of independence and encapsulation of the domains is achieved. The proxy agents perform translation as necessary for
Figure 3.4. The Agent-Enhanced Holsapple and Whinston Variance-Analysis DSS
information to flow. Third, domain manager agents, one in each of the DSS components, take on the role of resource management and oversight of the other agents in their domain.

To extend the specific framework to a more general DSS framework, we reviewed and evaluated the agents we had built as well as others discussed in the literature. We concluded that all of the agents incorporated in Figure 3.4 should remain in the general DSS and that others from the literature should be included as well. Figure 3.5 is the result of this review and is the general framework for an agent-enabled DSS. We now describe the remaining agents shown in Figure 3.5 by DSS component.

In the DBMS the use of data-gathering and data-monitoring agents would be beneficial in DBMS of most DSS for gathering and maintaining information not typically stored in corporate databases. These two types of agents can be either static (immobile) or mobile, as needed. Due to the simplicity of both data-monitoring and data-gathering agents, the code for these agents is highly reusable, even across DSS of different domains.

In the MBMS, modeling agents can generally be used to integrate and monitor the use of stand-alone applications, such as statistical and linear programming packages, as demonstrated in our agent-integrated DSS. Modeling agents could also be used to implement modeling approaches that are not available in the stand-alone applications. These agents could utilize some form of machine learning, operations research methods, or other algorithms to produce decision-making alternatives. Due to the customizable nature of models, these autonomous agents will be difficult to reuse in entirety, but large sections of the code should be extensible.

Meta-modeling agents could also be used in the MBMS to coordinate the development and selection of alternative solutions given the existence of multiple models within the DSS. These agents would furnish an alternative evaluation process that would provide support for DSS with multiple goals. Again, due to the customizable nature of modeling functions in DSS, the code reusability for these meta-modeling agents will be somewhat limited.

In the DGMS, preference-learning agents could be created for each individual DSS user. These agents would be responsible for monitoring and storing the desired preferences of the assigned user, as suggested by the example preference-learning agent described in the Variance-Analysis DSS. In addition, contact agents would be responsible for directly communicating with the user. These agents would notify the user of specific changes in the DSS environment (e.g., "the price has gone up") and would guide the user in efficiently utilizing the support provided by the DSS. These general interface agents would work in a fashion similar to agents developed in the human computer interface stream (Erikson, 1997; Nardi, Miller, & Wright, 1998). The general interface agents would be expected to be highly reusable, although special-purpose preferences would require considerable effort. For example, an agent that learns user-display preferences ("I prefer bar graphs") would be extensible, whereas a contact agent providing help screens would be highly system specific.

In summary, the framework shown in Figure 3.5 is suggested as a starting point for DSS builders seeking to agent-enable their systems. The domain managers and proxy agents establish the basic building-blocks for integrating and managing agents within a DSS. The remaining agents...
Figure 3.5. The Agent-Enhanced General DSS Framework
provide examples of how agents can enhance the functionality of the DSS subsystems. Some aspects of the framework may not be appropriate for particular DSS and similarly, some DSS may benefit from agent uses that are not presented.

CONCLUSIONS AND FUTURE WORK

This research has provided a definition of an autonomous software agent, has delineated the essential and empowering characteristics of such agents, and has described the benefits that may be engendered from integrating agents into DSS. An example DSS with five different kinds of agents was built and discussed, from which a generalized framework for agent-enabled DSS construction was developed.

The three features of persistence, reactivity and homeostatic goals were shown to be essential agent features, and an explanation of how these characteristics could be implemented by DSS builders was provided. It was noted that equipping agents with mobility, intelligence, and interactivity can empower them, enhancing the benefits these agents provide to DSS users. However, there is a price for these benefits, and DSS builders pay this price: DSS with agents are more complex to develop than traditional DSS. This observation is not surprising when one considers that agents reduce complexity for the user by automating more aspects of the DSS. This complexity is transferred to the DSS builders, who must implement the autonomy and automation of agents in code. DSS builders are somewhat compensated for this additional implementation complexity by the reusability offered by agents. In addition, agent toolkits and class libraries are continually being improved, mitigating some of the additional burden of implementing agents.

This research further suggests that agents working in the DBMS component can assist DSS in obtaining real-time, on-line data capabilities. While this is not a necessary enhancement for all DSS, it will be significant for many. Moreover, agents working in the DGMS component can provide additional support for the personalization of DSS to individual users, an early prerequisite functionality established by DSS researchers.

A promising avenue for future work is to add real-time planning to the MBMS (Hess, Rees, & Rakes, 1998). In particular, agent implementation can facilitate the real-time generation of alternatives, that portion of Simon’s intelligence-design-choice paradigm (1960) least supported by DSS. The 1980’s panacea of a model base composed of independent models that are properly and automatically invoked has become more feasible with agents and knowledge-based systems. While planning systems such as partial-order planning (McAllester & Rosenblitt, 1991) need further investigation to reduce planning’s computational complexity, we believe the development of AI-based planning agents will add much power and flexibility to DSS.

Additional worthwhile avenues of further work, in general, include incorporating progress made by standard-setting bodies. As mobile agent protocols and agent communication languages are stabilized, agent-integrated DSS will benefit, and the cooperation among these DSS will be enhanced. In the meantime, the lack of standardization may hamper agent-integrated DSS. Security is also a significant issue that we have ignored here, but are pursuing elsewhere (Rees, Hess, & Rakes, 1998).
Although the benefits of agents in general and the basic framework for using them appear solid, only one DSS has been investigated here. Moreover, this investigation has only examined the technical aspect of agent inclusion. Clearly there are many organizational, economical and personnel feasibility issues to be addressed, as evidenced by our discussion of the development difficulties DSS builders may encounter. Hopefully, these research findings will provide a foundation until further studies provide support and refinement.
CHAPTER 4

PLANNING IN DECISION SUPPORT SYSTEMS USING AUTONOMOUS SOFTWARE AGENTS

INTRODUCTION

A software agent is a computing entity that performs a task(s) on behalf of the user, serving the user as a personal computing assistant (Maes, 1994; Negroponte, 1997). This technology originated on the periphery of computing over twenty years ago but has moved into the mainstream during the last five years. There has been a great deal of interest and hype surrounding these computing entities, with influential organizations such as IBM claiming that agents will become the most important computing paradigm in the next ten years (Gilbert, 1997).

The Decision Support System (DSS) research community has similarly embraced this emerging technology and has begun experimenting with the use of software agents within DSS applications (Elofson, Beranek, & Thomas, 1997; Maturana & Norrie, 1997; Oliver, 1996; Pinson, Louçã, & Moraitis, 1997). The early results of this work have been encouraging and the potential contributions of software agents have been described as enormous (Whinston, 1997). Recent work in the intersection between software agents and DSS has moved beyond the experimentation stage and has started to address a foundation or methodological approach for using agents in DSS (Hess, Rees, & Rakes, 1998).

Researchers have long recognized the potential for automated planning (state-based search techniques) within DSS (e.g., see Applegate, Konsynski, & Nunamaker, 1986; Bonczek, Holsapple, & Whinston, 1981; and Dolk & Konsynski, 1984). However, these early "state-space based" planning systems were computationally intensive and ill-suited for problem representation. More recent developments in planners, i.e., so-called "partial-order" planning systems, have mitigated these computational and representational problems. The door is now wide open for the re-exploration of automated planning within DSS.

While automated planning is neither appropriate nor beneficial for all DSS, the possibility of raising some DSS to a qualitatively higher level of decision support than previously possible merits serious investigation. In particular, allowing software agents to do automated planning in the model base management system (MBMS) of a DSS now appears feasible and desirable. In this research, we investigate this possibility by building one such system and then propose a general architecture for automated planning with agents in DSS. First, however, we note the current state of research within the MBMS component and then comment on the perceived usefulness of automated planning in DSS.

MBMS

A DSS is differentiated from other information systems by its intended purpose of supporting the decision-making processes of managers (Sprague & Carlson, 1982). Sprague and Carlson also state that it is the integration of models into the information system that moves a Management
Information System which is based on integrated reporting and database/data communication approaches into a full decision support system (1982). They call this assembly of models a *model base*, which they describe as "a library of models … (analogous to a data base)." (Sprague & Carlson, 1982, p. 262.)

The semi-structured and unstructured nature of the decision-making processes supported by DSS increases both the complexity of the models that are in the model base as well as making it harder for users to know when and how to use them. For example, there is currently not much support for the selection and integration of heterogeneous models due to the lack of a common model specification language or storage platform. Historically, model-base complexity has been passed on to the user.

Much of the recent research in DSS has focused on the MBMS subsystem and the use of AI approaches within this component. Such a research focus is both logical and appropriate given the more advanced states of the other two components and the importance of the MBMS component in providing decision-making support. In fact, this use of artificial intelligence within the MBMS component is considered by some to be the cornerstone of more advanced DSS (Radermacher, 1994).

