A Real-Time Agent Architecture and Robust Task Scheduling

By

Zhao Lei

Supervised By :

Prof. Jimmy H.M. Lee

A Thesis Submitted in Parital Fulfilment

of the requirements for the degree of

Master of Philosophy

in

Computer Science & Engineering

August, 2002

A Real-Time Agent Architecture and Robust Task Scheduling

submitted by

Zhao Lei

for the degree of Master of Philosophy
at the Chinese University of Hong Kong

Abstract

The task at hand is the design and implementation of real-time agents that are situated in a changeful, unpredictable, and time-constrained environment. Based on Neisser's human cognition model, we propose an architecture for real-time agents. This architecture consists of three components, namely perception, cognition, and action, which can be realized as a set of concurrent administrator and worker processes. These processes communicate and synchronize with one another for real-time performance. The design and implementation of our architecture are highly modular and encapsulative, enabling users to plug in different components for different agent behavior. In order to verify the feasibility of our proposal, we construct a multi-agent version of a classical real-time arcade game "Space Invader" using our architecture.

We also study the issues and propose a multiple method approach to task scheduling in our real-time agent architecture. To be able to respond to requests in a timely manner while maintaining sufficient scheduling quality, the proposed hybrid method consists of both a greedy algorithm and an advanced algorithm. The greedy algorithm aims at catering for urgent events but sacrificing quality, while the advanced algorithm opts for optimal (or sub-optimal) solution sched-

ules. By giving a scheduling model of the architecture, we conduct an analysis of the bounds of the competitive ratio of the hybrid method. The validity of the analysis is verified empirically. Further experimental results confirm the robustness, efficiency, and quality of the proposed approach.

实时代理体系结构及任务排序

赵雷

论文摘要

如何设计并实现能够在复杂的、不可预测的、并有实时限制的环境中工作的实时代理已经是一个迫在眉睫的问题。以人类认知模型为基础，我们设计了一个实时代理体系结构。这个体系结构包含了感知、认知、动作三个子系统，每个子系统都由一组并行的管理者和工人进程组成。这些进程使用消息传递来通信并保持同步以满足实时要求。这个体系的设计和实现是非常模块化和封装化的，用户可以通过更动不同的模块来实现实时代理不同的行为。为了验证此体系的可行性，我们用其实现了一个多代理系统的实时射击游戏。

在这篇论文里，我们也讨论并设计了一个应用在我们的实时代理体系里面的多方法任务排序功能。为了能够在保持系统的效率同时还可以及时的响应各个要求，我们设计了一个同时包含了贪婪算法与先进算法的混合方法。贪婪算法可以在牺牲系统效率的情况下保证系统的响应性，而先进算法则是以响应性为代价去获得最优的（或者是次优的）排序结果。我们建立了这个方法的模型，并以此为基础分析了这个混合方法的上限。测试的结果证明了我们的分析的正确性，也证实了我们体系的鲁棒性及高效性。

Acknowledgments

I would like to thank my parents for providing me with their endless support and love in the past 22 years. I love them forever.

I would like to thank Prof. Jimmy H.M. Lee, my supervisor, for his guidance and patience. My research could not have been done reasonably without his insightful advice. For the past two years, he gave me encouragement, support, and guidance on my research.

My gratitude goes to Prof. Leung Ho Fung and Prof. Ng Kam Wing who marked my term paper and gave me valuable suggestions.

Finally, I thank to my fellow colleagues. They helped me in solving a lot of problems of my research and gave me a happy and wonderful university life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advent of distributed computing and distributed artificial intelligence (DAI), the efficient design and implement of distributed software has assumed an important role in computer science research. The object-oriented design method has been successful for non-distributed applications, but it also causes problems when it is applied to specific distributed applications [37]. It is difficult to apply the object-oriented model in open environments or heterogeneous systems effectively.

The rapid development of agent-based technologies started from the beginning of the 1990s [66]. This new discipline has emerged from many research areas, such as symbolic artificial intelligence, control theory, and distributed artificial intelligence. Software agents, which can be defined as autonomous intelligent software entities, is a new software design method with a great level of decentralization, an important characteristic for distributed environment. The agent-oriented method also has the following additional advantages for distributed environment. First, the autonomy of agents, which means the ability to make decisions based on an internal representation of the world, allows more efficient communication and management of distributed resources. Second, agents are flexible and responsive, which provide great benefit for real-time applications.

Third, the deliberation of agents allows them solving problems efficiently by making decisions in a goal directed manner.

The distributed and real-time nature of real world applications urge us to design and implement real-time agents, such as those used in military training systems, flexible transport systems, and industrial control systems. Such systems are situated in a changeful, unpredictable, and time-constrained environment. We define *real-time agent* as a proactive software entity that acts autonomously under time-constrained conditions by means of real-time AI techniques. The requirement of real-time AI provides the agent with the ability of making quality-time tradeoff, either discretely or continuously.

There have been other approaches towards real-time agents, such as Brooks's Subsumption architecture [6], Muller's InterRAP model [49], and Bonasso's 3T architecture [3], etc. Most of them are layered architecture and have been used in robot control applications. Their advantages and disadvantages are discussed in the following chapters.

In this dissertation, we develop a real-time agent architecture from Ulric Neisser's human cognition model [51]. In our architecture, a real-time agent is composed of a set of concurrent components. These components communicate and synchronize with one another for real-time performance. Our architecture has two distinct features: pluggability and dedicated task scheduling. First, components in our architecture are highly encapsulated with well-defined interfaces, so that components of different characteristics, functionalities, and implementations can be plugged in to form real-time agents for specific real-time applications. Second, our architecture is *meta* in the sense that we can plug in some existing agent architecture X, such as the subsumption architecture [6], to make X more real-time respondent while maintaining the characteristic behavior of X, *especially in overload situation*. This is achieved by the task scheduling component, which is designed to deal with tasks and requests arriving at unexpected time

points and being of various urgency and importance.

Our on-line task scheduling mechanism, relying on the cooperation of a greedy and an advanced scheduling algorithms, take the multiple method approach for quality-time tradeoff. The greedy algorithm aims at catering for urgent events but sacrificing quality, while the advanced algorithm can provide optimal (or sub-optimal) solutions. To demonstrate the effectiveness and efficiency of our proposal, we construct a multi-agent version of a classical real-time arcade game "Space Invader" using our architecture. In addition, we also test the competitive ratio, a measure of goodness of on-line scheduling algorithms, of our implementation against results from idealized and simplified analysis. Results confirm that our task scheduling algorithm is both efficient and of good solution quality.

This thesis is structured as follows.

Chapter 2 introduces the backgrounds of our research. We give a brief overview of agent theory, and survey real-time AI technologies. Related work is also discussed and evaluated with respect to its scope and limitations. Disciplines of on-line scheduling are presented at the end of this chapter.

Chapter 3 describes a real-time agent architecture. Based on the human cognition model, we give an agent model which is composed of three subsystems. We introduce Gentleman's [31] message passing method, a communication mechanism which satisfies the architecture's requirements, and give the agent architecture based on this special inter-process communication mechanism. We also show a real-time arcade game as a demonstration of our agent.

Chapter 4 illustrates the on-line task scheduling mechanism used in our agent. We describe the scheduling mechanism in details, and give a scheduling model of this problem. Some rules and special cases in scheduling are also given. At the end of this chapter, we introduce various scheduling algorithms which can be used in this mechanism.

Chapter 5 evaluates the task scheduling mechanism described in the previous chapter. We first introduce the theoretical analysis of this mechanism, and describe a simulation system implemented for testing. We give a series of experimental results and indicates how these results verify our analysis.

Chapter 6 summarizes the results of this thesis. We also identify topics for future research.

Chapter 2

Background

In this chapter, we introduce some background of our research. Section 2.1 introduces principles of agents theory and describes various kinds of agents. Section 2.2 explains current research of real-time AI technologies and compares two main types of approximate algorithms: Anytime Algorithms and Multiple Methods. Section 2.3 introduces and evaluates approaches towards real-time agents. Online scheduling problem is described in Section 2.4.

2.1 Agents

We apply agents in real-world applications not only because these applications are complicated, but also because the problems we met are physically distributed. For example, an industrial control system that is used on assembly lines is naturally distributed. Such an assembly line requires the intervention of a large number of specialists, who have only a local view of all the problems in the system. To create a know-all system is almost impractical. A more natural way is to create a set of subsystems. Every subsystem can work independently in its special area. And all these subsystems can communicate with each other for cooperation. We call such a subsystem an *Agent*, and the whole system a

Multi-Agent System (MAS).

While it is true that a basic definition for an *agent* is hard to give [66], we can use a list of agent properties to illustrate agents. Wooldridge and Jennings [66] define agent as a hardware or software-based computer system that enjoys the following properties:

- Autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state [10]. Some researchers also define autonomy in software agents as a process or a set of processes running as separate threads [35].
- Social Ability: agents interact with other agents (and possibly humans) via some kind of *agent communication language* [30].
- Reactivity: agents perceive their environment, which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined, and respond in a timely fashion to changes that occur in it.
- Pro-activeness: agents do not simply act in response to their environment, but they are able to exhibit goal-directed behavior by taking the initiative.

These properties provide a possible description of an agent. Huhns and Singh [37] also lists the intrinsic properties of an agent and how an agent should react and cooperate with the environment and other agents in Table 2.1 and Table 2.2 respectively. These tables define a finite set of characteristics that a generic agent possesses.

Table 2.1: Intrinsic Agent Characteristics

Property	**Range of Values**
Lifespan	Transient to Long-lived
Level of cognition	Reactive to Deliberative
Construction	Declarative to Procedural
Mobility	Stationary to Itinerant
Adaptability	Fixed to Teachable to Autodidactic
Modelling	Of environment, themselves, or other agents

Table 2.2: Extrinsic Agent Characteristics

Property	**Range of Values**
Locality	Local to Remote
Social autonomy	Independent to controlled
Sociability	Autistic, Aware, Team Player
Friendliness	Cooperative to Competitive to Antagonistic
Interactions	Style/Quality/Nature with agents/world/both Semantic level: declarative or procedural communications Logistics: direct or via facilitators

2.1.1 Deliberative Agents

Some agent models are based on Simon and Newell's physical symbol system hypothesis [52]. They assume that agents maintain an internal representation of the world, and can be modified by symbolic reasoning. These agents are called deliberative agents. Some interesting research approaches have discussed the modelling of agents based on beliefs, desires, and intensions. Architectures following this paradigm are known as *Belief, Desire, Intension* architectures (BDI).

Since Bratman first introduced BDI architecture in 1987 [5], it had become an interesting research area [54, 55]. The BDI architecture describes the internal processing state of an agent as a set of *mental categories*, and defines a control architecture to choose actions rationally. The mental categories are *belief, desire*, and *intentions* (or goals and plans).

Belief describes its expectations about the current state of the world and the effects achieved from different actions. Beliefs are modelled by possible worlds semantics, where a set of possible worlds is associated with each situation, denoting the worlds that the agent believes to be possible. More details of belief can be found in other references [33, 65].

Desire specifies future world states or actions. An agent is allowed to have inconsistent desires, but it does not need to believe these desires are achievable.

Intension describes how an agent to select a certain goal to commit to. Intensions determine the agent's actions, and feedback into the agent's future reasoning.

2.1.2 Reactive Agents

Researchers such as Brooks [6, 7], Chapman and Agre [1], Kaelbling [38], and Maes [46] develop new agent architectures that are *behavior-based*, *situated*, or *reactive*. Based on limited amount of information and simple situation-action rules, these agents make their decisions during runtime.

Some researchers, such as Brooks, deny the need of any symbolic representation of the world. Reactive agents make decisions directly based on sensory input. The design of reactive agent architectures is based on Simon's hypothesis in which the complexity of the behavior of an agent is a reflection of the complexity of environment. Reactive agents focus on achieving robust behavior instead of correct or optimal behavior.

2.1.3 Interacting Agents

Distributed Artificial Intelligence (DAI) [4, 27] deals with coordination and cooperation among distributed intelligent agents. Main topics in agent interaction and related works are introduced in the following:

- *Communication.* Communication among agents is important in interaction among agents. Some researchers have established standard agent communication languages. The Knowledge Query and Manipulation Language (KQML) [21] model (which is based on *Speech Act theory* [2, 58]) consists of three layers. The *communication layer* describes the lower-level communication parameters. The *message layer* forms the core of the language. It identifies the underlying protocol and supplies a performative which is attached to the message content. The *content layer* contains the actual contents of the message in an agreed-upon language, for example, KIF [29].

- *Game theory and agent interaction.* Rosenschein and Zlotkin design a game theoretic analysis of interaction among rational agents [56, 68, 57].

- *Distributed Problem-Solving (DPS).* DPS deals with the performance in a given task by using a set of distributed problem solvers. It focuses on the mechanisms for task decomposition, and discusses the protocols for the allocation of tasks to these problem solvers, for example, the Contract Net Protocol [16] and its various extensions [24]. Based on DPS, many research approaches has been done towards to the exploration of conflict resolution and cooperation based on negotiation [13, 40, 41].

- *Multi-Agent Planning.* Multi-agent planning is related to distributed problem solving closely. This work focuses on coordination mechanisms among agents, for example, relationships among plans of multiple agents [47].

2.1.4 Hybrid Architectures

Different kind of agents have different shortcomings. Purely reactive systems have a limited scope that it is hard to implement goal-directed behaviors on them. Most deliberative systems are based on general purpose reasoning mechanisms which are not tractable, and less reactive. Layered architecture can help us to overcome these shortcomings. It has become very popular over the past few years [38, 7, 8, 19, 23, 45, 14, 3, 59, 50]. Layered architecture is a powerful tool for structuring, controlling, and designing systems with some desired properties like reactivity, deliberation, cooperation, and adaptability. The key idea is to structure the functionalities of an agent into two or more hierarchically organized layers that interact with each other to achieve coherent behaviors.

2.2 Real-time Artificial Intelligence

Traditionally, artificial intelligence (AI) techniques have not been utilized in real-time environments due to their highly unpredictable performance. This is a result of the types of difficult problems that AI research has focused on which often involve searching as a component of the solution method. Complex algorithms that incorporate searching are unpredictable mainly because it is never analytically clear how much of the search space must be seen in order to compute an answer [26].

A major step forward in real-time AI research begins with the concept of approximate processing and approximate algorithms. To date, real-time AI research has been interested in two main types of approximate algorithms: *Anytime Algorithms* and *Multiple (Approximate) Methods* [15].