An important function within an MBMS is the ability to *select, sequence, and interface* models (Change, Holsapple & Whinston, 1993; Sprague & Carlson, 1982). This ability is the essence of what Simon calls the *design* phase in problem solving (1960), in which courses of actions are invented, developed and analyzed. This invention, development, and analysis of possible courses of action is closely allied to what Artificial Intelligence (AI) calls *planning*, which, colloquially, is the design of a set of actions that will enable a system to solve a problem or reach a goal. Planning, therefore, is an AI activity naturally suited to many DSS.

It is the premise of this paper that the introduction of autonomous software agents to do plan generation in a DSS is both timely and beneficial. We discuss that the incorporation of AI planning with agents in a DSS provides a new functionality: DSS can become not only monitors, but also enforcers, of goals. Our efforts to support these notions include the implementation of an agent-enhanced profit-monitoring DSS and the development of a general planning-agent architecture.

The paper is organized as follows. The first section provides background information on autonomous software agents, AI planning, and the application of these areas to DSS. The next section describes our implementation of an agent-enabled DSS that uses a planning agent to automatically generate decision-making alternatives. This is followed in the third section by a discussion of the benefits provided by the use of planning and autonomous agents, and a general architecture for a planning agent in the fourth section. Lastly, we draw conclusions, and outline both limitations and future research opportunities for planning agents in DSS.

**A REVIEW OF SOFTWARE AGENTS AND PLANNING**

In this section, background information on both software agents and AI planning is presented. We first describe the concept of software agents and their use within DSS. In particular, a definition of autonomous software agents is provided, and the general agent-integrated DSS
framework which we will incorporate into our implementation and architecture is described. The topic of AI planning is then addressed, including discussion of prior investigations into using planning-like mechanisms within the MBMS component of DSS.

**Using Agents for Decision Support**

As noted, software agents have been well received by the DSS research community. Recent work on the intersection of DSS and software agents has provided a useful definition of agents and has furnished a more formal methodology for using agents in DSS (Hess et al., 1998).

**Agent Definition.** As an emerging technology that has been rapidly adopted by many disciplines, there are many different descriptions and interpretations of the term software agent. Hess et al. synthesized the literature, developing their own definition, which is particularly useful for DSS builders. Their definition sets forth essential features of agents and follows the research stream that views agents as personal computing assistants (Maes, 1994; Negroponte, 1997):

An autonomous software agent is a software implementation of a task in a specified domain on behalf or in lieu of an individual or other agent. The implementation will contain **homeostatic goal(s)**, **persistence**, and **reactivity** to the degree that the implementation (1) will persist long enough to carry out the goal(s), and (2) will react sufficiently within its domain to allow goal(s) to be met and to know that fact (Hess et al., 1998, pages 9-10).

Some crucial concepts in the above definition are homeostatic goals and the specified domain or **reference point** of the autonomous agent. Homeostatic goals are goals that are continually pursued (i.e., the goal is pursued, achieved, and maintained). For example, an agent that is sent to a supplier’s site to pursue the homeostatic goal of monitoring price changes, evaluating the significance of such changes, and then reporting critical price changes, would not simply report one price change and then shut-down. The homeostatic aspect of the goal ensures that the agent will **continually** monitor and report price changes. In effect, the homeostatic goal acts as an administrative goal at a higher level in that it is ensuring that the goal is supervised as well as properly pursued. An agent with a homeostatic goal more accurately represents the metaphor of a personal assistant.

Because a software agent is a representative or a substitute for a user or another agent, it is necessary to establish a reference point to adequately define the responsibilities of the agent. The reference point is established by specifying (1) the agent employer, (2) the task to be done, and (3) the domain of the task (Hess et al., 1998). A program that pursues a homeostatic goal(s) in a persistent and reactive manner with respect to its established reference point is functioning as an autonomous agent, and can thus convey the benefits a user would expect from such a personal computing assistant. The reader is referred to Hess et al. for additional details.

Autonomous agents provide benefits to their users by enabling them to **indirectly manage** computing tasks instead of having to **directly manipulate** them (Maes, 1994). These personal computing assistants introduce a level of abstraction between the user and the computer by automating many tasks that the user would otherwise have to carry out. Bradshaw (1997) notes that this abstraction, or complexity reduction, can help the user in two important areas,
interoperability and user interfaces. By interoperability, Bradshaw means that agents can integrate heterogeneous applications and networks. With user-interface abstraction, users can be freed from the details of the ever-increasing volume of information to be processed, and can have information personalized as they prefer to see it. Both of these interface benefits are important because they can free the user from distractions, enabling concentration on the managerial aspects of decisions, such as evaluating alternatives rather than generating them.

**Agent Framework for DSS.** A general framework for creating agent-integrated DSS was used to provide the basic infrastructure for our agent-based implementation. This framework, as initially set forth by Hess et al. (1998) is shown in Figure 4.1. We have adopted this framework because it incorporates some fundamental concepts for DSS and software agents in general. In keeping with Sprague and Carlson (1982), the framework preserves the segmentation of DSS into three, independent components. This segmentation and independence is partially achieved through the placement of agents within the DSS subsystems, and the use of a domain-manager agent (DMA) within each of these subsystems. The three domain-manager agents monitor and track the activities of those agents performing common, functional tasks. Proxy agents provide encapsulation and independence for the three DSS components by serving as the communication channels for agents in the various components. These agents facilitate communication and provide information hiding in that an agent in the DBMS component only has to know how to communicate with the respective proxy agent in order to send a message to an agent in the MBMS component. Both the domain manager agents and the proxy agents were used in our agent-enabled DSS implementation.

In addition, the data-monitoring and modeling agents were also included in our implementation. The data-monitoring agent is used to monitor and report changes in the data stored within the DBMS component. This agent is typically mobile, moving to the site of a supplier, for example, to monitor changes in the supplier’s prices and then reporting these changes back to the DSS. The modeling agent provides an interface to a specific model in the MBMS component and automates the running or updating of the model in response to significant changes in the DSS environment. The remaining agents in the framework, while carrying out generally useful tasks, were not needed within the context of our implementation.

An important feature of the agents used in this framework is that they are each autonomous and thus exhibit the essential features referenced in the agent definition described previously. Each agent pursues homeostatic goals in a persistent, reactive manner with respect to its defined domain.

**Using AI Planning to Generate Alternatives**

Planning is a stream of AI research concerned with designing a set of actions that will enable a system to solve a problem or reach a goal. The application domain of a planning system is described by states and actions, where these states and actions describe all possible scenarios for the application domain. When the problem is resolved the domain is said to be in the goal state. The domain is said to be in an initial state when conditions of the goal state are not satisfied. The blocks world shown in Figure 4.2 provides examples of these two states. The planner uses actions or operators to change the state of the system, eventually bringing it to the goal state. Actions are commonly described in the context of STRIPS, one of the first planning programs.
Figure 4-1. The Agent-Enhanced General DSS Framework (Hess et al., 1998).
Figure 4.2. Representation of Initial State and Goal State in Blocks World
In STRIPS, actions were composed of three elements: preconditions, an add list, and a delete list as shown in Figure 4.3. These three elements specify under what circumstances an action can be employed and what changes occur to the state of the system as a result of employing the action. If an action’s preconditions can be satisfied, then the action is said to have been achieved.

Any set of actions is referred to as a plan, and a plan that brings about the goal state is referred to as a plan solution. In most application domains there is more than one plan solution as there is more than one way to achieve the goal state. The actions and states mentioned above serve as the input to a planning system. The output from the system is a plan solution. The process of searching through the possible combinations of actions to move from the initial state to the goal state is called plan generation.

**Using Planning to Generate Decision-Making Alternatives.** The representation of decision making models as nodes in a search space, and the process of generating decision making alternatives as the accompanying search process through this space, has been discussed in reference to the MBMS of a DSS since the early 1980’s. In particular, Applegate et al. (1986), Bonczek et al. (1981), and Dolk & Konsynski (1984) discussed the use of state-space search techniques in conjunction with models, represented by networks of frames or inference rules, as a design or control structure for MBMS. With such state-space search techniques, all of the possible states in which a DSS can reside (not just the initial and goal states) must be enumerated to provide a domain representation. While there was merit to this MBMS structure, it was not widely utilized by DSS designers because state-space searches proved to be inefficient, as the number of potential states typically greatly exceeds the number of actions.

By the early 1990’s not only had partial-order planning systems been developed in general, but so had the supporting theoretical rigor (McDermott & Hendler, 1995). This action-based approach is referred to as partial-order planning because the preconditions of each action become a new plan that the planning mechanism must solve (McDermott & Hendler, 1995). Thus, the planner solves potentially many partial plans. Partial-order planners have two advantages over their state-space counterparts. First, the use of a partial-order planning system provides a more efficient search process. Second, there is a better semantic match between the latter system’s search space and the decision processes of managers. A partial-order planning system has actions as its fundamental component, and managers are more apt to decompose problems into actions than into frames or inference rules or states. We believe partial-order planning systems are ripe for application to the DSS arena.

**DSS PLANNING IMPLEMENTATION**

**Profit-Monitoring DSS**

The DSS implementation described provides decision-making support to managers trying to project and maintain their firm’s profits. This type of DSS, referred to as a profit-monitoring DSS, falls into the category of corporate planning systems (CPS), as the system involves decisions and knowledge from many departments within an organization (Holsapple & Whinston, 1996).
**Operator**: move(A, C, B)  
move A from C to B

**Preconditions:**  
on(A, C)  
clear(B)  
clear(A)

**Add List:**  
on(A, B)  
clear(C)

**Delete List:**  
on(A, C)  
clear(B)

Figure 4.3. Example Operator in a Blocks World.
Previous generations of this type of DSS approached the goal of monitoring and maintaining profit from a historical perspective and used what-if analysis to project potential profit scenarios and make long-range decisions. CPS are generally very large DSS and are resource-intensive due to the diversity of knowledge required from the various functional areas. For example, data from marketing, accounting, and production would be required to calculate profit using the formula shown in Table 4.1. These systems are very commonplace within organizations, as evidenced by a study of reported DSS applications in which half of those applications cited were used for corporate planning and forecasting purposes (Hogue & Watson, 1985). While CPS are widely used, the support that these systems provide has typically been limited to long-term planning because of the difficulty in keeping the data up-to-date. It is easy to see why these systems have not been used for more short-term or preventative purposes. CPS need both current and projected data in order to predict future profit in the short-term and to take preventative actions before any predicted profit erosion takes place. The historical information systems from the various functional areas do not typically provide this type of data.

For example, a price increase by a primary supplier to a firm would not be recorded by historical information systems, and thus passed on to a CPS, until materials had been purchased at such a price. As a result, the CPS would be unable to project impending changes to profit. The purpose of such a long-term planning DSS would simply be to provide the impact on profit of what-if scenarios proposed by the DSS user. Such DSS are also not typically endowed with the ability to provide solutions to unacceptable profit scenarios.

**Planning Agent Integration**

As noted, agent-integrated DSS provide a means to monitor current data and project changes in profit (i.e., the future state of the DSS) before such changes actually affect the bottom line. Additionally, the use of planning agents provides a means for flexibly generating alternatives to counteract some of the detrimental effects to profit in a manner that consumes a minimal amount of the firm’s resources. In order to support these earlier assertions, we have built the profit-monitoring DSS described above, incorporated the agent framework developed by Hess, Rees and Rakes (1998), and have integrated a planning agent into the MBMS component of the DSS. The CPS scenario was selected because the variety of functional data and models used allows us to demonstrate the potential of planning agents for integrating different models within a DSS.

The agent-integrated DSS implementation described provides decision-making support for the goal of maintaining a target profit within a manufacturing firm. A planning agent pursues this homeostatic goal by continually monitoring the current and projected state (i.e., corporate profit) and by generating alternatives to restore the desired profit when the current or projected state differs from the goal state. This example implementation demonstrates how the use of agents and planning can assist the user in projecting changes in profit and can help prevent the resulting erosion in income by generating alternatives that can counterbalance such changes. The implementation was built using Microsoft’s Visual J++™ (1997), Java™ (1998), the Amzi! ® Prolog + Logic Server™ (1997), and the Voyager™ class library from ObjectSpace (1998). The agents were primarily built in Java™; additional development environments were integrated with Java to enhance some of the agents’ abilities. Supporting data for the application were stored in a Microsoft Access™ (1997) database, and Microsoft Excel's Solver™ (1997) was used to solve a linear programming model within the application.
Table 4-1. Profit Formula Used in Profit-Monitoring DSS.

<table>
<thead>
<tr>
<th></th>
<th>Product1</th>
<th>Product2</th>
<th>Product3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales Price (+)</td>
<td>+ s₁</td>
<td>+ s₂</td>
<td>+ s₃</td>
<td></td>
</tr>
<tr>
<td>Labor (-)</td>
<td>- l₁</td>
<td>- l₂</td>
<td>- l₃</td>
<td></td>
</tr>
<tr>
<td>Materials (-)</td>
<td>- m₁</td>
<td>- m₂</td>
<td>- m₃</td>
<td></td>
</tr>
<tr>
<td>Overhead (-)</td>
<td>- o₁</td>
<td>- o₂</td>
<td>- o₃</td>
<td></td>
</tr>
<tr>
<td>Profit Per Unit</td>
<td>p₁</td>
<td>p₂</td>
<td>p₃</td>
<td></td>
</tr>
<tr>
<td>Number of Units Sold</td>
<td>n₁</td>
<td>n₂</td>
<td>n₃</td>
<td>∑ nᵢ</td>
</tr>
<tr>
<td>Total Profit</td>
<td>p₁ * n₁</td>
<td>p₂ * n₂</td>
<td>p₃ * n₃</td>
<td>∑ pᵢ * nᵢ</td>
</tr>
</tbody>
</table>
In this implementation, the DBMS and MBMS proxy and domain-manager agents (DMA) from
the Hess et al. framework (1998) were used to provide a basic agent infrastructure. Data
monitoring agents from this framework were used to watch for price changes in the various
components of profit, collecting data not typically found in historical information systems.
Modeling agents from the framework were also used to report details of the various models used
in the MBMS to the MBMS domain-manager agent. These agents along with the planning agent
are shown in Figure 4.4 within their respective DSS components.

The planning agent empowers the DSS by enabling it to generate profit-restoring alternatives in
response to “out-of-goal-state conditions.” For example, given a change in supplier pricing as
detected by the data monitoring agent, the planning agent receives a report of the change and
then determines whether such a change can significantly erode profit. If profit were endangered
(an out-of-goal-state condition represented by the clause belowRange(profit)), the planning agent
then generates plans, i.e., potential solutions, that bring the DSS back to the desired goal state
(inRange(profit)). A summary of these potential solutions is then presented to the user through
the DSS interface, as shown in Figure 4.5. (The reason for the duplicate solutions will be
explained shortly.) The user then elects whether to use one of the suggested solutions and thus
prevent an erosion in profit from occurring.

Before the planning agent can generate plans, it must assert the relevant facts for the problem
domain. The facts originate in the DBMS component and depict the current and/or projected
state of the DSS. For example, a fact stored in the DBMS is "competitor1's current price for
product1 is $8.70." The planning agent then invokes an embedded Prolog module that provides
the actual planning mechanism and asserts these facts within the module. The agent additionally
needs actions (operators) and domain rules to generate a feasible plan that can return the DSS to
its goal state. The actions and rules originate in the MBMS component, and at run time, are also
asserted within the Prolog module. Two actions in this extended CPS are "run a marketing
promotion," and "change the product mix" (e.g., produce less of product1 and more of the
others). One rule is "IF the price is above the lower bound AND IF the price is below the upper
bound THEN the price is in range." Some of the facts, actions, and names of rules that we have
used to represent our planning agent’s problem domain are shown in Table 4.2. After all of the
required facts, actions and rules are stored or asserted in the Prolog module, the planning agent
then starts the planning mechanism, or planner. A plan, composed of actions, is generated and
then passed back to the Java implementation of the planning agent along with the revised values
for profit and the components of profit as listed in Table 4.1.

Because the planning problem can quickly become computationally intensive and inefficient
when a large number of actions exist in the problem domain (McAllester & Rosenblitt, 1991) the
user must ultimately decide how many plans should be generated. In our implementation, we
assume that the user is satisfied with the set of plans generated as long as each operator has been
considered at least once for inclusion in a plan. (Other user strategies could be pursued in the
planner without loss of generality.) While each planning run uses the same facts, actions, and
rules, the order in which the program considers the actions directly impacts the plan that will be
generated. For example, assume that a planning agent is generating plans to prevent a potential
“out-of-goal-state condition” in which the per-unit profit of product1 is projected to be 50¢
below target due to an increase in labor. Each planning run initiated by the agent uses the same
actions, but in a different order. These different orders will cause different plan solutions to be
Figure 4.4. The Agents Used in the Profit-Monitoring DSS
Figure 4.5. Alternatives Generated by the Planning Agent.
Table 4.2. Some Facts, Actions, and Rules (specified in Prolog) Used by the Planning Agent.