An anytime algorithm is an iterative refinement algorithm where a "default" answer is first generated and then refined through multiple iterations. It is also

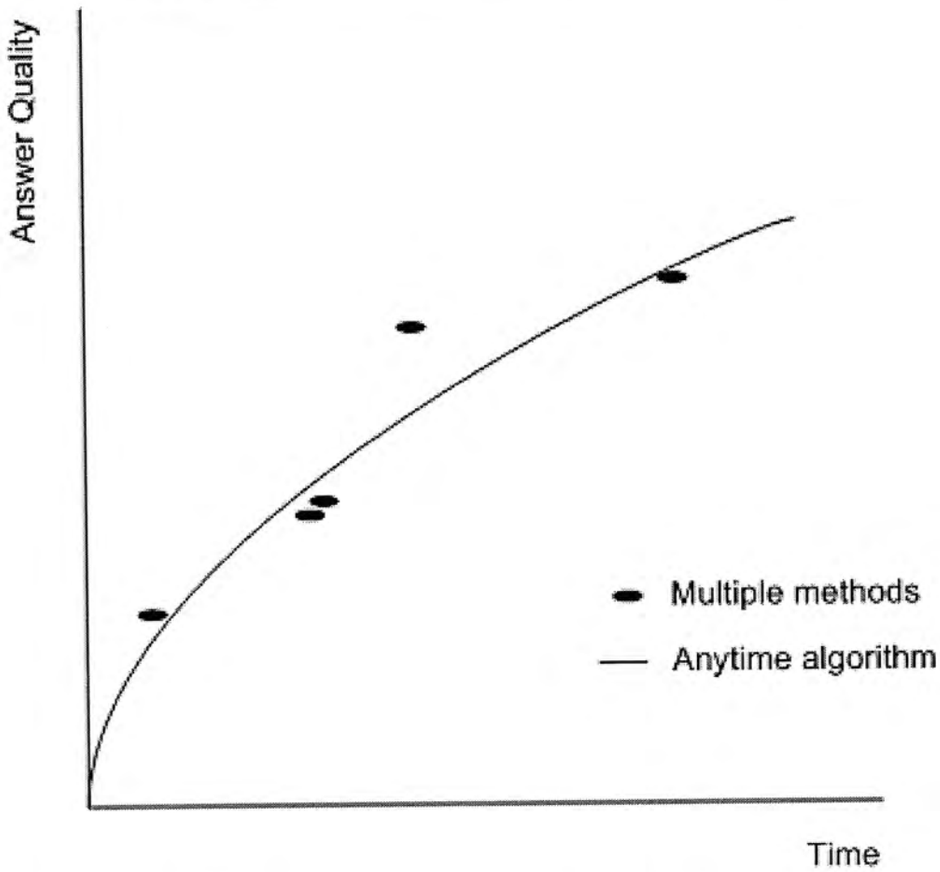

Figure 2.1: Anytime Algorithms and Multiple Methods

true that the quality of the solution increases proportional to the amount of time the algorithm executes. In addition, anytime algorithms always produce a result regardless of when they are interrupted [26, 25, 15].

The multiple method approach does not rely upon continuous processing to solve a problem. A set of available methods is used to solve a task. Each method has different characteristics that make it more or less appropriate given the current conditions. Every method solves the same problem, but different in the amount of time it needs to find the result and the quality of the result. There is a quality-time tradeoff between methods where a shorter execution time is achieved through reducing the quality of the answer [25, 26].

Figure 2.1 illustrates the major difference between an anytime algorithm and a multiple method algorithm. Both approaches provide tremendous flexibility.

The advantage of using anytime algorithm is that they can fit into any available time slots whereas the multiple method approach allows for multiple, yet discrete, execution times. Garvey and Lesser [26] give two potential advantages of the multiple method approach over an anytime algorithm approach. First, the multiple method does not rely on the existence of iterative refinement algorithms that produce incrementally improving solutions as the time increases. Some problems may not have a solution that can be implemented by an anytime algorithm. A second advantage to the multiple methods approach is that the methods may be completely different approaches to solving the problem. These approaches can have different characteristics depending on particular environmental conditions. An additional challenge to the anytime algorithm approach is developing an algorithm whose performance is independent of environmental variables.

2.3 Real-Time Agents

We define *real-time agent* as a proactive software entity that acts autonomously under time-constrained conditions by means of real-time AI techniques. The requirement of real-time AI provides the agent with the ability of making quality-time tradeoff, either discretely or continuously. Besides sharing all common characteristics of intelligent agents, real-time agents also have specific features for survival in real-time environments, listed as follows:

- *Real-Time AI*: Real-time agents must be able to consider time's effect in the system. From knowledge or experience, agents must know how to control resources to meet various hard and soft timing-constraints and perform quality-time tradeoff. This calls for real-time AI techniques, which are approximate processing and algorithms of two main types: anytime algorithm and multiple (approximate) methods [15].

- *Perception*: Because of the data distribution of environments, real-time agents must be able to collect data from environments as correctly and completely as possible. Any data may be useful. The extent that this can be achieved is greatly influenced by the agents' sensory capability and the buffer size we set.

- *Selectivity*: Since agents try to perceive as much data as they can, they sometimes cannot process all data in time (data glut). Agents must be able to select useful data (or data which the agents think useful) from received data. Unprocessed data can remain in buffer, and can be flushed by new arriving data. Depending on the application and environment, different agents can have different control of selectivity.

- *Reaction*: Agents must be able to react to different events in the environment. The more urgent a situation is, the more quickly the agent should respond to it, even if the event is unexpected.

There has been a scattering of work towards real-time agents over the years. Most of them are layered system and have been used in robot control applications [7].

2.3.1 The Subsumption Architecture

One of the earliest real-time agent design is the subsumption architecture given by Brooks [6]. The basic idea of the subsumption architecture can be characterized as follows:

1. Each module is connected in parallel between input and output.

2. Modules form layers in which higher layer can subsume lower layer functions.

3. Lower layers control basic behaviors and higher layers add more sophisticated behavior.

4. The total behavior of the system can be changed by adding a new layer at the top without changing existing layers.

When a new layer is added at the top, it is sometimes necessary to change the behavior of existing layers. An implementation of the subsumption architecture is depicted in Figure 2.2, in which higher layers send bias signals to lower layers to deceive their inputs.

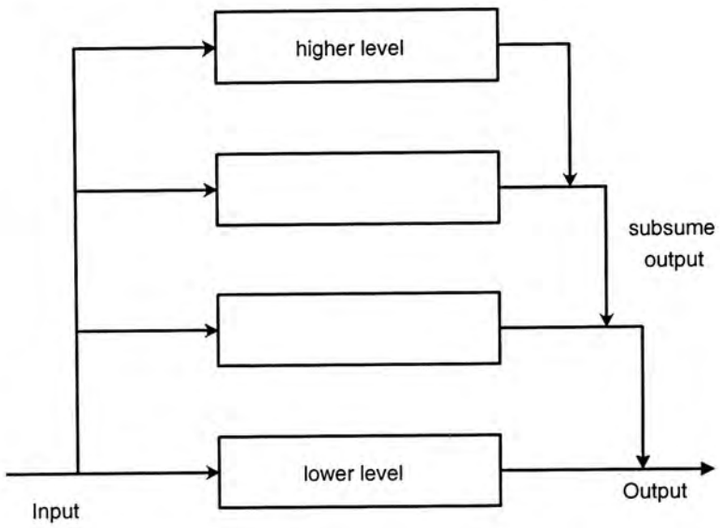

Figure 2.2: Implementation of subsumption architecture

A shortcoming of the subsumption architecture is the inflexibility. If the application changes, the architecture must be reconstructed from scratch. Another shortcoming is that the circuit is fixed and cannot be easily reconfigured to adapt new situation.

Nakashima's dynamic subsumption architecture [50] extends the original subsumption architecture to meet these problems. Figure 2.3 illustrates the dynamic subsumption architecture.

The dynamic subsumption architecture consists of the following elements.

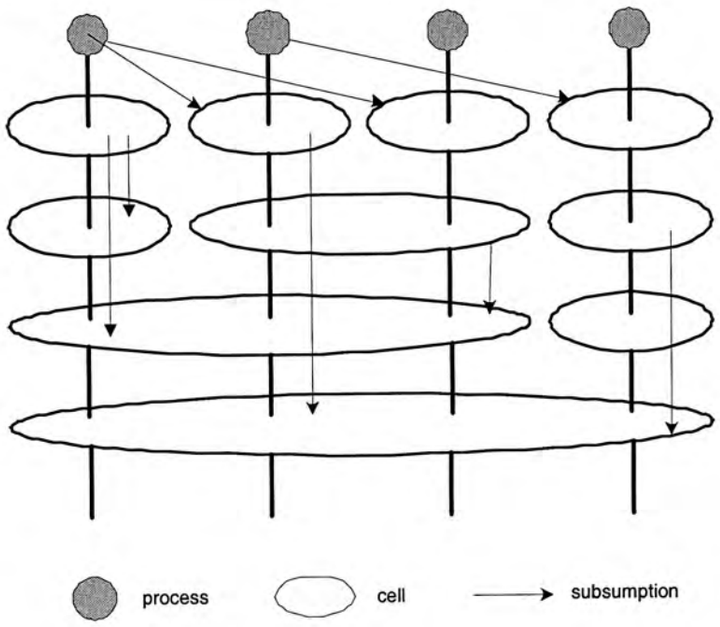

Figure 2.3: Dynamic subsumption architecture

- Processes: which execute programs. Each process combines some cells to perform some coherent behavior.
- Cells: which store fragment of programs. Various cells represent various modalities or behaviors of the agent. They may differ from one to another. These cells are also divided into different functional layers. Cells farther from the processes (lower in figure) are used to implement lower-level functions, and cells closer to the processes are used for high-level functions.
- Subsumption: which connects cells among layers. These combinations can be dynamically changed during runtime. Through choosing different combination of cells, a process can realize various behaviors.

2.3.2 The InterRAP Architecture

In the InterRAP model [49], an agent is composed of three interacting control and knowledge layers:

- *Behavior-based layer* controls the reactivity and procedural knowledge for routine tasks.
- *Local planning layer* provides the facilities for means-ends reasoning for the achievement of local tasks and produces goal-directed behavior.
- *Cooperative planning layer* enables agents to reason about other agents and supports coordinated action with other agents.

These layers and the control architecture defined for them combine reactive and deliberative reasoning, and incorporate the ability to interact with other agents. InterRAP architecture is a BDI architecture [54]: the informational, motivational, and deliberative state of an agent [55] is described by means of beliefs, desires, and intentions.

The components of the mental state of an InterRAP agent are layered. *Beliefs* are split into three models: a world model, a mental model, and a social model. The world model contains object level beliefs about the environment. The mental model controls meta-level beliefs the agent has about itself. The social model holds beliefs about other agents.

The InterRAP architecture extends the planner-reactor architecture (Bresina and Drummond [17]) by adding a cooperation layer. Such a combination is feasible and has been realized in the FORKS application both as a simulation and as a physical system in real robots [49].

2.3.3 The 3T Architecture

Bonasso's 3T architecture [3] is an intelligent robot control architecture. This architecture separates the general robot intelligence problem into three interacting layers or tiers (that is why it is called 3T):

- A dynamic reprogrammable set of reactive skills that is coordinated by a skill manager [67].

- A sequencing capability that can activate and deactivate sets of skills to create networks, which change the state of the world and accomplish specific tasks. For this 3T architecture use the Reactive Action Packages (RAPs) system [22].

- A deliberative Planning capability that reasons in depth about goals, resources and timing constraints. This 3T architecture uses a system known as Adversarial Planner (AP) [18].

The 3T architecture uses several levels of abstraction and description languages. A robot can be realized with just the first or the first and second layers. Skills can be robot specific, the RAPs and APs are generalized among different manipulators and platforms.

An advantage of the 3T architecture is that the programmer does not need to explicitly process the data coming to and from a skill, since the skill manager framework can coordinate it. For example, there are skills for avoiding obstacle and tracking moving objects. By simply feeding the output of the tracking moving objects skill to the input of the avoiding obstacle skill, the robot could follow people while still avoiding obstacles. Another advantage is that the 3T architecture allows for modifications without having to reinitialize the robot controllers.

The 3T architecture is very similar to ATLANTIS [28], which embodies the "sequencer in control" approach to coherent behavior. 3T also shares many aspects of Cypress [63].

A variety of useful software tools can be used to help implement this architecture on multiple real robots. This architecture has been implemented on some robot systems using a variety of processors, operating systems, effectors

and sensor suites [48, 36, 64].

2.4 On-line Scheduling in Real-Time Agents

As stated by Tanenbaum [60], real-time scheduling algorithms can be characterized by the following parameters:

1. Hard real-time versus Soft real-time.
2. Preemptive versus Non-preemptive scheduling.
3. Dynamic versus Static.
4. Centralized versus Decentralized.

These algorithms attempt to schedule a set of tasks for either a single processor or multiple processors. Every task will have a deadline before which it must be executed.

Hard real-time systems require all tasks to be finished before their deadlines. Soft real-time systems are more lax. Soft real-time is generally characterized by "as close as possible" algorithm.

Preemptive and non-preemptive algorithms differ in their handling of task execution. A preemptive scheduling algorithm has the ability of suspending a running task so that a more important task can be performed, after which the execution of the suspended task is resumed. The importance of a task can be defined by a higher priority, a shorter execution time, or a better profit. Non-preemptive scheduling algorithms do not have this suspension-resume ability. Once a task is started, the system must finish it before perform other tasks.

Dynamic and static algorithms are different upon when they make decisions about scheduling. Dynamic algorithms make the decisions "on the fly" during execution. Static algorithms make all scheduling decisions before runtime.

A centralized system uses a single machine to collect information and to perform decision-making. In a decentralized system, more than one processor are available, decisions are made at the processor level.

In the real-time agent architecture we describe in this thesis, tasks are arriving over time. In *offline* problems, an algorithm is allowed to know the entire list of inputs and all the details of tasks in order to compute the optimal solution without time constraint. But in our architecture, at each time, we only know tasks which have arrived. All the futures are unknown. The algorithm we used must be able to find solutions of the current state on time. Such a problem is called *on-line problem.*

A common on-line scheduling algorithm is *Earliest Deadline First* (EDF) scheduling. An EDF algorithm maintains a list of waiting tasks to be executed which is always sorted by deadline with the first having the earliest deadline. When a new task enters the system, it is inserted into the list of waiting tasks. When the system resource is free, the first task in the list is removed and executed.

Chapter 3

A Real-Time Agent Architecture

This chapter introduces our real-time agent architecture. Using a human cognition model, we first explain the three subsystems in our agent architecture in Section 3.1. Section 3.2 introduces message passing method, an inter-process communication mechanism, and defines two kind of useful processes: administrators and workers. Based on message passing, we design our agent as a group of concurrently and synchronously running processes. The details of these processes are given in Section 3.3. The implementation of an agent based real-time arcade game is given in Section 3.4.

3.1 Human Cognition Model

Since human is the best example of real-time agents, we first introduce the principles of human cognition. Neisser [51] views human cognition as a perpetual process, which keeps working as long as we are awake. Figure 3.1 illustrates different parts and their relations in human cognition. In this model, human acquires samples by exploring outer environment (Exploration). These samples bring useful information of the world (Object available information). By modifying the information, human makes decisions and plans (Schema), which guide us

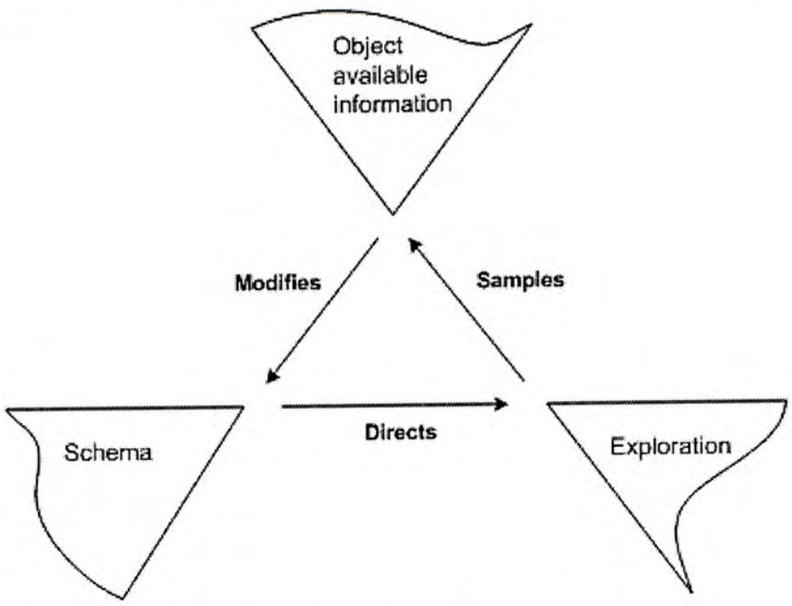

Figure 3.1: The Perpetual Cycle

to explore the new world and obtain further information. These three parts work concurrently, and function the same from neonatal children to world leaders.

In the wake of Neisser's model, we develop a real-time agent architecture which is composed of three subsystems: perception, cognition, and action. These three subsystems work concurrently and synchronously to acquire from and respond to the environment via real-time AI reasoning. These subsystems work autonomously and individualistically. None of them have the superiority to control the other two subsystems.

These subsystems are connected with predefined communication protocols. Keeping its protocol unchanged, a subsystem can be arbitrarily rewritten without affecting other two subsystems. The flexibility of this architecture is much higher than other real-time architectures we introduced in Chapter 2. We also design an on-line task scheduling mechanism to improve the efficiency. In other architectures, tasks are not specially scheduled.

3.1.1 Perception

Similar to the object-available-information part in Neisser's model, the *perception* subsystem observes the environment and collects all possible information. The scope of this information is decided by the techniques of observation.

In a real-time environment, a serious problem is data glut—the environment feeds more data than an agent can process [43]. The perception subsystem is thus responsible for information selection/filtration in addition to preprocessing and summarizing *raw signals* into semantically meaningful *events*, which describe the states of the environment and are for subsequent consumption by the cognition subsystem.

3.1.2 Cognition

The *cognition* subsystem is the kernel of a real-time agent. It makes decisions or plans from the events collected by the perception system. These decisions and plans are dispatched in the form of *tasks*, which consist of a recipe of actions and their corresponding sequencing constraints. A task is sent to the action subsystem once generated.

Various cognitive mechanisms can be used in the cognition subsystem. If we are more interested in reactive behavior, we can use the subsumption architecture [6], the dynamic subsumption architecture [50], or even simply a set of reaction rules for mapping events to tasks directly and efficiently; if intelligence is more important, we can use a world model with a set of goal directed rules (or logical formulae) [62].

3.1.3 Action

As the exploration part in Neisser's model, the *action* subsystem dispatches and performs tasks to explore and react to environment. The knowledge of how to perform these tasks is owned by the action subsystem. Neither the perception nor the cognition subsystem need to know this knowledge. The cognition subsystem needs only to generate tasks with digested information which can be understood by the action subsystem.

The action subsystem also stores and manages tasks, and chooses the most important and urgent task to perform first. An efficient on-line scheduling algorithm is thus central in the functioning of the action component.

Some simple rules can be used in the action subsystem and the perception subsystem to guide them to process external information or perform tasks. For this we can use the Reactive Action Packages (RAPs) system [22].

While these subsystems have individual responsibilities and goals, they must cooperate to act as a collective whole. A good inter-process communication (IPC) mechanism is needed. We also note that such a mechanism can also be used for effective synchronization purposes. The following characteristics are desirable for a good communication mechanism:

- *Simple*: a complicated mechanism may increase the complexity of the agent architecture, making the agents harder to understand and construct.
- *Efficient*: the volume of data exchanges among these subsystems is high in practice, demanding extreme efficiency especially in a real-time environment.
- *Autonomous*: the communication must be performed without central monitoring or supervision.

- *Robust*: message transmission should incur little errors.

In the following, we study a particular inter-process communication mechanism and the operating system in which this IPC mechanism is embedded in, before giving a process structure design of an implementation of our real-time agent architecture.

3.2 Real-Time Message Passing Primitives and Process Structuring

Message passing is a method of synchronizing and communicating among sequential processes. We choose message passing in our agent architecture because of not only the obvious interpretation for distributed systems but also its other advantages. The semantics for message passing is easy to deal with and easy to get right. The structure of processes, via the viewpoint of server (or an object), is simple and natural. It is also easy to understand the abilities to control processes, queue work requests, do load balancing, or other decentralized control actions based on message passing.

A set of message passing primitives are first designed by Gentleman [31] with special blocking semantics for efficient inter-process communication and process synchronization. Based on these primitives, different processes, each class with different functionalities, can be defined, enabling the design and implementation of deadlock-free and efficient real-time systems.

QNX [34] is an operating system designed by QNX Software Systems. As a real-time operating system, QNX is ideal for embedded realtime applications. It provides almost all essential ingredients of an embedded realtime system.

QNX is a multi-process operating system. It consists of a small group of

cooperating processes. QNX is the first commercial operating system of its kind to make use of message passing as the fundamental means of IPC. In QNX, a message is a parcel of bytes passed from one process to another. The operating system attaches no special meaning to the content of a message – the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

In the following section, we first introduce the details of message passing primitives and use an example to explain them. Based on the message passing mechanism, we define two kinds of useful processes: administrators and workers, which will be used to construct our agent architecture.

3.2.1 Message Passing as IPC

Three primitives of message passing are defined in QNX:

- *Send()*: for sending messages from a sender process to other processes. The sender process must specify the process ID of the process that is to receive the message. A process ID is the identifier by which the process is known to the operating system and other processes.
- *Receive()*: for receiving messages from other processes. The receiver process do not need to specify the sender process, it receives any messages send to it.
- *Reply()*: for replying to processes that have sent messages. After received a message, the receiver process knew the process ID of the sender process. Using this process ID, reply message can be sent back.

In a collaborating relationship, agents cannot work away without synchronizing with partners' progress. Communication is a means for informing others of

work progress, but a properly designed protocol can be used to effect synchronization behavior. In many occasions, a process must suspend its execution to wait for the results/response of a partner process. We say that that the waiting process is *blocked*. Semantics of blocking in a communication protocol must be carefully designed so that good programming style can be defined to avoid deadlock behavior. A process will be blocked in one of the following three conditions:

- *Send-blocked*: the process has issued a *Send()* request, but the message sent has not been received by the recipient process yet.
- *Reply-blocked*: the process has issued a *Send()* request and the message has been received by the recipient process, but the recipient process has not replied yet.
- *Receive-blocked*: process has issued a *Receive()* request, but no message is received yet.

Figure 3.2 shows the states involved in a typical send-receive-reply transaction. More details can be found in QNX manual [44].

Suppose process A sends a message to process B, they would undergo the following steps. This example is come from QNX manual [44].

1. Process A sends a message to process B by issuing a *Send()* request to the kernel. At same time, process A becomes *Send-blocked*, and must be blocked until B finishes processing the message.
2. Process B issues a *Receive()* request to the kernel.
   (a) If there has been a waiting message from process A, then process B receives the message without block. Process A changes its state into *Reply-blocked*.

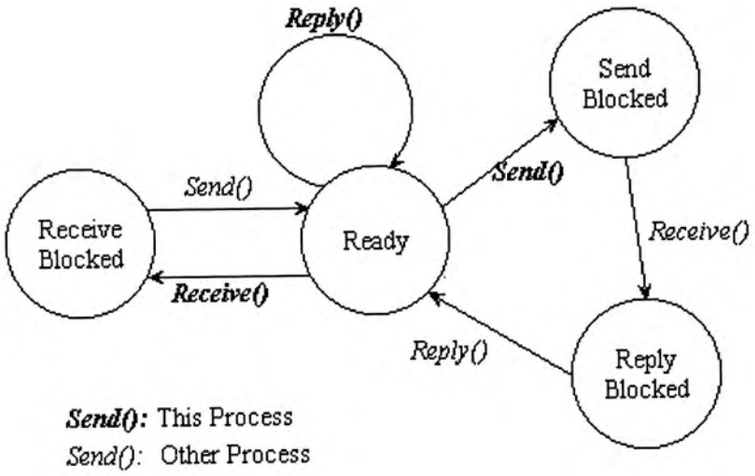

Figure 3.2: States involved in send-receive-reply transaction

(b) If there are no waiting messages from process A, then process B changes its state into *Receive-blocked*, and must wait until a message from A arrives, in which case process A becomes *Reply-blocked* immediately without being *Receive-blocked*.

3. Process B completes processing the received message from A and issues a *Reply()* to A. The *Reply()* primitive never blocks a process, so that B can move on to perform other tasks. After receiving the reply message from B, process A is unblocked. Both process A and process B are ready now.

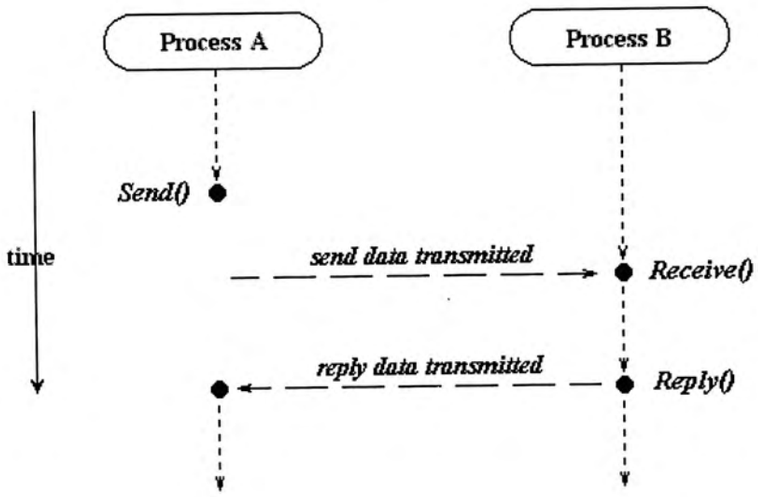

Figure 3.3: Process A sends a message to process B

3.2.2 Administrator and Worker Processes

The message passing example we described above illustrates the most common use of message passing: in which a server process is normally *receive-blocked* for a request from a client in order to perform some task. This is called *send-driven messaging*: the client process initiates the action by sending a message, and the action is finished by the server replying this message.

We use another messaging in this thesis: *reply-driven messaging*, in which

the action is initiated with *reply()*. Under this method, a "worker" process sends a message to the server indicating that it is available for work. The server does not reply immediately, but "remember" that the worker is ready for work. At some future time, the server may decide to initiate some action by replying to the available worker process. the worker process will do the work, and finish the action by sending a message containing the results to the server.

Reply-driven messaging enables us to define two kinds of useful processes: *administrators* and *workers*. An administrator process owns one or more worker processes. Administrator stores a set of jobs and workers perform them. Once a worker finishes a job, it issues *Send()* to its administrator to report the result of last job and require for a new job. Administrator receives this request for work and reply to the worker with a new job assignment. Administrators do only two thing repeatedly: *Receive()* and *Reply()*. Thus administrators are never blocked, since they never issue *Send()* messages, allowing administrators to handle to various events and requests instantly. This is in line with the behavior of top management officials in a structured organization: a manager must be free of tedious routine work, and allowed time to make important decision and making job allocations to her inferiors. On the other hand, low-level workers can only be either performing job duties or wait for new assignments.

3.3 Agent Architecture

We propose an implementation of our architecture on the QNX platform. In our definition, an agent is composed of a set of workers and administrators. They work concurrently and synchronously, communicating with each other and cooperating to react to the environment. Figure 3.4 reveals also the detailed implementation of the architecture. A real-time agent consists of the following components: the sensor administrator, sensor workers, cognition workers, the

task administrator, the task scheduler worker, and executor workers. We describe each component in the rest of this section.