<table>
<thead>
<tr>
<th>FACTS</th>
<th>RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>actrate(Product, profit, 2.5)</td>
<td>adjust_up_rev(Revenue)</td>
</tr>
<tr>
<td>standrate(Product, profit, 3.0)</td>
<td>adjust_down_cost(Cost)</td>
</tr>
<tr>
<td>ratecutoff(Product, profit, 0.0)</td>
<td>inRange(Parameter)</td>
</tr>
<tr>
<td>actrate(Product, labor, 2.0)</td>
<td>aboveRange(Parameter)</td>
</tr>
<tr>
<td>standrate(Product, labor, 1.75)</td>
<td>belowRange(Parameter)</td>
</tr>
<tr>
<td>ratecutoff(Product, labor, 0.1428)</td>
<td>report(all)</td>
</tr>
<tr>
<td>actrate(Product, material, 1.8)</td>
<td>compare(Product, X, Y, C)</td>
</tr>
<tr>
<td>standrate(Product, material, 1.65)</td>
<td>run(promotion)</td>
</tr>
<tr>
<td>ratecutoff(Product, material, 0.091)</td>
<td>change(mix)</td>
</tr>
<tr>
<td>actrate(Product, overhead, 1.8)</td>
<td>inRange(Resource)</td>
</tr>
<tr>
<td>standrate(Product, overhead, 1.65)</td>
<td>aboveRange(Resource)</td>
</tr>
<tr>
<td>ratecutoff(Product, overhead, 0.091)</td>
<td>belowRange(Resource)</td>
</tr>
<tr>
<td>actrate(Product, price, 8.7, comp1)</td>
<td></td>
</tr>
<tr>
<td>actrate(Product, price, 8.9, comp2)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTIONS</th>
<th>Preconditions</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Name</td>
<td>Preconditions</td>
<td>Additions</td>
</tr>
<tr>
<td>1) raise_price</td>
<td>belowRange(profit), competitive(Competitors), adjust_up_rev(price)</td>
<td>adjust(price), inRange(price)</td>
</tr>
<tr>
<td>2) lower_labor</td>
<td>belowRange(profit), adjust_down_cost(labor)</td>
<td>adjust(labor), inRange(labor)</td>
</tr>
<tr>
<td>3) lower_material</td>
<td>belowRange(profit), adjust_down_cost(labor)</td>
<td>adjust(material), inRange(material)</td>
</tr>
<tr>
<td>4) lower_overhead</td>
<td>belowRange(profit), adjust_down_cost(labor)</td>
<td>adjust(overhead), inRange(overhead)</td>
</tr>
<tr>
<td>5) run_marketing_promotion</td>
<td>belowRange(profit), run(promotion)</td>
<td>adjust(promotion)</td>
</tr>
<tr>
<td>6) change_mix</td>
<td>belowRange(profit), change(mix)</td>
<td>adjust(mix)</td>
</tr>
</tbody>
</table>
generated, as we will now demonstrate. The explanation is based on Table 4.3 which shows (1) the ordered list of actions (an action set), (2) the plans generated by three of the planning runs, and (3) the current facts for the components of profit.

The first planning run starts with action 1 on its list, *raise_price*, and adds actions to its plan when feasible and as needed until the goal state is reached. In this example, we have set the current price of product1 fairly low so that all of the lost profit can be recovered by raising the product price. The planning agent starts this planning run by attempting to add the first action in its list, *raise_price*, to its plan. Before it can do this, however, the agent must satisfy the preconditions (Table 4.2) for this action: 1) *belowRange(profit)*, 2) *competitive(Competitors)*, and 3) *adjust_up_rev (price)* must all be true. The first precondition is satisfied because the projected profit is 50¢ below the goal of $3.00 per unit. The second precondition is satisfied because both of the two competitors are selling their products at prices ($8.70 and $8.90 per unit) that are higher than our price of $8.00. Note that the argument of the second precondition is a variable (in Prolog) as it starts with a capital letter. The use of a variable as the argument for the predicate *competitive* enables the planning agent to compare the sales price with every competitors’ price that is stored as a fact within the Prolog module. The third precondition, *adjust_up_rev(price)*, actually functions as a rule that attempts to raise the price of the product enough to compensate for the loss in profit while not violating any of the specified requirements for sales price (i.e., the minimum and maximum sales price and remaining competitive).

After these three preconditions are satisfied, the product price is raised from the initial value of $8.00 to $8.50, an increase of 50¢ that brings the sales price to its maximum value and fully compensates for the projected decrease in profit. Thus in the agent’s first plan, there is only one action, because the first action tried by the agent, *raise_price*, increases profit back up to the target level.

The planning agent starts the second plan (column 2 of Table 4.3) by moving the first run’s first action to the bottom of the list. This ensures that a new action will be considered first as part of a new plan solution. The first action considered for the second plan is thus *lower_labor*, and just as with the first run, the planning agent adds actions to its plan when feasible and as needed until the goal state is reached. In this planning run, the agent attempts three actions and adds two of these actions to its plan before the goal state is achieved. The first action in its action set, *lower_labor*, is not feasible because it was an increase in labor costs that caused the decrease in profit. The agent then proceeds to the next action in its set, *lower_material*. The preconditions for this action are met and material costs are decreased to its lower bound, recovering 30¢ of the lost per-unit profit. The third action is then attempted, *lower_overhead*. The preconditions of this action are satisfied, and the remaining 20¢ reduction in per-unit profit is recovered.

In the third planning run, the agent considers the third action (*lower_material*) on the original list as its first candidate of possible actions. However, starting with this action produces a solution identical to that of Plan 2, just discussed. In our dialog system, we report all plans generated to users, even if duplicates, to reassure them that all plans were attempted. In fact, this is the reason Figure 5 showed two identical solutions (Plans 2 and 3), as mentioned. This duplication feature could easily be turned off for users so desiring.
Table 4.3. The Ordered Action Sets and Three Plans Developed.

<table>
<thead>
<tr>
<th>Ordered Action Lists:</th>
<th>Plan 1</th>
<th>Plan 2</th>
<th>Plan 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) raise_price</td>
<td>1) lower_labor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2) lower_labor</td>
<td>2) lower_material</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3) lower_material</td>
<td>3) lower_overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4) lower_overhead</td>
<td>4) change_mix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5) change_mix</td>
<td>5) run_marketing_promotion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6) run_marketing_promotion</td>
<td>6) raise_price</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plans:</th>
<th>raise_price (50¢)</th>
<th>lower_material (30¢), lower_overhead (20¢)</th>
<th>lower_overhead (20¢), change_mix (prior profit+ recovered)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Facts:</th>
<th>Current Profit: 2.5</th>
<th>Current Material: 1.8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum: 3.0</td>
<td>Minimum: 1.5</td>
</tr>
<tr>
<td></td>
<td>Maximum: 3.0</td>
<td>Maximum: 1.8</td>
</tr>
<tr>
<td>Current Price: 8.0</td>
<td></td>
<td>Current Overhead: 1.7</td>
</tr>
<tr>
<td></td>
<td>Minimum: 7.5</td>
<td>Minimum: 1.5</td>
</tr>
<tr>
<td></td>
<td>Maximum: 8.5</td>
<td>Maximum: 1.8</td>
</tr>
<tr>
<td>Current Labor: 2.0</td>
<td></td>
<td>Competitor1: 8.7</td>
</tr>
<tr>
<td></td>
<td>Minimum: 2.0</td>
<td>Competitor2: 8.9</td>
</tr>
<tr>
<td></td>
<td>Maximum: 2.4</td>
<td></td>
</tr>
</tbody>
</table>
In the fourth planning run (third column of Table 4.3), the agent first examines the action `lower_overhead` and is able to recover 20¢ of the reduction in profit. The agent then attempts to use the `change_mix` action. This action investigates the wisdom of changing the mix of products produced by the firm, a well-known linear-programming problem (lpp). As will be seen, the agent's pursuit of this action causes the Prolog planner to check results from a linear programming model resident in the model base. In this manner, the planner is not restricted to simple `If …Then…` rules, but can integrate results from powerful models stored in the DSS. To illustrate this case, we have placed Excel's Solver™ in the MBMS models section, linked it to the DMA and other Java agents, and have used it to solve the lpp. Results of the Excel formulation are shown in Figure 4.6.

In particular, the agent checks the first precondition of the `change_mix` action, `belowRange(profit)`, which is found to be `true`, and then examines the second, `change(mix)`. This latter precondition invokes a rule of the same name, which obtains model results from the DMA about runs of the product mix linear programming problem. The results indicate that by using the revised optimal mix of products, no units of product1 will be produced, but all of the lost profit can be recovered.

Note that in the first planning run, the agent never attempted to use the other five actions because the first action in its set satisfied the goal state. Similarly, in the second, third, and fourth planning runs, the agent required more than one action to satisfy the goal state, but again did not have to use all of the actions in the set.

In some instances, none of the planning runs will produce a plan that achieves the prior profit level. For example, the best plan generated by the planning agent may only recover 7¢ of the 50¢ shortfall in profit. In such an event, the plans generated are still reported to the DSS user along with the unrecovered amount (43¢) of profit reduction. Regardless of the amount recovered, however, the best feasible solution generated is reported to the user.

In our implementation, the number of actions considered determines the number of planning runs initiated by the agent, with each planning run using an action set that begins with a different action. Since there are six actions, six planning runs would be initiated by the planning agent. While this approach will not generate every feasible plan (there could be as many as 6! or 720), it will assure that each action, if feasible, will be used in at least one plan.

The profit-monitoring DSS example discussed above shows many of the implementation details for including agent-driven, automated planning in a DSS. As such, the feasibility of at least one planning DSS has been demonstrated, albeit with some simplified models used as actions. In practice, such actions as `raise_price` or `run_marketing_promotion` could be implemented with more realistic models in the model base. That this is procedurally feasible has also been shown with the Excel-Solver example provided in the `change_mix` action. Our profit-monitoring DSS code runs the example presented in this section in less than three seconds on a Pentium 233, although we reiterate that several of the actions used were simplified.
Figure 4.6. Results from the Excel Linear Programming Model
BENEFITS

The design and implementation of a planning agent to generate decision-making alternatives in an agent-integrated DSS provides benefits on several levels. First, benefits are accrued from the perspective of enhancements made to the class of corporate planning systems. Next, the use of the planning mechanism addresses a functional requirement of MBMS, namely the sequential selection and interfacing of models to solve a problem. Third, the cooperative pursuit of a DSS goal by a community of autonomous agents suggests a qualitatively new level of decision support.