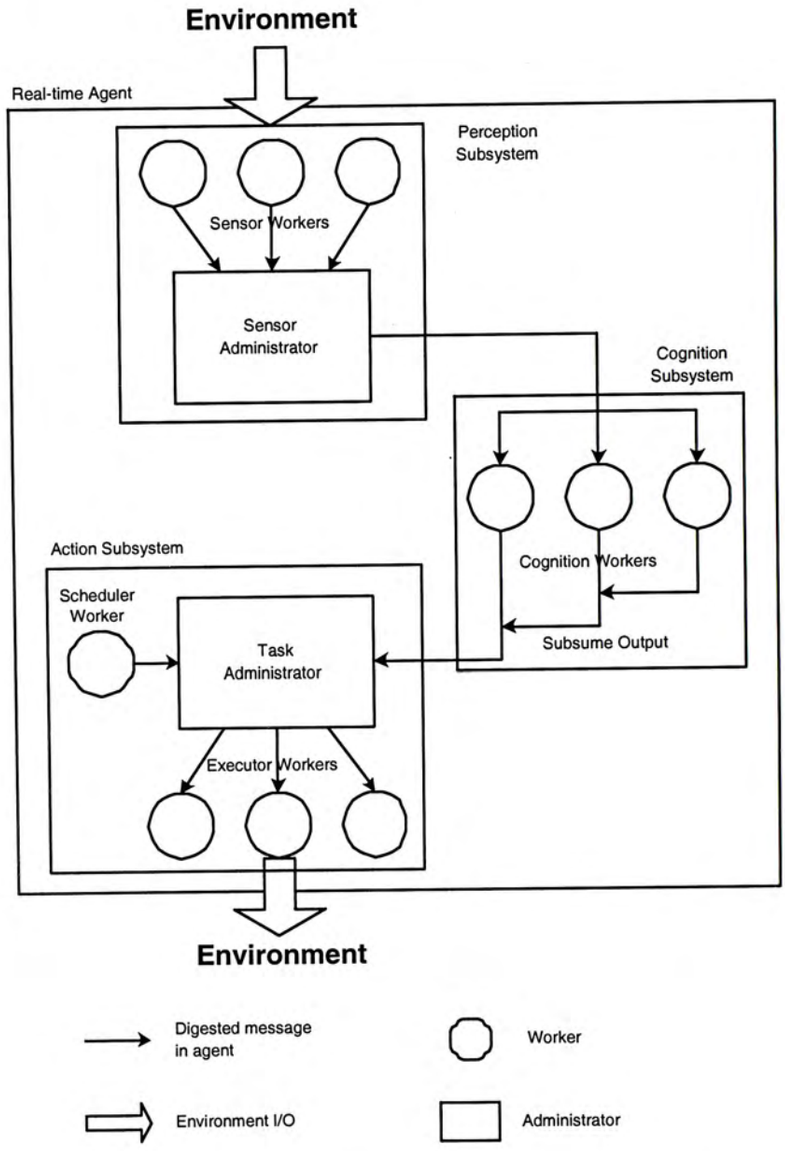

Figure 3.4: Real-Time Agent Architecture

3.3.1 Sensor Workers and the Sensor Administrator

The sensor administrator and sensor workers constitute the perception subsystem. The sensor administrator receives messages from other agents and environment signals detected via the sensor workers. The administrator also prepro-

cesses the input messages and signals, and translate them to events which can be utilized by the cognition subsystem. The administrator contains an event queue for storing received events, just in case the cognition subsystem is busy. When the cognition subsystem requests for new events, sensor administrator can reply with events in this queue. If there are no new events, the cognition subsystem simply blocks. An event stored in the sensor administrator will be removed if this event is past its deadline, or has been viewed by all cognition workers in the cognition subsystem.

The sensor administrator owns more than one sensor workers to detect different kinds of environment signals. In some cases, sensor workers are not necessary. For example, an agent only receives messages from other agents. In that case, we have specially designed *couriers*, a type of workers, for delivering messages between administrators. Sensor workers are designed to monitor particular environment signals and report them to the sensor administrator. A sensor worker may contain some particular resources, such as a keyboard or a communication port. Other processes do not need to know the details of the resource.

Once we assign a sensor worker to monitor some signals, we do not need to control this sensor worker any more. This sensor worker automatically repeats monitoring signals and issuing reports to the sensor administrator, which only needs to wait for new requests/reports.

Ideally a sensor administrator may have many sensor workers. As long as the sensor administrator knows how to preprocess these messages and signals captured by the workers, we can add/drop any workers without reprogramming the administrator. If we want to add some workers for new signals, we only need to add some new preprocessing rules in the administrator.

3.3.2 The Cognition Workers

The cognition workers are responsible for mapping events to tasks. Suppose we want to adopt Brooks's subsumption architecture [7] in the cognition component. We can use more than one cognition worker, connected in parallel between input and output. Every cognition worker can be seen as a set of rules or a finite state machine implementing a layer, with the lower layers governing the basic behavior and the upper layers adding more sophisticated control behavior. If a cognition worker is free, it sends a request to the sensor administrator for new events. After receiving a reply message, the cognition worker maps the received event to a set of tasks, which are sent to the task administrator, and moves on to process other events, if any.

The cognition workers determine the cognition level of an agent. If reaction rules are used for mapping events, then we get a reactive agent. We can also design a rational agent by building a world model in these cognition workers (or some of them) and perform reasoning on them. However, there is time consideration in deciding the level of reasoning that the cognition workers should perform.

3.3.3 The Task Administrator, the Scheduler Worker and Executor Workers

The action subsystem consists of the task administrator, the scheduler worker and executor workers. These components cooperate with one another to dispatch and execute tasks as efficiently as possible, while adhering to the timing and priority constraints. In many real-time applications, tasks have different *priorities*, which indicate how important a task is. If an agent is also in overload state, which means it is impossible to finish all tasks in time, the agent must be able

to handle and complete as many high priority tasks as possible. To achieve this end, we employ on-line scheduling algorithms for task dispatching.

The task administrator receives tasks generated by the cognition workers and stores them in a *task queue*. The administrator contains also a greedy scheduling algorithm to schedule the received tasks. This greedy algorithm must have the following two characteristics. First, the algorithm must be efficient, since an administrator cannot afford to perform heavy computation, deterring its response to important events. Second, the algorithm should be able to produce reasonable quality, albeit sub-optimal, schedules. When the scheduler worker cannot respond in time with a better scheduling result, the action subsystem will have to rely on results of this greedy algorithm to ensure continuous functioning of the subsystem and also the agent as a whole.

The scheduler worker maintains a task queue which is synchronized with that in the task administrator. This worker should employ an advanced scheduling algorithm to try to achieve global optimal scheduling results, and sends the result back to task administrator. While efficiency is still a factor, the more important goal of the worker is in producing good quality scheduling result, perhaps, at the expense of extra computation time. Once the task administrator receives results from the scheduling worker, it will combine the results with those of its own greedy algorithm and allocate the queued tasks to the executor workers for actual deployment. More details of the combined scheduling mechanism are introduced in following section.

An agent can have one or more executor workers, each in charge of a different execution duty. Similar to the sensor workers, executor workers enjoy full autonomy in terms of task execution without intervention from the task administrator. After finishing a task, an executor worker sends a request to report to the task administrator and wait for new assignment. Executor workers can encapsulate resources, such as a printer or the screen. The task administrator does not need

to know the details of these resources and how they are handled. The administrator allocate tasks according to only the task nature (and which executor work can handle such tasks) and the priority (including deadline). Thus we can easily add/drop executor workers.

3.4 An Agent-Based Real-time Arcade Game

To demonstrate the viability of our proposal, we construct a multi-agent implementation of the real-time arcade style game "Space Invader." In this game, a player uses the keyboard to control a laser gun on the ground to defend against flying space invaders. The game implementation consists of five real-time collaborating agents: input agent, game environment agent, game administrator agent, timer agent, and screen agent. Figure 3.5 illustrates the system architecture of the demonstration game.

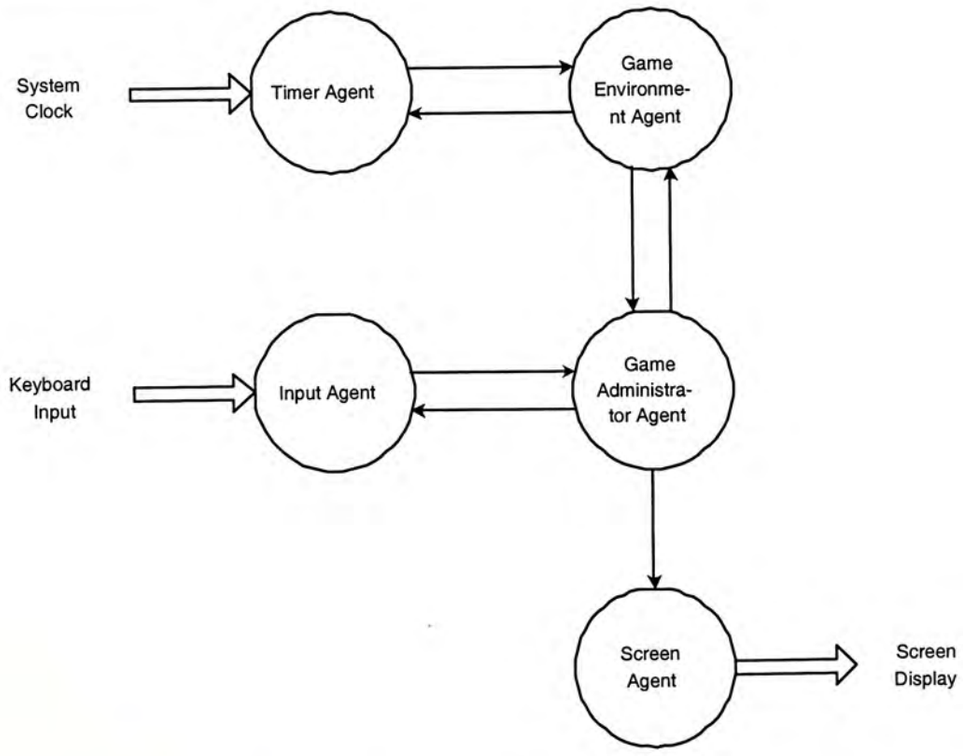

Figure 3.5: Architecture of the Demonstration Game

The input agent controls the keyboard input, the timer agent controls time events, and the screen agent controls output to screen. These are system agents responsible for common game tasks (low level I/O and devices). They can be reused in all real-time game implementations.

The game administrator agent stores the world model and determines the interactions in the world. The game environment agent controls all time-triggered events in the world, such as the movement of enemies.

We build these agents as reactive agents. The cognition subsystem of every agent is controlled by a set of reaction rules.

The rules in the input agent:

- *Rule 1: if keyboard input received then send user input message to the game administrator agent.*
- *Rule 2: if game end message received then end all components in this agent and release resources.*

The rules in the timer agent:

- *Rule 1: if system timer signal received then send time message to the game environment agent.*
- *Rule 2: if game end message received then end all components in this agent and release resources.*

The rules in the game environment agent:

- *Rule 1: if time received then check time-triggered events.*
- *Rule 2: if time-triggered event found then send time-triggered event message to the game administrator.*

- *Rule 3: if model change message received then update time-triggered events.*
- *Rule 4: if game end message received then end all components in this agent and release resources.*

The rules in the game administrator agent are:

- *Rule 1: if user input received then update the model.*
- *Rule 2: if time-triggered event message received then update the model.*
- *Rule 3: if model updated then check its rationality.*
  - *Rule 3.1: if laser beam hits the enemy then the enemy and the laser beam vanished, model changed.*
  - *Rule 3.2: if bomb hits the laser gun then the laser gun and the bomb vanished, model changed, and game ends.*
  - *Rule 3.3: if laser beam and bomb hit each other then laser beam and bomb vanished, model changed.*
- *Rule 4: if model changed then send model change message to the game environment agent.*
- *Rule 5: if model changed then output new model to screen agent.*
- *Rule 6: if game ends then send game end message to the input agent, game environment agent, timer agent, and screen agent, end all components in this agent and release resources.*

The rules in the screen agent:

- *Rule 1: if new model received then update screen display.*

- *Rule 2: if game end message received then end all components in this agent and release resources.*

These sets of if-then rules is enough for a simple game. In more complicated applications, user may need a finite state machine or a set of reasoning rules to control the cognition of agents.

Chapter 4

A Multiple Method Approach to Task Scheduling

Task scheduling is an important issue in our agent architecture. In real world applications, an agent may be in overload state, which means it is impossible to finish all tasks in time. Agents must be able to handle and complete tasks as quickly and many as possible. Efficient task scheduling is essential for real-time response.

This chapter is organized as follows. Section 4.1 introduces the task scheduling mechanism in detail and gives the pseudocode of the task administrator and task scheduler. Section 4.2 models the task scheduling problem formally. Section 4.3 introduces some combination rules used in combining the results of the two algorithms and special cases during runtime. Section 4.4 describes algorithms which can be plugged into this mechanism.

4.1 Task Scheduling Mechanism

Our approach combines two different on-line algorithms for task scheduling. The greedy scheduling algorithm, usually simple and fast, used in the task administra-

tor opts for efficiency, but there is no guarantee on the quality of the scheduling results. An example is the *Earliest-Deadline-First* (EDF) algorithm. The complexity of greedy algorithms are usually linear in nature, so that they work well also in heavy load situation.

On the other hand, the advanced algorithm in the scheduler worker opts for solution quality. An example is local search algorithm [9] for finding a suboptimal performing tasks order. These algorithms, however, usually suffer from at least a quadratic complexity. They might not be able to respond in a timely manner in a heavily loaded real-time environment. The idea is to combine the greedy and the advanced algorithms so that they can supplement each other.

With the combination, we obtain the best of both worlds. When the time constraint is not strict, the mechanism should be able to produce high quality solutions; when the system is in heavily loaded state, however, the mechanism would ignore complicated and time-consuming scheduling computation when efficiency is needed most. That is when the greedy algorithm comes into place.

4.1.1 Task and Action

A *task* consists of a set of jobs that the agent needs to do, such as printing a set of documents on printer, sending a message to other agents, etc. Tasks are independent of one another. Since tasks are generated by cognition worker and arrived at the task administrator in succession. If a task in task administrator is related with other tasks which are still not arrived yet, this task is hard to schedule. Task administrator cannot predict when the successive tasks arrive and the result of performing this task without consider its successors.

For example, there are two tasks T_1 and T_2 in a system. T_1 has arrived at the task administrator and T_2 has not. T_1 and T_2 have *alternative* relation, which means that only one of them needs to be performed. Once T_1 or T_2 has started

execution, the other can be abandoned.

In this case, the task administrator cannot decide if it should perform T_1 immediately or wait for T_2. If the task administrator performs T_1 and T_2 arrives soon after, which requires only much less time and resources, the performance of the system is dropped from this wrong decision. On the other hand, the task administrator may choose waiting, but when T_2 finally arrives, the task administrator may sadly discover that besides the waiting time, T_2 requires even more performing time and resources.

We require tasks to be independent of one another to avoid unpredictability. In real world applications, relations among different jobs are common. To describe these relations, we define a task to be composed of a set of *actions*. An action can be viewed as the atomic unit of jobs. Once an action starts performing, it cannot be interrupted. It is non-preemptable. Actions in the same task may depend on one another, but not on actions in other tasks. Three kind of relations among actions are defined (A_1 and A_2 are actions in the same task):

- **Parallel** (A_1, A_2). A_1 and A_2 have parallel relation if and only if both A_1 and A_2 need to be performed and they can be performed in any order. A_1 and A_2 do not affect each other.
- **Sequential** $(A_1 \prec A_2)$. A_1 and A_2 have sequential relation if and only if both A_1 and A_2 need to be performed and A_2 can start only after A_1 is finished.
- **Alternative** $(A_1 \mid A_2)$. A_1 and A_2 have alternative relation if and only if the system can choose only either one (and exactly one) to perform.

Complicated tasks can be defined using these relations. For example, an user may want to print a set of documents (Doc_1, Doc_2, Doc_3) on two available printers (P_1, P_2). He hopes that these three documents can be printed in the same printer

(so that he does not need to go to two locations to gather the output), and a notification indicating which printer is used can be sent to the user after these documents are printed (so he knows the output is ready for collection). In this task, the actions can be defined as following:

$$Task = (((A_{11}, A_{21}, A_{31}) \mid (A_{12}, A_{22}, A_{32})) \prec A_4)$$

Where A_{ij} means printing document Doc_i on print P_j, and A_4 means issuing the final notification.

This task also can be represented as figure 4.1.

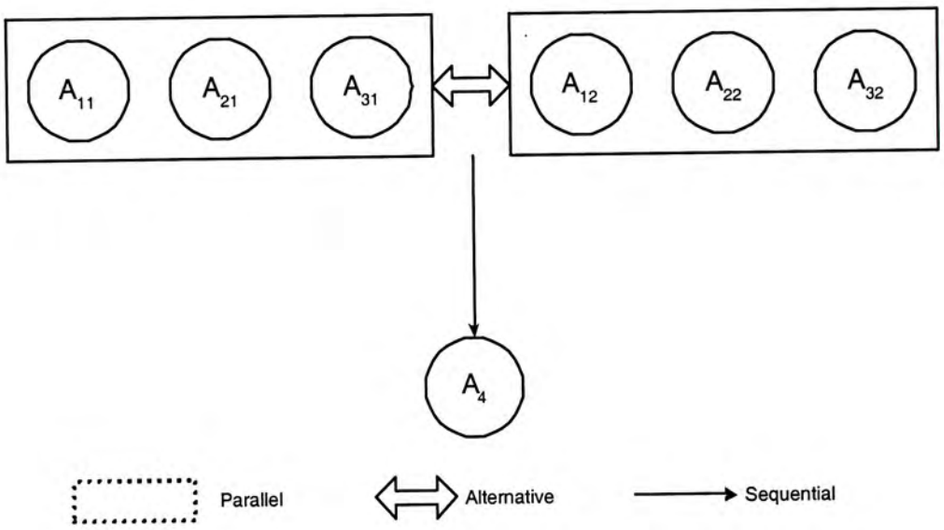

Figure 4.1: Different Relations among Actions

4.1.2 Task Administrator

The task administrator stores and manages tasks generated by the cognition worker. The administrator consists of a task queue, a greedy scheduling algorithm, next-ready-action-places (NRAP), and a scheduling result buffer (shown in Figure 4.2).

When new tasks arrive, the greedy algorithm, usually linear in nature, inserts the new tasks into the task queue according to a set of particular criteria, such

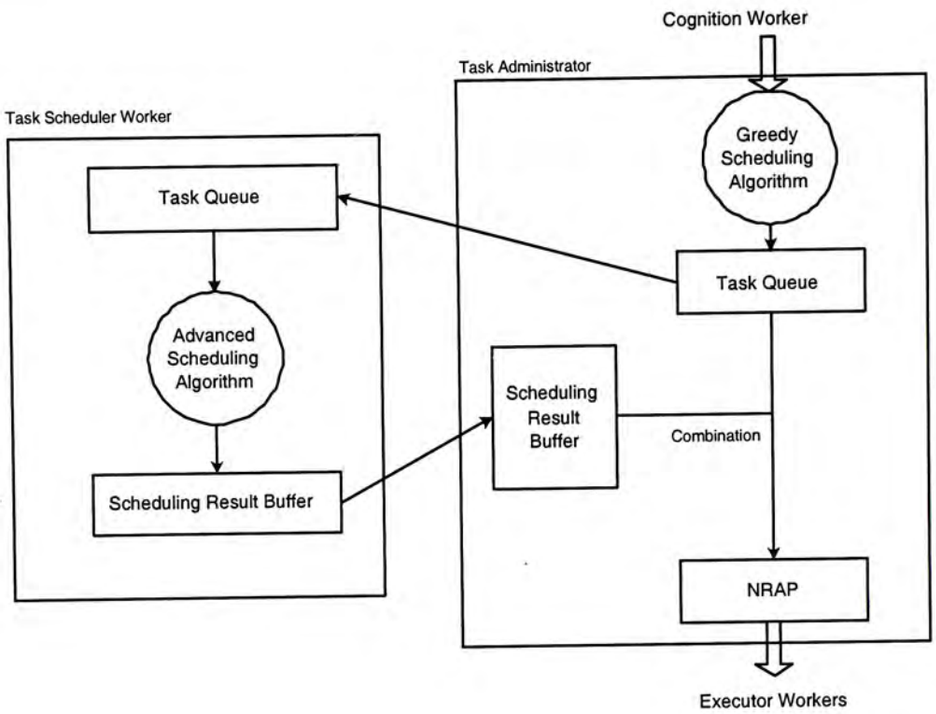

Figure 4.2: The Task Administrator and the Scheduler Worker

as the associated deadlines of the tasks. This maintenance of sortedness allows the task administrator to dispatch the next available task almost instantly.

NRAP, next-ready-action-places, is a buffer for storing the next ready task for each executor worker, which has an associated *slot* in NRAP. Once an executor worker finishes a task, the worker reports to the task administrator, which replies with the ready task in the corresponding slot in NRAP, if available, to the worker. If the slot for the worker in NRAP is empty, the task administrator makes this executor worker idle, and searches for the next ready task from the scheduling result buffer to fill the slot in NRAP. If there are no task in the scheduling result buffer for the executor worker, search proceeds to the task queue.

The scheduling result buffer stores the scheduling result given by the scheduler worker obtained using the advanced algorithm. This result, supposingly of higher quality, always subsumes that of the greedy algorithm. That is why the task

administrator always searches this result buffer for the next available task first, before resorting to the task queue.

```
initialization;
while running do
    receives any messages;
    if message sender is the cognition worker then
        run the greedy algorithm to sort and store the newly arrived task;
        update executor workers and NRAP;
        if the scheduler worker is blocked then
            unblock the scheduler worker;
            reply to the scheduler worker with all new information;
        end
    end
    if message sender is executor worker W_i then
        store the result;
        if the ith slot of NRAP is not empty then
            reply to W_i with actions in the i slot of NRAP;
        end
        block W_i;
        if the scheduler worker is blocked then
            unblock the scheduler worker;
            reply to the scheduler worker with all new information;
        end
        update executor workers and NRAP;
    end
    if message sender is the scheduler worker then
        record the new scheduling result in the result buffer;
        if nothing happened from last scheduling then
            block task scheduler;
        end
        update executor workers and NRAP;
    end
end
```

Algorithm 1: Pseudocode of the Task Administrator

Both new task arrival and new result from the scheduler worker can trigger the task administrator to update NRAP and reply to blocking executor workers (and reply to blocking scheduler worker, if it is blocked). More special cases

in scheduling are illustrated in the following section. Algorithm 1 gives the pseudocode of the task administrator.

4.1.3 Task Scheduler

The scheduler worker is quite simple, consisting of only three main components: a task queue, an advanced scheduling algorithm, and a scheduling result buffer. Figure 4.2 illustrates these components and communication among them.

The task queue in the scheduler worker are kept synchronized with that in the task administrator. Using this task queue as input, the advanced algorithm in the scheduler worker outputs and stores the scheduling result in the scheduling result buffer. Once the advanced algorithm finishes, the scheduler worker sends the content of the result buffer to the task administrator. This message contains a sequence of tasks which indicates the task performing strategy suggested by the task scheduler. Upon receiving the result, the task administrator stores it, and replies to the scheduler worker with information of new tasks, if any. The scheduler Worker then updates its task queue and starts the scheduling cycle again.

The task administrator replies a message which contains updated task queue to the task scheduler. For minimizing the data transmission, this message only contains following components:

- For newly arrived tasks which are not contained in the task queue of the task scheduler, the message contains their task ID, actions, position in the task queue, and other information which is necessary for scheduling, for example, priority, deadline, etc.
- For tasks which has been included in the task queue of the task scheduler but has changed since last scheduling, the message only contains their task

ID and the updated information.

- For tasks which has been included in the task queue of the task scheduler and has not changed since last scheduling, the message does not contain their information.
- The message also contains the current states of NRAP and the executor workers.

Algorithm 2 gives the pseudocode of the scheduler worker.

```
initialization;
while running do
    send new scheduling results to task administrator;
    use the replied message to update task queue;
    run the advanced scheduling algorithm;
    store new results in scheduling result buffer;
end
```

Algorithm 2: Pseudocode of the Scheduler Worker

4.2 A Task Scheduling Model

We model the hybrid scheduling mechanism as a quadruple (T, A, S, W), where T describes the task list, A provides details of the task administrator, S contains description of the scheduler worker, and W is the set of executor workers. Task list T is comprised of a sequence of tasks $(T_1, \ldots, T_n)$. A *task* in turn consists of a set of *actions*, which are the actual jobs that must be completed by the agent. Each executor worker can be responsible for one or more types of actions. There can be precedence constraints among actions in a task, but tasks are independent of one another.

Each task T_i has the following attributes:

- The *arrival time* represents when task T_i first reaches the task administrator. In real-time environments, events occur with a certain distribution, so do tasks which are usually generated as a result of some events taking place. Once a task T_i arrives, the task administrator and the scheduler worker take over to schedule and dispatch T_i for execution by corresponding executor workers.

- The *deadline* specifies the time by which a task must be finished. Completing a task after its deadline is non-fruitful and wasteful of computation resources. The arrival time and the deadline defines the *lifespan* of task T_i.

- The *priority* (or weight) of a task represents the importance of the task relative to the other tasks in the agent. The profit of a task is usually a measure of the utility of completing the task.

- Every task contains a set of *actions* $\{A_1, \ldots, A_s\}$, which are atomic in nature. Thus, in our model, we allow preemption at the task level, but not at the action level. Within the same task, actions can depend on one another. We allow three kinds of relation *Parallel*, *Sequential*, and *Alternative* among actions, which we have defined in Section 4.1.1. Each action A_j is associated with the following attributes:

  - The *execution time* is the estimated time to complete A_j.
  - The execution worker responsible for A_j.
  - The possible relations (parallel, sequential, and alternative) with other actions within the same task as A_j helps to define complex tasks.

The A field describes the task administrator. Different real-time agent instances can have different task administrators, but these administrators share essentially similar structure and components. The only difference is the greedy scheduling algorithm they employ. This greedy algorithm also represents a task

administrator's attitude towards tasks. A task administrator using the EDF algorithm thinks that urgent tasks should be handled first, but while another administrator using the HPT (*highest-priority-first*) algorithm gives way to more important tasks.

The S field contains description of the scheduler worker. Similar to the case of the task administrator, the advanced scheduling algorithm can give rise to the only possible difference among various scheduler workers. An agent may use an extremely complex algorithm for optimal results, or just use a relatively simple algorithm for a lower complexity.

The W field defines the set $\{W_1, \ldots, W_m\}$ of available executor workers. Each executor worker is capable of performing one or more kinds of action. These workers work concurrently. The more workers an agent has, the more actions can be performed at the same time.

4.3 Combination Rules and Special Cases

In this section, we introduce the combination rules used in the hybrid mechanism and discuss the special cases during scheduling. These rules are used to decide which task will be assigned to executor workers or NRAP.

First, we introduce what happens when a new task T_{new} arrives into the task administrator. In this case, the task administrator tries to use this task to suffice free executor workers and NRAP first, then stores this task in the task queue:

1. The task administrator first checks if there exists blocking executor worker or empty NRAP.

   - Blocking executor worker: If there exists blocking executor worker W_j , the task administrator checks if actions in the newly arrived task can

be assigned to this worker. If there $\exists$ action A_i, $A_i \in T_{new}$ and the execution worker responsible for A_i (we present it as $w_{new,i}$) is W_j, W_j is blocking, then the task administrator allocates this action A_i to the executor worker W_j.

- Empty NRAP: Similarly, the task administrator tries to allocate actions in this task to the empty NRAP. If there $\exists A_i$, $A_i \in T_{new}$ and $w_{new,i} = W_j$, $NRAP_j = \emptyset$, then $NRAP_j = A_i$.

2. Afterwards, the task administrator checks non-empty NRAPs and tries to use the newly arrived task to update them. If there $\exists A_i$, $A_i \in T_{new}$ and $w_{new,i} = W_j$, and the original action in NRAP is A_{old}, $A_{old} \in T_{old}$, and $p_{old} < p_{new}$(assuming p_i is the priority of task T_i), A_{old} is not chosen from the scheduling result buffer, then $NRAP_j = A_i$ (condition $p_{old} < p_{new}$ also can be replaced by $t_{deadlineold} > t_{deadlinenew}$. It is depending on which scheduling strategy we use).

3. Run the greedy algorithm, insert the new task into the task queue.

4. Finally, check if the task scheduler worker is blocking. If the scheduler worker is blocking, the task administrator unblocks the scheduler worker and replies it with the updated task queue.

Second, when an executor worker W_i finishes its job, it issues *Send()* to send a report to the task administrator. After receiving this report, the task administrator knows the result of the last action performed by W_i and W_i is ready for new action:

1. The task administrator first checks if the corresponding NRAP is empty. If $NRAP_i = \emptyset$, then W_i blocks. Otherwise, the task administrator allocates action in $NRAP_i$ to W_i, $NRAP_i = \emptyset$.

2. If $NRAP_i = \emptyset$, search the scheduling result buffer to find a new action for $NRAP_i$.

3. If $NRAP_i = \emptyset$, search the task queue to find a new action for $NRAP_i$.

4. Unblock the task scheduler worker and reply it with the updated task queue if one or more following conditions are satisfied.

   - New tasks arrived since the latest communication with the task scheduler worker.
   - Actions finished since the latest communication with the task scheduler worker.
   - Any other changes in task queue.

Third, when the scheduler worker finishes scheduling tasks, it sends new scheduling results to the task administrator:

1. The task administrator stores the new result in the scheduling result buffer and replaces the old result. Some special cases during the replacing are listed as following:

   - Tasks in the new result have been finished. Due to various mechanisms, it is possible that a task which is re-scheduled by the task scheduler worker has been finished before the new result returns. In this case, the task administrator removes this task from the new result. Similarly, if a task is partly finished - some actions in this task has been performed - the task administrator also modifies the new result by removing these actions.
   - Content of tasks changed from last scheduling. Since the task scheduler worker only returns a name sequence but not the complete task list with their content, any change to the content of tasks in the task queue is also efficient to this result.

2. If there is no new task arrived or action finished or other changes in the task queue since last scheduling, the scheduler worker blocks. Otherwise, the task administrator replies the scheduler worker with the updated task queue.