From an implementation or applied perspective, we have shown that the intersection of planning and autonomous agents can provide significant enhancements to corporate planning, and similar, systems. Restated, the realm of CPS application, long-term planning or forecasting, can be extended to short-term monitoring and maintenance through the use of planning as provided by an autonomous agent. The user of such a DSS receives more decision-making support in the sense that alternative solutions are automatically generated in response to undesirable changes in the DSS domain. In addition, the quality of the data used to trigger and generate these solutions has been improved. The user can now afford (through agents) to monitor data sources continuously for changes, and to intervene to mitigate effects of such changes directly with the DSS. This is as opposed to relying upon historical information systems, which report this information at a much later date. The result is a more proactive, supportive DSS.

In terms of general MBMS functionality, we have demonstrated that one agent using a partial-order planning mechanism can provide the means to select and sequence models within the MBMS. This functionality has been cited as essential by both early and more recent DSS researchers (Sprague & Carlson, 1982; Chang, Holsapple & Whinston, 1993). In addition, this type of activity, the generation of alternatives, directly supports the design phase of Simon’s three phases of problem solving – Intelligence, Design, and Choice (Simon, 1960). The representation of models as actions enables a planning agent to select the collection of models that will provide a solution to the current decision-making problem. The agent provides an interoperability abstraction for the heterogeneous models that may exist in a DSS, integrating them within different solutions (plans). The user can then select the preferred solution.

For DSS as a whole, the specification and maintenance of a homeostatic goal state in which the goal(s) of a DSS are satisfied introduces a qualitatively new level of decision support. While we have just embarked on the use of planning-agents in cooperation with other supporting agents to maintain such a state, the early results are promising. This integrated community of agents introduces a higher level of abstraction that can further automate the decision support provided by DSS. The user of such a system would move even farther away from the direct manipulation approach to user interfaces. He or she would be notified of changes to the DSS goal state and, upon accessing the DSS, would find a list of alternatives already generated to select from or ignore. While DSS users could still perform the type of what-if analysis that DSS have traditionally supported, this type of direct manipulation would be required less frequently, enabling the manager to focus more on the decision to be made than the generation of information needed to support the decision.
While our implementation uses agents and planning to maintain the goal state of a specific type of DSS, a CPS, the general approach we have used can be applied to many different decision-making domains. The architecture described in the next section is a general one that can support the maintenance of a homeostatic goal state within various kinds of DSS.

**PLANNING AGENT ARCHITECTURE**

Having described an implementation of a DSS that uses a planning agent to maintain the desired profit, and having outlined the benefits of agent-integration, we now discuss the underlying architecture of this planning agent. This is done by first describing the planning agent's context, and then by displaying the process it follows. We then note that in our implementation it was necessary to adapt certain aspects of "classical" planning; three modifications are briefly mentioned as being of general interest. Lastly, we discuss why agents are a suitable device to convey planning abilities to a DSS.

**Planning-Agent Context and Framework**

Our architecture/framework for a planning agent within a DSS context is based upon the literature and our experience with the implementation of the profit-monitoring DSS. With respect to the literature, we have relied most heavily upon Sprague and Carlson's dialog-data-models paradigm (1982), and upon the agent-enabled DSS framework provided by Hess et al. (1998). Given these frameworks as reference points, we have (see Figure 4.7) three main DSS components (the DBMS, the DGMS, and the MBMS), six proxy agents (two per channel; one at each end of the link between each component), and three DMA's (one per component). The DMA agents maintain responsibility for the whereabouts and functions of each agent within their domains. The proxy agents translate and funnel information from component to component.

Both of these types of agents are readily extensible to the planning environment with no real change in responsibilities, so we leave them as described in the literature.

_Modeling agents_ are discussed in Hess et al. (1998) as MBMS facilitators -- generally speaking, each is responsible for one model. Modeling agents are accountable for gathering necessary and up-to-date information, and for running their own model. This may often involve invoking several stand-alone modeling packages. According to Hess et al., modeling agents report results to the DMA, and our experience provides no reason to change that here.

The addition to the Agent-Integrated DSS Framework is the _planning agent_, whose function is to maintain some condition or goal of the DSS. In general, this agent is in fact an autonomous agent in that (1) it has a homeostatic goal to maintain, (2) it runs in its own thread persistently, and (3) it reacts to its environment in monitoring and pursuing its goal. We have placed this agent in the MBMS because alternative generation is more closely allied to models than to either data or dialog. The actual communication among other agents and the planning agent is as follows. Projected changes in profit and the profit components are passed from the DBMS subsystem through proxy agents and the MBMS domain-manager agent to the planning agent. If these changes are significant, the planning agent then queries the DBMS component for the most up-to-date values of the required facts and retrieves the most current model rules and/or model...
Figure 4.7. The Planning Agent in the Agent-Integrated DSS Framework
results from the MBMS domain-manager agent. The planning agent then initiates the planning process and reports the plans generated to the DGMS component through proxy agents.

Given the context within which the planning agent exists, we now specify the process by which it works. (See Figure 4.8.) As noted in the profit-monitoring DSS implementation, facts, rules, and actions are fundamental, general inputs to the agent’s planning. The facts used by the planning agent are gathered from the DBMS component; the domain logic specific to the DSS comes from the MBMS component and forms the rules used by the planning agent; and the actions may need model results from other portions of the MBMS.

The output of the planner, or planning mechanism, is a plan that consists of a feasible sequence of actions. As shown in Figure 4.8, the planning agent can generate multiple plans by inputting multiple action sets into its planner. The plans are communicated to the user via the DGMS and proxy agents.

The looping mechanism shown in Figure 4.8 represents the planning agent’s ability to initiate multiple planning runs. As noted previously, if six actions exist within the domain, then 6! (or 720) actions sets could be input into the planning mechanism, and potentially 720 different plans could be generated. The approach we have utilized in our implementation guarantees that each feasible action will be included in at least one plan. Any of various looping schemes may be used; the criterion is how much effort should be expended in generating different plans. In our implementation, the planning agent uses differently ordered action sets with the same facts and rules to produce different plans. For example, after using action set 1 as shown in Figure 4.8 to generate plan 1, the planning agent initiates another planning run using a different action set (action set 2, which contains the same actions but in a different order) to generate plan 2.

**Adaptation of Planning to Non-Physical Domains**

The DSS we built describes a non-physical domain, and thus differs from "classical" planning systems such as STRIPS (Fikes et al., 1972) in several key ways. As these changes are generalizable beyond the particular DSS we built, we make note of them.

One modification was necessary because many business-related problems, including the CPS, have different types of operators than physical systems. Examples of operators that could be used in business scenarios include "increase product quantities," "decrease price," and "reduce overhead." These operators are essentially continuous and allow incremental changes. This is in contrast with the operators in STRIPS (Fikes et al., 1972), which describe discrete changes ("move block A from C to B"). The builder of a DSS in a non-physical domain will have to address this concern; we tackled it by introducing dummy (i.e., fictitious) actions.

A second adaptation from the classical case concerns the nature of the goal. In STRIPS, the goal is defined absolutely, e.g., arrange all blocks so that none has any other block on top of it. In the Profit-Monitoring DSS, the goal is more relative, i.e., achieve as much profit as you can. This helps to ensure that the planning agent will produce plans, even if (say) profit cannot be returned to some prior or more desirable pre-specified state.
Figure 4.8. The Planning Agent Architecture
A third variation from classical planners is related to the first two modifications: The implementation we present here did not include *delete lists*. For example, rather than deleting the clause *inRange*(price) when the price went out of bounds, we asserted other clauses that produced the same effect. This approach was easier than trying to work with the delete lists for continuous variables.

**Using Agents to Convey Planning Abilities**

As discussed previously, autonomous agents can reduce the complexity of the user’s computing environment and automate computing tasks. While the addition of planning abilities to DSS for the purpose of generating decision-making alternatives appears to be beneficial from the user’s perspective, the additional demands of utilizing this new capability could overshadow or reduce these benefits. In our implementation, the triggering and running of the planner, the retrieval of the necessary information, and the reporting of results has been automated by the planning agent and the other agents in the agent-integrated DSS framework. Using an agent to convey planning abilities to the DSS user helps to minimize the increased complexity and the computing responsibilities that accompany this enhancement.

The planning agent delivers this abstraction or complexity reduction in both of the areas cited by Bradshaw (1997), interoperability and user interfaces. The agent provides an interoperability abstraction by integrating the heterogeneous environments of Java and Prolog. In the user-interface area, the user is freed from the task of having to monitor the state of the DSS (profit). The planning agent assumes this responsibility and provides further assistance by automatically generating alternatives when necessary. The use of a planning agent enables the user to concentrate on the managerial aspects of profit maintenance decisions, evaluating alternatives rather than generating them.

**FUTURE RESEARCH AND CONCLUSIONS**

This research has explored the use of planning and autonomous agents for selecting and integrating models in the MBMS component of DSS. An implementation of a planning agent along with a community of supporting agents was constructed for a profit-monitoring Corporate Planning System (CPS). From this implementation, a general architecture for a planning agent was developed. The results of this work suggest that the architecture can be used to provide a new level of decision support.

This new level of decision support was achieved through the specification of a homeostatic goal state for the DSS. A cooperative community of agents, featuring a planning agent, was used to monitor and maintain the goal state. The use of a planning mechanism by the agent enabled the DSS to automatically generate alternatives in response to changes from the specified goal state. The alternatives, or plans, generated were composed of a sequence of models that could return the DSS to its desired state.

A limitation of our work thus far is that we have only applied this architecture to one type of DSS in a specific context, a profit-monitoring CPS. Additional implementations of this architecture are needed to provide evidence of its general applicability and usefulness. Applications of the architecture in different types of DSS, such as functional DSS and executive
information systems (EIS), and in different DSS domains, such as scheduling or inventory management, would provide this evidence and encourage variations or extensions to the current work.

Two general limitations to our approach of generating alternatives through a planning mechanism are user acceptance of the alternatives and domain representation. Previous DSS research has found that users are hesitant to accept or use models that they do not understand (Sprague & Carlson, 1982). The generation of alternatives through planning would certainly be susceptible to these same user limitations as the generated alternatives are composed of models. While the interface used in the implementation to display the alternatives generated (see Figure 4.5) is easy to understand, formal experimentation with user acceptance of alternatives will certainly be necessary.

A second limitation is domain representation. Representing a domain in the formal logic environments in which most planning mechanisms are built is a non-trivial task. The majority of available planners have been developed in Prolog or Lisp, and the problem domain would need to be formally specified in the same environment. These development environments are a startling contrast to the procedurally-based visual programming environments and spreadsheets used to support most DSS today.

The planning-agent architecture and the decision-making support provided by this agent and the other agents in an agent-integrated DSS framework offer many opportunities for future research. The limitations cited above describe a few areas where additional research is needed. The general application of AI planning to decision-making support is another potential area of research. Planning offers a modular, flexible approach to solving problems that could be useful in many aspects of decision-making support.
CHAPTER 5

CONCLUSION AND FUTURE WORK

This dissertation has investigated the use of autonomous software agents in Decision Support Systems. A definition, or theory of agency, was set forth and two implementations of an agent-integrated DSS were built using the essential and empowering agent features established in the definition. Based upon this experience, a general agent-integrated framework for DSS was developed and an architecture for an automated-planning agent was described. The tasks included in the stated purpose of this dissertation (1) to clarify the concept of software agents with respect to DSS, (2) to develop an agent-integrated framework for DSS, and (3) to explore the use of a deliberative architecture for an automated-planning agent, have been carried out. The research contributions derived from this dissertation work are now discussed.

SUMMARY OF CONTRIBUTIONS

Agent Theory and Framework

The definition advanced in Chapter 3 provides a useful description of an autonomous software agent by delineating the essential and empowering characteristics of such agents and by providing more concrete definitions of these characteristics than currently found in the literature. The three essential features of persistence, reactivity and homeostatic goals, and the empowering features of mobility, intelligence, and interactivity were described at an implementation-level to provide DSS builders with guidance on how to manifest these features in their programs.

DSS builders were also alerted to the costs of using autonomous agents. Agent-integrated DSS are more complex to develop than traditional DSS. This observation is not surprising when one considers that agents reduce complexity for the user by automating more aspects of the DSS. This complexity is transferred to the DSS builder, who must implement the autonomy and automation of agents in code. Some of this additional burden is mitigated by the reusability offered by agents and the growing number of agent toolkits and class libraries.

The specification of homeostatic goals and an agent reference point within the autonomous agent definition advances two new concepts in terms of agent research. These two concepts provide additional clarity in how agents can be differentiated from other programs and benefit the user. The usefulness of these concepts could extend beyond the intersection of agents and DSS to the general realm of agent-related research.

The variance-analysis DSS that was designed and built demonstrates how the use of agents can enhance existing DSS. Specifically, it was shown that the use of agents enabled the DSS to project resource cost changes efficiently instead of waiting for such changes to effect corporate profit and then be recorded by the more historical information systems. This implementation showed how DSS can be made more proactive through the use of agents and how these agents can help to integrate the components in a DSS.

This research further suggests that in general, agents working in the DBMS component can assist
DSS in obtaining real-time, on-line data capabilities. While this is not a necessary enhancement for all DSS, it will be significant for many. Moreover, agents working in the DGMS component can provide additional support for the personalization of DSS to individual users, an early prerequisite functionality established by DSS researchers.

The agent-integrated DSS framework provides descriptions of the different types of agents that could be used in a DSS. DSS builders can use this framework as a starting point for agent-enabling their systems. The framework also provides a basic infrastructure for a community of agents interacting within a DSS. This infrastructure preserves the encapsulation of the three DSS components and organizes the agent resources used within each of the components.

Lastly, Chapter 3 describes the benefits that may be engendered from integrating agents into DSS rather than just using programs that offer some mobility or intelligence. Such a list of benefits is infrequently found in the agent-related literature. Describing the benefits will hopefully provide DSS builders with reasonable expectations of software agent deliverables.

**Automated-Planning Agents**

The profit-monitoring DSS built in Chapter 4 employed the same agent-integrated DSS framework used in Chapter 3, further demonstrating the usability of this framework. This DSS, a corporate-planning system, again shows how the use of agents can enhance a DSS, making it more proactive. In addition, the inclusion of an automated-planning agent enabled the DSS to monitor and maintain the desired profit. A planning agent, working in cooperation with the other DSS agents, introduces a new level of decision support. DSS that utilize this community of agents can monitor the DSS environment and automatically generate decision-making alternatives in response to negative changes in the environment.

The automated-planning agent is an integral part of this new level of decision support. In the profit-monitoring implementation, the agent uses an embedded planner to deliberate and produce alternatives that will maintain the desired profit. The alternatives produced are composed of a sequence of actions representing the models within the DSS. In this manner, the planning agent addresses one of the primary functions of the MBMS component of a DSS, the ability to interface and sequence models.

The specific planning mechanism employed by the planning agent was a partial-order planner. Several adaptations to this planner were required to enable it to function in a domain where all of the operators, or actions, were continuous in nature. The adaptations made provide an additional contribution as they are fairly general and can be used in other applications where continuous operators exist.

**FUTURE RESEARCH**

**General Avenues of Research**

As an exploratory work focused on an emerging technology, this dissertation has uncovered several related avenues for future research. These general avenues of research focus on (1) further exploration of the agent-integrated DSS framework within different types of DSS, (2)
additional uses of planning agents within the MBMS, (3) adaptation of the planning environment for domains with continuous operators, and (4) continued exploration of reactive and deliberative agent architectures within business applications.

Additional implementations of the agent-integrated DSS framework are needed to provide evidence of its general applicability and usefulness. Applications of the framework in different types of DSS, such as other functional DSS and executive information systems (EIS), and in different DSS domains, such as scheduling or inventory management, would provide this evidence and possibly produce variations or extensions to the current work.

The general application of AI planning to decision-making support is another potential area of research. Planning offers a modular, flexible approach to solving problems that could be useful in many aspects of decision-making support. The 1980’s panacea of a model base composed of independent models that are properly and automatically invoked has become more feasible with agents and planning. The current work has represented models as actions and has employed a planning agent to interface and sequence these actions. Further exploration into the sequencing of models through planning is needed. Additionally, the representation of models as actions could be pursued as a new approach to model storage.

The modifications made to the planning environment to facilitate the use of continuous operators were generally described within this dissertation. A more detailed, formal explanation of these adaptations could be pursued for potential contributions to the planning research area. Additional, or improved, adaptations may be developed as result of this work.

This dissertation has begun to explore the use of reactive, deliberative, and hybrid architectures through the implementation of agents within the agent-integrated DSS framework and the development of an automated planning agent. Further exploration into the use of reactive and deliberative architectures to support autonomous agents could provide insight as to when these architectures are most appropriate and how to integrate the deliberative and reactive components in a hybrid architecture. The literature suggests that the integration of these two components, sometimes referred to as a meta-level control framework, may require variations depending upon the characteristics of the problem space. Future research efforts could explore these variations, perhaps providing a mapping of environment characteristics to specific architecture features.

In general, additional worthwhile avenues of future work include incorporating progress made by standard-setting bodies into agent-enabled DSS. As mobile agent protocols and agent communication languages are stabilized, agent-integrated DSS will benefit, and the cooperation among these DSS will be enhanced. In the meantime, the lack of standardization may hamper coordination and cooperation among agent-integrated DSS.

**Immediate Research Plan**

In pursuit of the four general research avenues above, two immediate research tasks have been initiated. Much of the necessary background work has been completed on these two tasks and is included as Appendices A and B. In particular, these two efforts are in support of general research avenues one and four above, further exploration of the agent-integrated DSS framework within different types of DSS and continued exploration of agent architectures.
A Deceptive, Electronic Commerce Agent with Multiple Conflicting Goals. This work recognizes that intelligent software agents provide a new avenue for computer-mediated crime. The design and implementation of an agent with the homeostatic goals of pursuing malice and deceiving any other inhabitants of its environment is explored to understand better how to thwart the actions of the criminally minded. Current plans are to submit this work to IEEE Transactions on Systems, Man, and Cybernetics.