3. Uses the newly received scheduling result to reply blocking executor workers.

4. Use the newly received scheduling result to update all NRAP, includes these actions selected from the old scheduling result buffer and the task queue. The task administrator always believes that the newly result given by the task scheduler worker is the most appropriate result.

4.4 Scheduling Algorithms

Various scheduling algorithms can be plugged into this on-line scheduling mechanism. In this section, we introduce some basic algorithms that can be used in the architecture.

From the strategy of the on-line scheduling algorithms, we can classify them into following types [32].

1. **FIFO.** The FIFO algorithm does not make sense in most real-time environments. This algorithm approach to control the order in which tasks is performed completely works without consider any efficiency issues: FIFO serves tasks in the order of appearance.

   The disadvantage of FIFO algorithm is obvious. Although the FIFO algorithm is almost the poorest scheduling algorithm, it also has some advantages. First, FIFO algorithm does not cost time in scheduling. It is one of

the fastest scheduling algorithms. Second, in some real world applications, such as the producing line, the first come first serve strategy does work.

2. **Greedy.** An scheduling algorithm is *"greedy"* means this algorithm always makes a "locally most promising" decision. Greedy algorithms do not take into account the possible future and the global sight. For example, a greedy algorithm only decides upon the next request to be served, it does not plan into the future or does not consider the system state after the service.

   Although the above greedy algorithm is very shortsighted and the result maybe sub-optimal it is very popular because it has following advantages:

   - Easy to implement
   - Usually real-time compliant
   - It produces a stable, predictable behavior since no decision is revised

   Here we introduce some typical greedy algorithms. These algorithms also been used in following implementation.

   - **EDF.** The *Earliest Deadline First* (EDF) is one of the most famous scheduling algorithms. In EDF algorithm, when a request is received, the algorithm chooses the task that has the earliest deadline in all tasks to serve the request.
   - **HPF.** In *Highest Priority First* (HPF), the algorithm always uses the task which has the highest priority in all tasks to serve request.

3. **Replan.** The Replan algorithms for an on-line problem compute an optimal (or almost optimal) solution to the static optimization problem (the current state of the on-line problem) at a specific point in time.

   While the greedy algorithms acts as locally as one think, replan algorithms are another extremes case: at any time replan algorithms try to find a

solution which is as globally optimal as possible, with the information it has at that time.

Replan algorithms maintain a "plan" containing the result of scheduling the already known tasks. This result is followed as long as no relevant event happens. Whenever a relevant event happens, such as a new task arriving, the algorithms need to reschedule all current tasks. At any point in time, replan algorithms keep an optimal solution that is globally optimal at that particular moment.

Replan algorithms have the following disadvantages. First, to find an optimal scheduling result is an NP problem. It is a time-consuming strategy. Second, replan algorithms may completely revise all results, which was still possible a short time ago. This will lead to unpredictable behavior over time.

4. **Ignore.** Ignore algorithms assume that we have a way of computing optimal (or suboptimal) solutions of the off-line version of the problem. The main idea of this method is to make sure that every step of the on-line solution is part of the solution of off-line problem.

   Ignore algorithms also contain a plan. Unlike replan algorithms, ignore algorithms do not rebuild the plan but finish it. All new arrived tasks were stored and ignored. Until this plan is finished, the ignore algorithm computes a new plan with all stored tasks.

   Although the final solution may be not the optimal solution, this solution is also sub-optimal in every particular step. Comparing with replan algorithms, ignore algorithms can save lots of system resources used on scheduling.

Both FIFO and greedy scheduling algorithms have short response time. In our architecture, we can use them in the task administrator. Replan and ignore

algorithms have good solution quality but long response time; we can use them in the task scheduler.

Chapter 5

Task Scheduling Model: Analysis and Experiments

In this chapter, we introduce the theoretical analysis and experimental results of the hybrid task scheduling mechanism. Section 5.1 introduces the measurement of the quality of on-line mechanisms. Section 5.2 explains the theoretical analysis of the hybrid mechanism and gives the estimated bounds. In Section 5.3, we present the simulation system implemented to test the scheduling mechanism. Section 5.4 gives the experimental results and discussions.

5.1 Goodness Measure

For any scheduling problems, suppose we know future in advance, we can easily determine how to achieve the maximal profit and/or the minimal cost. This is the *off-line problem.* The solution to the off-line problem is called the *optimal solution* and the cost/profit of the optimal solution is called the *optimal cost/profit* [53].

Scheduling is difficult in general. The added complexity to on-line scheduling is that there is no way to know the exact arrival patterns of the tasks in advance. Tasks arrive independently over time, and the existence of a task is

not known until its arrival. If the future is known, the problem is reduced to off-line scheduling, in which a globally optimal solution can be computed. A well adopted measure of the goodness of on-line scheduling algorithms is *competitive ratio*, which is a ratio between the off-line optimal solution and the on-line solution:

$$r = \frac{\text{Profit of On-line Solution}}{\text{Profit of Off-line Solution (Optimal Solution)}}$$

In this definition, the higher the competitive ratio an algorithm has, the better the algorithm is. The upperbound of r is 1. When $r = 1$, this on-line algorithm is an optimal algorithm.

In our simulation system, we assign a profit value p_X for every task T_X. Once an agent successfully finishes a task, it receives the profit as a reward for performing this task. The object of agent is to achieve the maximal profit.

This profit value also can be viewed as the priority of a task. The higher the profit value a task has, the more profit the agent gets. Agent is apt to perform these tasks as they are more "important" than other tasks.

We also can assign all tasks with the same profit. In this case, agent is inclined to finish as many tasks as possible.

5.2 Theoretical Analysis

We identify the following main factors affecting the performance of our proposed scheduling mechanism.

- **Actions:** Longer average execution time of the actions gives more room for the advanced scheduling algorithm to complete its computation.
- **The Advanced Scheduling Algorithm:** An algorithm of too high a

complexity can seriously degrade the performance of the agent in terms of scheduling result quality.

- **Overload Factor ($f_{overload}$).** The overload factor is defined by:

$$f_{overload} = \frac{\text{Total Execution Time}}{\text{Total Available Time}}$$

where "Total Execution Time" is the sum of all tasks' execution time, and "Total Available Time" is the during between the earliest arrival time and the latest deadline. The factor indicates if the agent is overloaded with tasks. If $f_{overload} \geq 1$, the system is overloaded; otherwise, the system is in non-overload state.

- **Competitive Ratio of Greedy Algorithm ($c.r.g$).** This parameter describes the competitive ratio of the greedy algorithm used in the task administrator.

- **Competitive Ratio of Advanced Algorithm ($c.r.a$).** This factor is the competitive ratio of the advanced algorithm in the scheduler worker.

When the system is in non-overload state, $f_{overload} < 1$, the scheduling mechanism is insignificant. In non-overload state, even a simple greedy algorithm can achieve the optimal solution. For example, the EDF algorithm is optimal in non-overload state [42]. In the following discussion, we focus only on situations in which the system is in overload state.

Suppose $t_{scheduling}$ is the time used by the scheduler worker to run the advanced algorithm and $t_{performing}$ is the time used by the task administrator to finish all tasks ordered by the scheduler worker. As shown in the left-hand graph in Figure 5.1, if $t_{scheduling} \leq t_{performing}$, then the task administrator always utilize results from the advanced algorithm. Our proposed multiple method approach takes effect only when $t_{scheduling} > t_{performing}$, as shown in the right-hand graph in Figure 5.1.

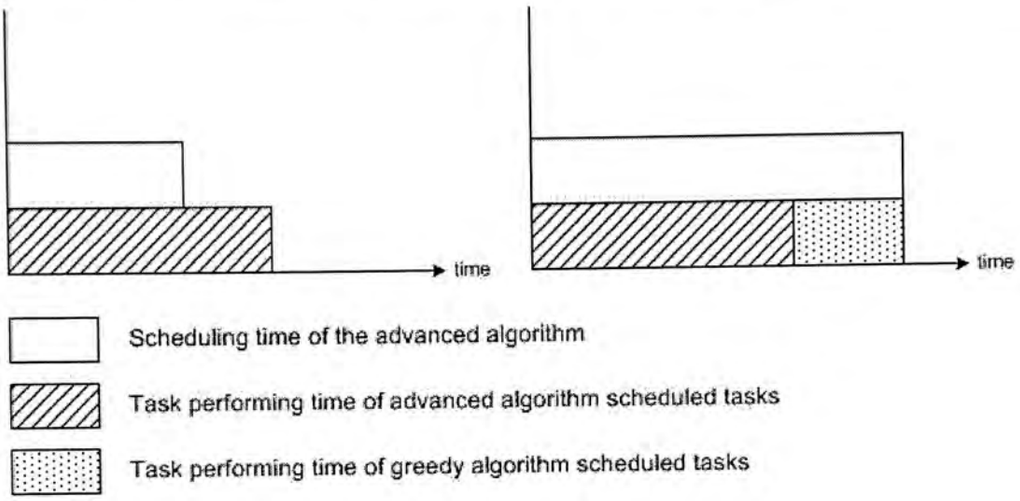

Figure 5.1: Task scheduling time and task performing time

Suppose $t_{scheduling} > t_{performing}$ and task list T is $(T_1, \ldots, T_n)$, stored in the task queues of both the task administrator and the scheduler worker. We can thus partition T into $T_G = (T_{G_1}, \ldots, T_{G_{S_G}})$ and $T_A = (T_{A_1}, \ldots, T_{A_{S_A}})$, where tasks in T_G are dispatched and executed as a result of the greedy algorithm and tasks in T_A are executed as a result of the advanced algorithm.

Given $c.r.g$ and $c.r.a$ as the competitive ratio of the greedy algorithm and the advanced algorithm respectively. For task T_X, we use p_X to denote the *profit* of T_X. The goal of any scheduling algorithm is to achieve as high a profit as possible. We can then define the competitive ratio of our mechanism $c.r.$ as follows:

$$c.r. = \frac{\text{on-line profit}}{\text{optimal profit}} = \frac{\sum_{i=1}^{S_A} p_{A_i} + \sum_{j=1}^{S_G} p_{G_j}}{S_A + S_G} = \frac{S_A \cdot c.r.a. + S_G \cdot c.r.g.}{S_A + S_G} \tag{5.1}$$

The greedy algorithm only works when the advanced scheduling algorithm cannot give a scheduling result in time. Once the scheduler worker gives new scheduling results, the task administrator stops running the greedy algorithm and uses these results. In this case, the scheduler worker repeats running the advanced algorithm without any delay. It also means that the time used by the task administrator to perform all tasks given by the greedy algorithm and the

advanced algorithm should be equal to the time used by scheduler worker in scheduling. We define $t_{X_{exe}}$ to be the total execution time of task T_X.

$$\sum_{i=1}^{S_A} t_{A_{iexe}} + \sum_{j=1}^{S_G} t_{G_{jexe}} = \text{Scheduling Time of } T_A \tag{5.2}$$

For a given scheduling algorithm and task list, it is possible to estimate the time the algorithm used in scheduling this task list. For example, if we use an algorithm which has the complexity of $O(n \log n)$ to schedule a task set $T_{A_1}, \ldots, T_{AS_A}$, then we can estimate the upper bound of the scheduling time as $S_A \log S_A$. Let $\alpha = \log S_A$, so we have:

$$\text{Scheduling Time of } T_A \leq S_A \alpha \tag{5.3}$$

Similarly, we can define α for any given scheduling algorithm and task list. So (5.3) is valid in all cases.

For T_A and T_G, we define t_{avgA} and t_{avgG} as the average execution time of tasks in T_A and T_G respectively. From (5.2) and (5.3), we have:

$$S_G \cdot t_{avgG} + S_A \cdot t_{avgA} \leq S_A \alpha \rightarrow S_G \leq S_A \frac{\alpha - t_{avgA}}{t_{avgG}} \tag{5.4}$$

Combining (5.1) and (5.4), we get

$$\begin{aligned} c.r. &= \frac{S_A \cdot c.r.a + S_G \cdot c.r.g}{S_A + S_G} = \frac{S_A(c.r.a + c.r.g\frac{\alpha - t_{avgA}}{t_{avgG}})}{S_A(1 + \frac{\alpha - t_{avgA}}{t_{avgG}})} \\ &\leq \frac{c.r.a - c.r.g}{1 + \frac{\alpha - t_{avgA}}{t_{avgG}}} + c.r.g = c.r.a - \frac{c.r.a - c.r.g}{1 + \frac{t_{avgG}}{\alpha - t_{avgA}}} \end{aligned} \tag{5.5}$$

From (5.5), we can see the parameters that can affect the performance of the system. The chosen greedy and advanced algorithms determines $c.r.a$ and $c.r.g$ respectively. The advanced algorithm fixes also α. The quantities t_{avgA} and t_{avgG} depend on the distribution of the execution time of tasks. Simplifying this model

further, we assume that $t_{avgA} = t_{avgG} = t_{avg}$.

$$\begin{aligned} c.r. \leq c.r.a - \frac{c.r.a - c.r.g}{1 + \frac{t_{avgG}}{\alpha - t_{avgA}}} &= c.r.a - (c.r.a. - c.r.g)\frac{\alpha - t_{avg}}{\alpha} \\ &= c.r.g + (c.r.a. - c.r.g)\frac{t_{avg}}{\alpha} \end{aligned} \tag{5.6}$$

While (5.5) gives the upper bound of the system's competitive ratio, the lower bound of the system is determined by $c.r.a$ and $c.r.g$.

$$c.r. \geq Min(c.r.a, c.r.g) \tag{5.7}$$

The entity α can be viewed as the average scheduling time per task. For a given task list and advanced scheduling algorithm, we can estimate the upper bound of α. Suppose the number of tasks is n, and we choose an algorithm of order $O(n^2)$ from the *Ignore* [32] family of algorithms as the advanced algorithm. In an *Ignore* algorithm, every task is scheduled once and the results are committed. Thus the scheduler worker always appends newly scheduled tasks to old results. During processing, since tasks arrive over time, the task list may be partitioned and handled as m sublists. Assuming the number of tasks of these subsets are $N_1, \ldots, N_m$, we have

$$N_1 + \cdots + N_m = n, N_i \geq 1, n \geq 1, m \geq 1$$

The total scheduling time of these task sublists is

$$N_1{}^2 + \cdots + N_m{}^2 \leq n^2.$$

Thus

$$\frac{N_1{}^2 + \cdots + N_m{}^2 \leq n^2}{n} = \alpha \leq n \tag{5.8}$$

Given the fact that most scheduling algorithms, except perhaps FIFO, have an order of at least $O(n)$. Actually, for any *Ignore* algorithm of order $O(n^i), i \geq 1$, we have

$$\frac{N_1{}^i + \cdots + N_m{}^i \leq n^i}{n} = \alpha \leq n^{i-1} \tag{5.9}$$

The time used by the advanced algorithm reaches its upper bound only when all tasks arrive together and are scheduled in one go.