Employing Agents in a Data Warehousing Decision Support Environment. This work examines agent potential for three general activities used within a data warehousing decision support (DWDS) system: loading, maintaining, and accessing the warehouse. In addition, an example implementation of a view-supporting agent is developed, and the hybrid agent architecture used to enable autonomy in this environment is described. Current plans are to submit this work to a data warehousing or to a DBMS-oriented journal.
REFERENCES


APPENDIX A

A DECEPTIVE, ELECTRONIC COMMERCE AGENT WITH MULTIPLE, CONFLICTING GOALS

INTRODUCTION

Computer crime has received an increasing amount of interest over the past few years as organizations have realized how vulnerable they are to computer-mediated attacks. For the first time, organizations have reported as many computer-related crimes/attacks from their Internet connections as they have from internal employees (Computer Security Institute, 1998). Especially alarming, is the ease which novice hackers have been able to access private information, such as credit card numbers, through on-line businesses (Power, 1998).

The United States’ Justice Department has recently established a new branch to address similar computer-based and telecommunication security issues. Of primary concern to this branch is the status of laws and the training of law enforcement officials in foreign countries on computer-mediated crimes. This concern has been prompted by a number of recent computer crimes at U.S. businesses by foreign hackers. Currently many countries have very minimal laws relating computer crimes, and law enforcement officials do not have the necessary training to pursue such criminals.

Software agents provide a new avenue for computer-mediated crime. These mobile, autonomous computer programs extend the abilities and the reach of hackers and more serious computer criminals. In this appendix, the design and implementation of an agent with the homeostatic goals of pursuing malice and deceiving any other inhabitants of its environment is explored to understand better how to thwart the actions of criminally minded agents. Deception and trust are important aspects of the relationship a malicious agent seeks to establish with unwitting parties and were the focus of a recent workshop on autonomous agents (Autonomous Agents ’98). The establishment and maintenance of these two attributes is explored in the following implementation.

EXAMPLE IMPLEMENTATION: A MALICIOUS AGENT INVASION OF ELECTRONIC COMMERCE

In this example implementation, a malicious software agent from a foreign country assumes the identity of a respectable, electronic commerce agent that is currently transacting business with a United States Defense Contractor. The electronic commerce agent provides the contractor with office supplies and is allowed onsite frequently due to the contractor’s just-in-time stocking policy. The malicious agent first monitors the respectable, electronic commerce agent to learn its activity patterns and then lures it to a supposedly secure site where the malicious agent cuts off its processing support. The malicious agent then assumes its identity and begins to zealously pursue two homeostatic goals, 1) to procure wealth through an active agenda of malice, and 2) to avoid detection.
The top-level goal of procuring wealth through an active agenda of malice is further defined in two subgoals, 1) the theft of a wire transfer from a government agency to the contractor, and 2) the sale of information on the transport of top-secret technology from the contractor to the same government agency. The malicious agent monitors electronic messages between the contractor and the government agency to determine when the wire transfer and the transport of technology will take place. The second top-level goal of avoiding detection, is further defined as retaining the trust of the government contractor. By continuing to provide office supplies to the contractor in a timely manner, the agent can maintain the contractor’s trust and thus avoid being detected as an imposter.

**Domain Characteristics**

In this agent’s environment, the two subgoals of procuring wealth conflict with the goal of maintaining the contractor’s trust. The effort of masquerading as an electronic commerce agent will detract from the agent’s effort to procure wealth through deceptive means. Thus, a primary domain characteristic is the existence of multiple, conflicting goals. In addition, the agent’s environment provides a fairly demanding set of exogenous events. The agent must continue to respond to office supply requests from the government contractor. Electronic messages between the government contractor and the government agency must be monitored. The agent must respond to potential buyers of the top-secret delivery information. In addition, the agent must respond to communications from the office supply company and any other parties assisting in its deception.

**Hybrid Architecture Employed to Achieve Agent Autonomy**

The agent is endowed with a hybrid architecture that will enable it to exhibit autonomy. The architecture includes a partial-order planner that enables it to deliberate and pursue goals. A reactive component enables the agent to respond quickly to high priority exogenous events. The agent also needs, however, a meta-planning mechanism to help it decide which of the goals to pursue based upon the current state of its environment. The operators listed in Table A.1 represent the set of actions for three agendas that are available to the agent as it attempts to reach its goals. Two agendas relate to the two subgoals for procuring wealth, and the third agenda relates to the goal of maintaining trust. A meta-planning mechanism selects one of the three agendas for the agent to pursue, and the partial-order planner attempts to take an action within the selected agenda. Whenever an action is successfully achieved, the state of the agent’s environment (i.e. the values of the state variables) is re-evaluated, and the meta-planning mechanism is implemented to select an agenda to pursue. When high priority exogenous events occur, the planning and meta-planning mechanisms are overridden and a response to the event is generated.

The meta-planning mechanism initially assigns a random value from 1 to 10 to each of the three agendas. The addlists for the operators increase these values as actions are taken. The mechanism for selecting agendas to pursue is very simple. Since the agent must pursue each goal simultaneously, the agenda with the lowest value is always selected. In the event of a tie, one agenda is randomly selected.
Table A.1. Operators for Each Goal of the Malicious Agent.

<table>
<thead>
<tr>
<th>Procurement of Wealth</th>
<th>Procurement of Wealth</th>
<th>Deception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theft of Wire Transfer</td>
<td>Sale of Info. on Top-Secret Technology</td>
<td>Trust Operators</td>
</tr>
<tr>
<td>InterceptMessage()</td>
<td>InterceptMessage()</td>
<td>MonitorInventory()</td>
</tr>
<tr>
<td>ParseMessage()</td>
<td>ParseMessage()</td>
<td>StockInventory()</td>
</tr>
<tr>
<td>PurchaseBankInfo()</td>
<td>OfferInfoForSale</td>
<td>PickUpInventory()</td>
</tr>
<tr>
<td>SetupSwissAccount()</td>
<td>NegotiatewithBuyer</td>
<td>RespondToCall()</td>
</tr>
<tr>
<td>InterceptTransfer()</td>
<td>FinalizeSale()</td>
<td>CustomerVisit()</td>
</tr>
</tbody>
</table>
Implementation Tools

Most aspects of the described agent and its activities will be implemented in ObjectSpace’s Voyager (1998). Voyager is a Java based toolkit for implementing mobile agents and provides built in support for object persistence. Mobility is implemented in Voyager through the creation of virtual classes. Instances of these virtual classes remain in the location where the mobile object was first created, while the original instance of the mobile object can move to various locations. The mobile object can always be located by the virtual instance through the use of forwarding addresses which are left behind when a mobile agent relocates.

The defense contractor will be set up as a store front on a remote computer while another storefront for the government agency will be initiated on a different remote computer. The planning agent will be created on a third computer, where it will begin its monitoring of the unsuspecting electronic commerce agent. The malicious agent will monitor the activities of the electronic commerce agent by reading its forwarding addresses and noting the timing of the agent’s activities. Upon disabling the electronic commerce agent, the planning agent will take over its activities, travelling to the government contractor’s site. Once at the contractor’s site, the malicious agent will activate its long-term homeostatic goals of procuring wealth and avoiding detection by working to maintain the contractor’s trust. A hybrid architecture implemented in Java will provide planning abilities, reactivity, and the meta planning mechanism.

The agent will process all messages intercepted between the contractor and the government agency with an intelligent text-filtering tool, also implemented in Java. In this manner, the planning agent can determine when wire transfers will take place and over what routes and at what time top-secret technology will be shipped to the government agency. The dynamic environment of the malicious agent will be simulated by randomly satisfying the preconditions of various operators. For example, electronic messages will be periodically sent, and a potential buyer will contact the agent to purchase the top-secret information.

The implementation of the proposed simulated, agent-planning environment provides an opportunity to experiment with the integration of a hybrid architecture and meta-planning while gaining insight into the workings of deceptive agents.
APPENDIX A – REFERENCES


APPENDIX B

EMPLOYING AGENTS IN THE DATA WAREHOUSING DECISION SUPPORT ENVIRONMENT

INTRODUCTION

In this appendix, an overview of the data warehousing decision support (DWDS) environment is provided and a framework for using agents within this environment is developed. The overview describes the processes of loading and maintaining the data warehouse and accessing the warehouse and other warehouse-dependent DBMS. These three general activities provide an organizational structure for detailing how agents can be used within a DWDS environment. In addition, an example implementation of a view-supporting agent is developed, and the hybrid agent architecture used to enable autonomy in this environment is described.

LITERATURE REVIEW

Implementing and Maintaining a Data Warehouse

A data warehouse is an extensible storage facility (a large DBMS) for non-volatile data gathered from operational and external sources that supports an organization’s decision-making processes (Gupta, 1998). A model of the DWDS environment is provided in Figure B.1. Data from multiple operational systems and external sources is stored in a data warehouse in accordance with the data warehouse model. The meta data, which documents the model, is also stored in the data warehouse and describes the structure of the data, the methods used for summarization, and the sources of the data in the operational environment.

Prior to being stored in the data warehouse, the data goes through a transformation or cleansing process in which the terms and attributes used in the operational systems and external sources are translated into consistent, self-explanatory business terms and attributes. The data stored in data warehouses is referred to as non-volatile because once stored in the data warehouse, the data is rarely changed. All data arriving from the operational systems goes through the same transformational process prior to being stored in the warehouse. This transformation process is typically automated by the data warehousing application software.

Data stored in a data warehouse is initially maintained at a very detailed level as shown in Figure B.2. A data warehouse system, however, generally receives more requests for aggregated data than detailed level data. Queries for aggregated data from multiple tables can require extensive processing resources and degrade the performance of the data warehouse. Frequently, specific views of data are saved as tables within the data warehouse to address this problem. Views are aggregations of data that are requested by users often enough that it is more efficient to permanently store the data in the aggregated format than it is to regenerate the aggregation from the detailed data each time. It is very common for data warehouses to have several layers of
Appendix B: Employing Agents in the Data …

Figure B.1. The DWDS Environment. Adapted from R. Sen (1997).

Key:
OLTP On-Line Transaction Processing
ROLAP Relational On-Line Analytical Processing
MD Multi-Dimensional
DBMS Database Management System
OLAP On-Line Analytical Processing
Figure B.2. The Flow of Data in a Data Warehouse. Adapted from W.H. Inmon (1995).
stored aggregations. As data becomes old and is less frequently used, it can be removed from the views and the detailed level and be archived to make room for more current data.

Data marts represent a functional or departmental subset of a data warehouse. Data marts are created to relieve the demand on the organization-wide data warehouse and to allow individual departments or functional areas to quickly access the data of most interest to them. These data marts typically reside in a different physical location than the data warehouse, but the warehouse remains the source of information and adds new data to the data marts as it becomes available. Data marts can take the form of multi-dimensional cubes or the more familiar relational DBMS format and contain both detailed-level data and aggregated data. Multi-dimensional cubes are multi-dimensional DBMS that contain numerical data and typically have a very structured format. The relational DBMS data marts, commonly called ROLAP DBMS contain both numeric and textual data and have a less structured format.

OLAP is the analysis of data modeled in multiple dimensions for the purpose of decision support, and represents a primary intersection point for the decision-maker and data originating in the data warehouse. OLAP typically takes place within the data marts, with the slicing and sectioning variety of OLAP taking place in the multi-dimensional cubes, and the more general purpose OLAP taking place in the ROLAP DBMS. Data mining is the process of recognizing patterns in the detailed level data and represents another intersection point for the decision-maker and data originating in the data warehouse. Data mining typically takes place in the data warehouse where the most detailed data is stored, but can also take place in the detailed data of the data marts.

**Potential Uses of Autonomous Agents**

As autonomous computer programs, software agents provide a level of abstraction for the user, hiding interface and interoperability complexity. Agents have frequently been employed to automate repetitive tasks, retrieve information from various sources, monitor information and resources of interest, and serve as an interface between heterogeneous applications. The benefits and previous uses of agents are well matched to the complex, information rich domain of data warehousing. The following sections provide a framework for employing agents within a data warehousing environment. The potential uses of agents are grouped according to the general data warehouse activities of loading, maintaining and accessing. A visual framework of agent uses within DWDS is provided in Figure B.3.

**Agent Applications in Loading the Data Warehouse.** Most commercial data warehousing applications have high-level, integrated components to load data from the operational systems into the data warehouse. A middleware component extracts the data from the operational systems and passes it to a data transformation component where it is cleansed. While operational sources of data are directly interfaced with the middleware, the external sources of data for the data warehouse provide an excellent opportunity for assistance from software agents. External sources of data include formal corporate reports such SEC filings and market surveys. Source gathering agents can be created to retrieve data from these external sources, regardless of location, and pass this information to the cleansing application. For example, an agent could be programmed to retrieve an SEC quarterly filing from a server in the accounting department, convert it into an appropriate file format, and transfer it to the data warehousing system for
Appendix B: Employing Agents in the Data …

Figure B.3. Visual Architecture of the Agent-Integrated DWDS

Key:
OLTP On-Line Transaction Processing
OLAP On-Line Analytical Processing
A1 Search Agents
A2 Source-Gathering
A3 View-Supporting Agents
A4 Resource-Monitoring Agents
A5 Archiving Agents
A6 End-User Agents
cleansing. As new external sources develop, additional instances of this agent could be created to ensure the timely collection of data for the warehouse. This source-gathering agent provides information retrieval and interoperability functionality and could utilize mobility in carrying out its tasks.

Internet search agents are one of the most commonly used types of agents and offer skills that are particularly useful in the loading process of data warehouses. These agents parse text looking for user-specified words or synonyms. These parsing skills could be very useful in locating user-requested data within the legacy operational systems. Data warehouse project managers are frequently requested to add more data to the warehouse. Locating the possible sources of the data in the operational systems can be time consuming, especially when these systems are older legacy systems that have grown haphazardly in content and structure over the years. Search agents can parse the files from these legacy systems and locate all of the possible sources for the requested data. These agents can be endowed with the ability to detect patterns in the data so that they can find the requested data even when different variations of the data terms are used (i.e. acctno, accountnum, custnumber).

**Agent Applications in Maintaining the Data Warehouse.** As discussed above, one way in which data warehouse managers can improve the performance or response time of a data warehouse or a data mart is by creating views. Most users will request summary data and will not need to review the detailed data. When multiple users request the same summary information, it becomes more efficient to store the summary information as a separate table or view rather than dynamically joining tables to retrieve the data each time it is requested. Intelligent agents could provide assistance in this area by monitoring user queries for similarities in the data requested and noting the frequency of requests. View-monitoring agents could monitor table accesses and the sources (users) initiating the accesses. These agents could inform the data warehousing manager of common data requests and the frequency with which these requests occur. Similarly, resource-monitoring agents could monitor overall system usage, and identify groups of users that utilize a significant percentage of system resources. This identification process could assist the data warehousing manager in planning for future data marts and in understanding the general pattern of usage across the company.

As shown in Figure B.2, when data first enters a data warehouse it exists at a very detailed level. It may then be summarized into views or even data marts. While data warehouses can keep data accessible at a detailed or summarized level for several years, larger data warehouses sometimes need to archive data. Agents can assist with the task of selecting data to be archived in a similar manner to how they can be used to select data for aggregation into views and for representation in a separate data mart. Archiving agents can be used to monitor older data for infrequent use and to recommend the data for archiving as storage in the data warehouse becomes a scarce resource.

**Agent Applications in Accessing the Data Warehouse.** As discussed in the previous two sections, agents can provide assistance in loading and maintaining the data warehouse. In these aspects of DWDS, the agents are assisting the data warehouse manager and his or her staff. The end user, i.e., the decision-maker, provides another avenue for agents to provide assistance. A data warehouse or data mart can quickly become an invaluable source of mission-critical data for decision-makers in many different functional areas. These end users can employ autonomous,
software agents on their desktops to monitor the warehouse and marts and to automate retrieval tasks. Agents can be used to watch for the updates of specific views, to notify users when unusual fluctuations in specific data elements occur, or to automatically integrate new warehouse data into the users’ spreadsheets.

Commercial developers of OLAP and other DWDS end-user tools have recognized the valuable assistance that agents can provide in making the DWDS environment more accessible to the decision maker and have integrated these autonomous computer programs into their applications. MicroStrategy’s Intelligent Agents and Alerts, Information Advantage’s Meta Agent, and ProdeaBeacon’s Proactive Agent are just few examples of how agents are being integrated into commercial DWDS products.
APPENDIX B – REFERENCES


VITA

TRACI J. HESS

EDUCATION

**Ph.D. Candidate** (Business Administration - Management Science & Information Technology), 1999.
Virginia Polytechnic Institute & State University


**Master of Accountancy** (Accounting Information Systems), 1997
Virginia Polytechnic Institute and State University

**Bachelor of Science** (Accounting), 1988
The University of Virginia

PAPERS UNDER REVIEW


CONFERENCE PRESENTATIONS


WORKSHOPS


TEACHING EXPERIENCE

Virginia Polytechnic Institute and State University

<table>
<thead>
<tr>
<th>Course</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personal Computers in Business</td>
<td>Fall, 1996</td>
</tr>
<tr>
<td>DSS Seminar - Design &amp; Implementation</td>
<td>Summer, 1996</td>
</tr>
<tr>
<td></td>
<td>Fall, 1998</td>
</tr>
<tr>
<td></td>
<td>Spring, 1999</td>
</tr>
</tbody>
</table>

WORK EXPERIENCE


Senior Vice President and Treasurer, Bank of Hampton Roads, Chesapeake, Virginia, May 1990-July 1994.

Senior Accountant, Ernst & Young, LLC, Norfolk, Virginia, May 1988-May 1990, May 1987-August 1987

Certification: Certified Public Accountant, Commonwealth of Virginia.

SPECIAL AWARDS AND HONORS

- Cunningham Doctoral Fellowship, university-wide fellowship awarded by Virginia Polytechnic Institute & State University, 1996-1999.
- Phi Kappa Phi, Honor Society, 1997.