Members of the *Replan* [32] family of scheduling algorithms reconsider scheduled tasks when new tasks arrive. For a *Replan* algorithm of order $O(n^2)$, we can estimate the upper bound as (assuming tasks are partitioned into m sublists, and sublist s has N_s tasks)

$$\begin{aligned} {N_1}^2 + (N_1 + N_2)^2 + \cdots + (N_1 + \cdots + N_m)^2 \leq \\ 1^2 + \cdots + n^2 = \tfrac{1}{6}n(n+1)(2n+1) \end{aligned} \tag{5.10}$$

Thus $\alpha \leq \frac{1}{6}(n+1)(2n+1)$.

5.3 Implementation

The hybrid task scheduling mechanism described in Chapter 4 has been implemented as a simulation system on QNX. This prototype is based on the agent architecture described in Figure 4.2. This system is composed of the following components: a task generator, a task administrator, a task scheduler, and executor workers. Figure 5.2 illustrates the implementation of the simulation system. Each component is implemented as a process which uses message passing as the IPC mechanism.

This section introduces some details in the implementation. Section 5.3.1 explains the implementation of the task generator. The implementation of the executor workers is given in Section 5.3.2.

5.3.1 Task Generator Implementation

We use the task generator to replace the perception and the cognition subsystems in the agent architecture. The task generator generates the entire task list before

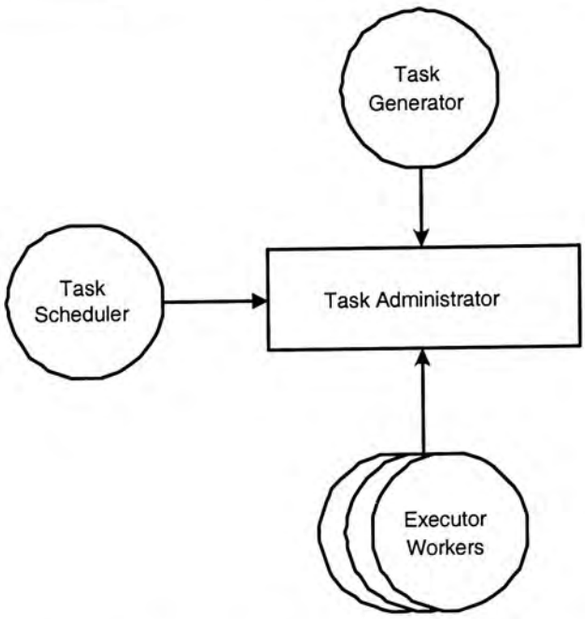

Figure 5.2: Implementation of the Simulation System

scheduling, and release the tasks in the list to the task administrator in real-time according to the generated arrival time of these tasks during runtime. The basic characteristics of the generated task list are parameterized by the following characteristics.

- *Average execution time.* This parameter defines the average execution time of actions.
- *Deadline.* This parameter defines the length of deadline comparing with the execution time of a task. The deadline of a task T_i is:

  $$\text{Arrival Time of } T_i + \text{Execution Time of } T_i \cdot \text{Deadline}$$

- *Action number.* This parameter defines action number in each task.
- *Action relations.* This parameter indicates the relations among actions.
- *Overload factor.* This parameter determines the interval among tasks. The higher the overload factor, the shorter the interval is, thus the task administrator needs to process more tasks in the same period.

The task generator randomly generates a task list which satisfies the specified characteristics. A task may contain more than one action, but we restrict the number of actions in a task up to eight for ease of implementation. For the same reason, we assume the number of executor workers in each agent is also up to 8. Every task is assigned a unique task ID as the "name" of this task when it is generated.

These generated tasks are ordered and stored according to the generated arrival times. The task generator maintains a system timer which fires off repeatedly. In each pulse, the task generator checks the first task in the task list. If the current time is great or equal to the arrival time of this task, the task generator sends a message which contains this task to the task administrator. The task administrator replies with an empty message as soon as the message is received. After receiving the reply message, the task generator removes the first task in the task list and waits for the next pulse.

Another function of the task generator is to compute the optimal cost of the given task list. After generating the entire task list, the task generator knows the arrival times and the performing time of all tasks. It is straight forward then to compute the optimal solution of this offline problem. This optimal solution is used to compute the competitive ratio of various on-line scheduling algorithms.

5.3.2 Executor Workers Implementation

The task administrator allocates actions to the executor workers. Once an executor worker acquires an action, this worker also knows the exact performing time of this action. In our simulation system, the executor workers do not actually perform the actions but only idle for the duration of the execution time of the current actions. The executor workers do not share resources with the task administrator and the task scheduler.

We assume that the executor workers always finish actions successfully. An executor worker sends message to the task administrator to report the result (if any) of the newly finished action. After receiving this message, the task administrator removes this action from the task queue.

In our implementation, the number of executor workers is restricted up to 8. Every action has an *ActionType* data field to indicate the type of this action. For example, an action may has *ActionType* as *SendMessage*, which means the content of this action is sending messages to other agents. Every executor worker has the ability of performing one or more kind of actions. After being generated, an executor worker registers its ID and *ActionType* of actions it can perform for the task administrator. For every action, the task administrator scans this information to select executor workers that can be used to perform it.

5.4 Experimental Results

This section details the tests suites we used and the test results. These results are based on the simulation system we described in Section 5.3, and run on Pentium III 850 MHz machine running QNX Neutrino.

In the following experiments, we choose EDF of order $O(n)$ as the greedy algorithm and an *Ignore* algorithm [32] of order $O(n^2)$ as the advanced algorithm. We use $f_{overload}$ to measure how overloaded a system is. The higher the overload factor is, the longer time the algorithm takes to schedule the tasks.

Table 5.1 gives the average scheduling time (in ms) for different states.

We can see that although the advanced algorithm can give better quality scheduling results in general, its efficiency worsens dramatically when there are too many tasks in the system. Assuming an average performing time $500ms$, the advanced algorithm would fail to respond in time. On the other hand, the greedy

Table 5.1: The average scheduling time of tasks (ms) in different algorithms and different overload states. $t_{avg} = 500ms$

Overload	1.0	2.0	3.0	4.0	5.0	6.0
EDF	9	16	25	37	47	56
Ignore	41	170	366	672	1102	1503
Combined	13	181	380	446	325	230

algorithm works well within the timing constraint; so is the hybrid algorithm.

5.4.1 Hybrid Mechanism and Individual Algorithms

Here we verify our upper bound and lower bound analysis empirically. Figure 5.3 compares the competitive ratio of the hybrid mechanism against those of the individual algorithms. The test suites are generated from the following parameters.

- Average execution time: 500ms.
- Deadline: 4.
- Action number: 8.
- Action relations: all.
- Overload factor: 1 to 8.

The estimated theoretical upper bound of the mechanism, according to (5.6), is also displayed in the figure. The theoretical lower bound is the minimum of the individual algorithms.

When $f_{overload} \leq 1.0$, the system is in non-overload state, all algorithms are optimal. When $f_{overload} > 1.0$, the quality of the algorithms begins to drop,

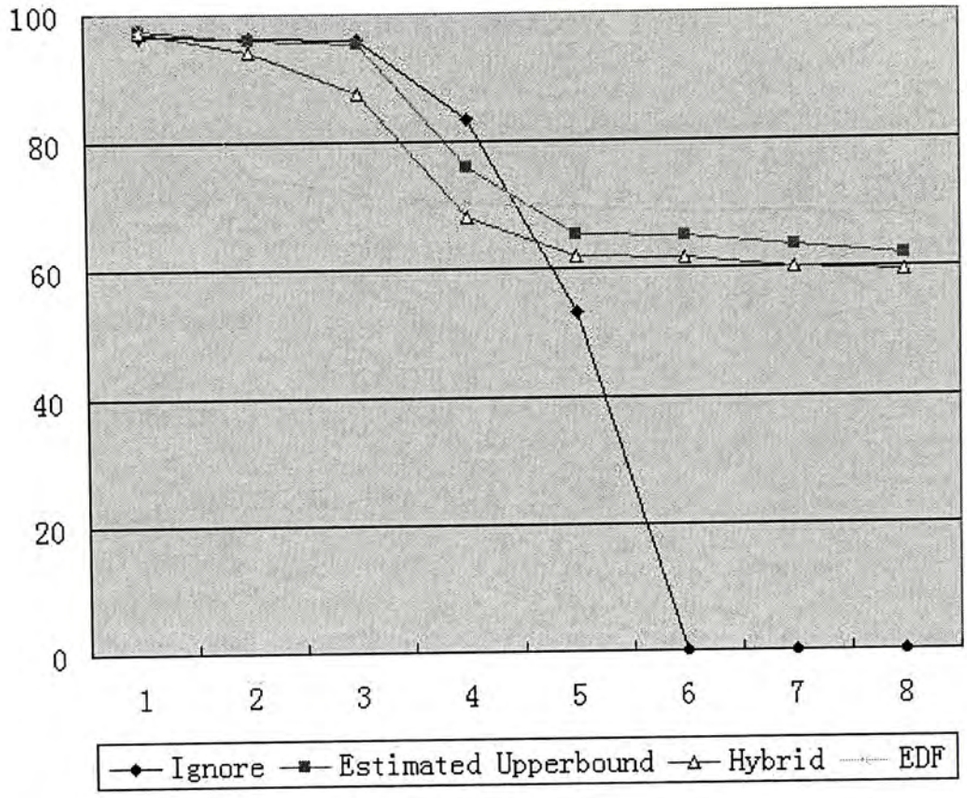

Figure 5.3: Hybrid Mechanism v.s. Individual Algorithms

most noticeably that of the EDF algorithm as expected. It is important to note that the curve of the hybrid algorithm stays very close to that of the estimated theoretical upper bound, which is in turn very close to the curve of the advanced algorithm when $f_{overload} \leq 4.9$. The *Ignore* algorithm begins to break down when $f_{overload} = 4.9$, while our hybrid algorithm continues to perform at only slightly sub-optimal level of quality. The hybrid algorithm is also always above the theoretical lower bound, except when $4.1 \leq f_{overload} \leq 4.6$ The hybrid algorithm performs way better than EDF when $f_{overload} \leq 3.8$, degrading only slowly after $f_{overload} = 3.8$. First, our theoretical analysis agrees with the experimental results. Second, our hybrid algorithm is robust and only slight sub-optimal, demonstrating a good time-quality tradeoff.

5.4.2 Effect of Average Execution Time

From (5.5) and (5.6), we can see these parameters which can affect the performance of the system are: t_{avgA}, and t_{avgG}, α, $c.r.a$, and $c.r.g$. We testify on these parameters in the following experiments. The test suites are based on the following characteristics.

- Average execution time: 500ms, 750ms, 1000ms.
- Deadline: 4.
- Action number: 8.
- Action relations: all.
- Overload factor: 1 to 8.

Figure 5.4 illustrates how different t_{avg} affect the system performance. In this result, we assume that $t_{avgA} = t_{avgG} = t_{avg}$ and use three task lists with the same number of tasks and same arrival time, but different performing time to test our system. Form (5.6), we can see if the performing time is longer, more task can be scheduled by the scheduler worker. It also means we can acquire a better performance. In the case $t_{avgA} \neq t_{avgG}$, the relation between average time and performance can be determined from (5.5). If t_{avgA} increases, more tasks can be scheduled, and the quality of the result also increases. So does t_{avgG}.

5.4.3 Effect of the Greedy Algorithm

We use the following test suites to testify the effect of $c.r.g$.

- Average execution time: 500ms.
- Deadline: 4.

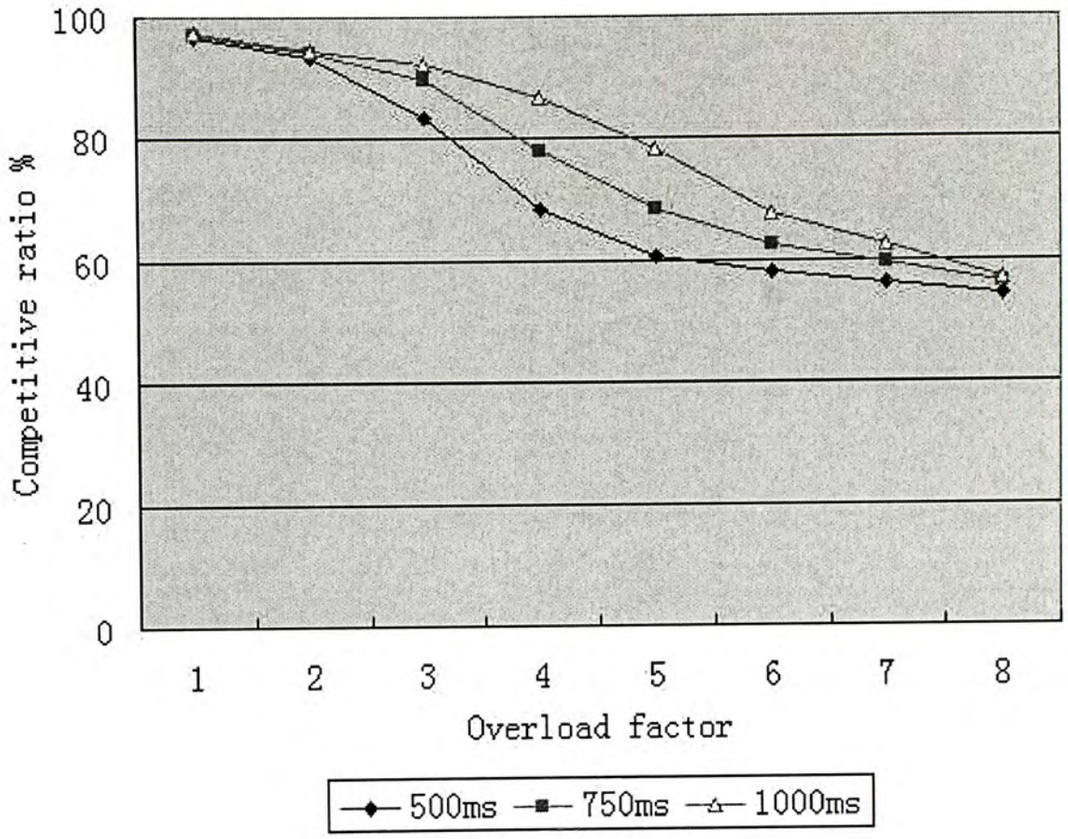

Figure 5.4: Hybrid Mechanism with Different t_{avg}

- Action number: 8.
- Action relations: all.
- Overload factor: 1 to 8.

Figure 5.5 shows shows how different $c.r.g$'s affect the result. In this example, we choose three different greedy algorithms: *FIFO*, *EDF*, and *HPF* (highest-profit-first, choose the task with highest profit-time ratio to perform) to combine with the same *Ignore* advanced algorithm. The advanced algorithm is again an *Ignore* algorithm. From the graph, we can see when the system is not overloaded too much, which means the advanced algorithm still work in time, the results are similar. When the system is heavily loaded and the advanced algorithm cannot give results in time, the hybrid algorithm with a better greedy algorithm gives better results. HPF is the best since it optimizes the profit, hence also the

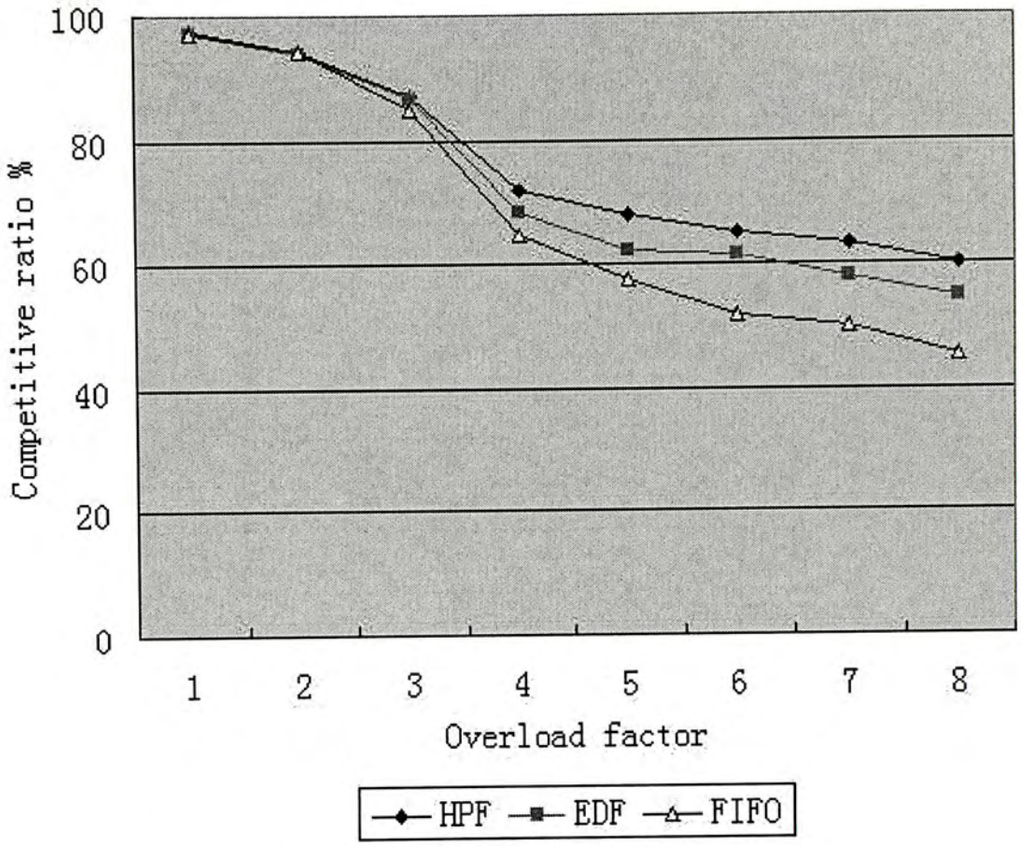

Figure 5.5: Hybrid Mechanism with Different *c.r.g*

competitive ratio.

5.4.4 Effect of the Advanced Algorithm

The test suites we used to testify the effect of *c.r.a* are based on the following characteristics.

- Average execution time: 500ms.
- Deadline: 4.
- Action number: 8.
- Action relations: all.
- Overload factor: 1 to 8.

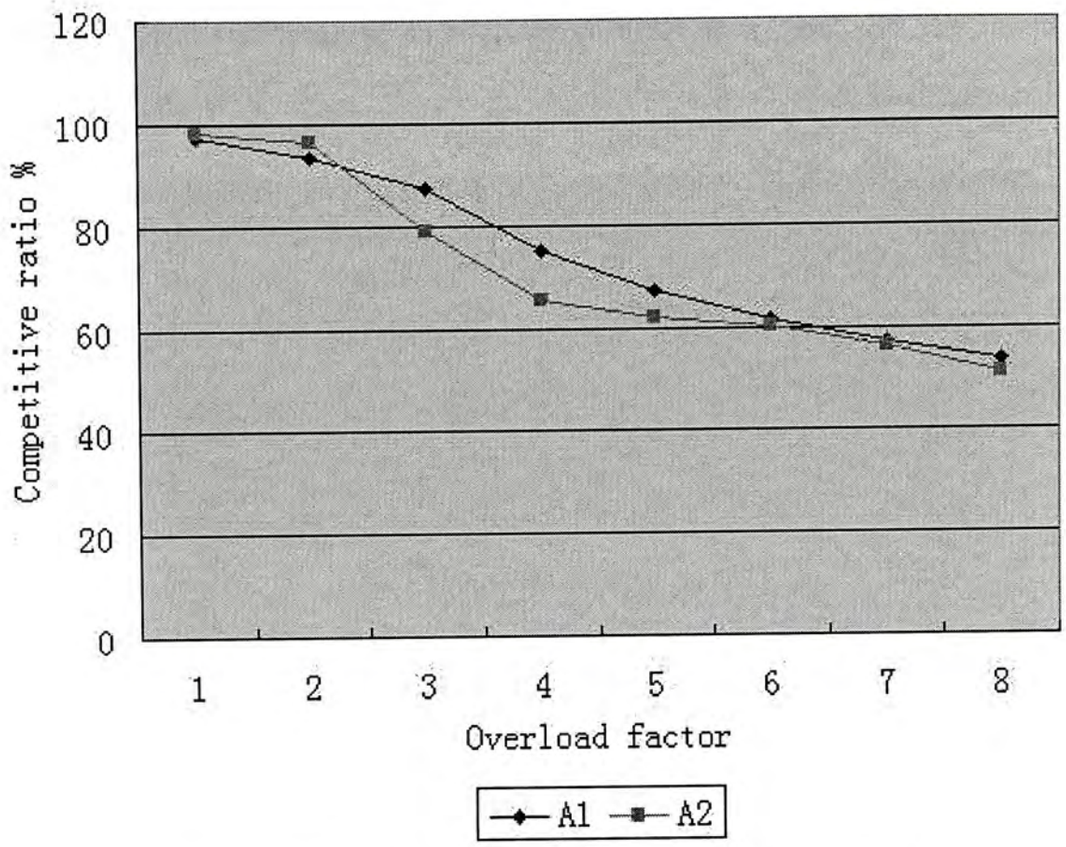

Figure 5.6: Hybrid Mechanism with Different *c.r.a*

We compare also the effect of different advanced algorithms. Figure 5.6 illustrates how different *c.r.a*'s affect the performance. Here we combine EDF with two different advanced algorithms respectively. The first algorithm A_1 is of order $O(n^2)$, while the other A_2 is of order $O(n!)$. Given enough time, A_2 has a higher *c.r.a* than that of A_2, but then a higher complexity also implies a higher α for A_2. As we can see from the graph, the hybrid algorithm with A_2 produces better results initially when $f_{overload}$ is small. As $f_{overload}$ grows, the performance of A_2 degrades more rapidly than that of A_1 since A_2 fails to respond in time.

5.4.5 Effect of Actions and Relations Among Them

Here we show how various actions and relations affect the system performance. Our theoretical analysis is a simplified analysis which does not include all possible

parameters. We testify these unadopted parameters in this and the following sections.

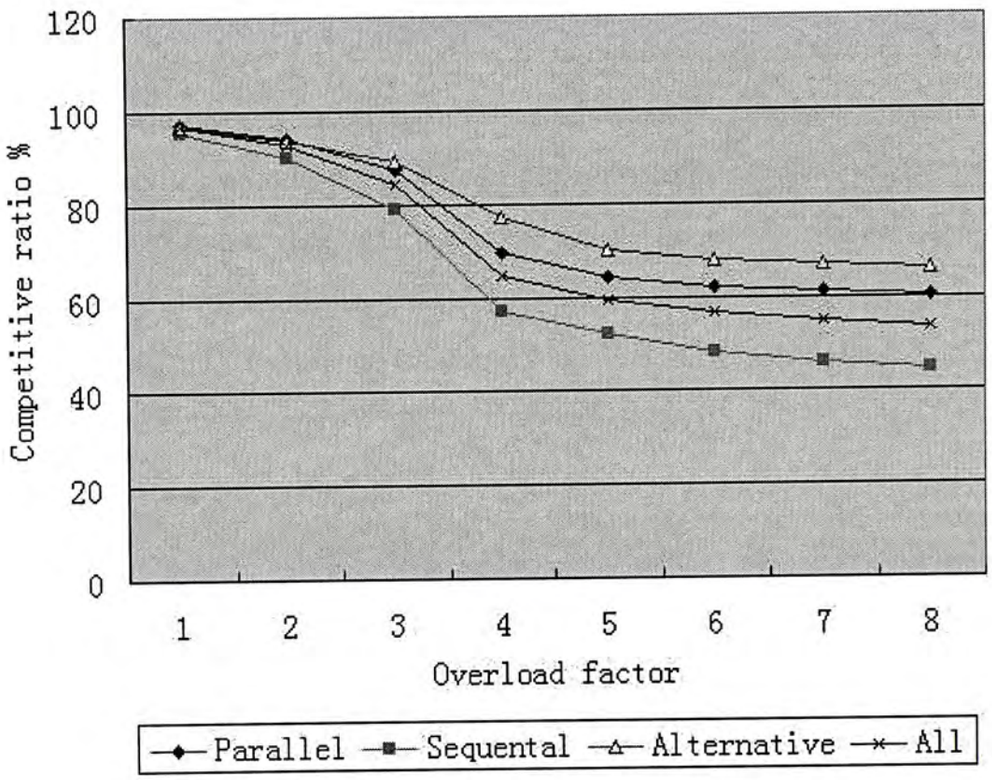

Figure 5.7: Different Relations Among Actions

Figure 5.7 illustrates the effect of the relations among actions. The test suites we used are based on the following parameters.

- Average execution time: 500ms.
- Deadline: 4.
- Action number: 8.
- Action relations: parallel, sequential, alternative, or all the three relations.
- Overload factor: 1 to 8.

One task list only has parallel relation among actions, another has all three relations generated randomly. The result shows that the more complicated the

tasks are, the more time is needed in scheduling and performing them. The performance is lower when the task list has different kinds of relations among actions. As we can see from the result, the difference is not obvious when $f_{overload}$ is small. As $f_{overload}$ grows, the difference becomes more obvious since more time is spent on scheduling actions with complicated relations. The sequential relation is the most complicated relation in scheduling, since actions in the same task only can be performed one by one. On the other hand, the alternative relation is the simplest relation in scheduling, especially when $f_{overload}$ is large. It is because that only one action needs to be executed in a task. The parallel relation is also simple than the combination of all the three relations.

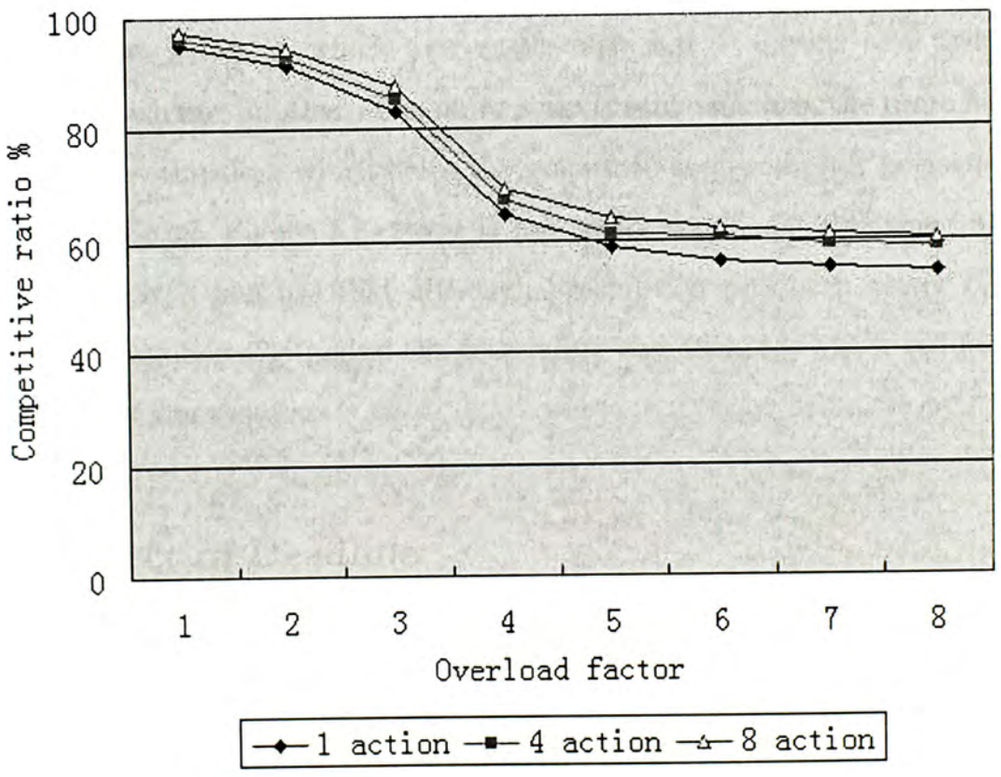

Figure 5.8: Different Action Number per Task

Figure 5.8 illustrates how different action number per task affect the performance. The test suites are like the following. The total number of actions in these test suites are the same. The suite with 1 action per task has 8 times more

tasks than the suite with 8 action per task.

- Average execution time: 500ms.
- Deadline: 4.
- Action number: 1 action per task, 4 actions per task, and 8 actions per task.
- Action relations: parallel relations.
- Overload factor: 1 to 8.

The relations among actions are parallel relation, since we compare these suites with test suites in which every task only has 1 actions and tasks are independent with one another. The more actions each task has, the more flexible it is in doing preemption, which helps the system to acquire better performance. As we can see from Figure 5.8, there is not really significant difference among these results. It is because that although preemption produces better results, larger action number also makes the scheduling complicated, which reduces the performance of the system.

5.4.6 Effect of Deadline

Figure 5.9 illustrates the effect of different deadlines. In this test, we use task suites with parameters as the following.

- Average execution time: 500ms.
- Deadline: 2, 4, 8.
- Action number: 8 actions per task.
- Action relations: All.

- Overload factor: 1 to 8.

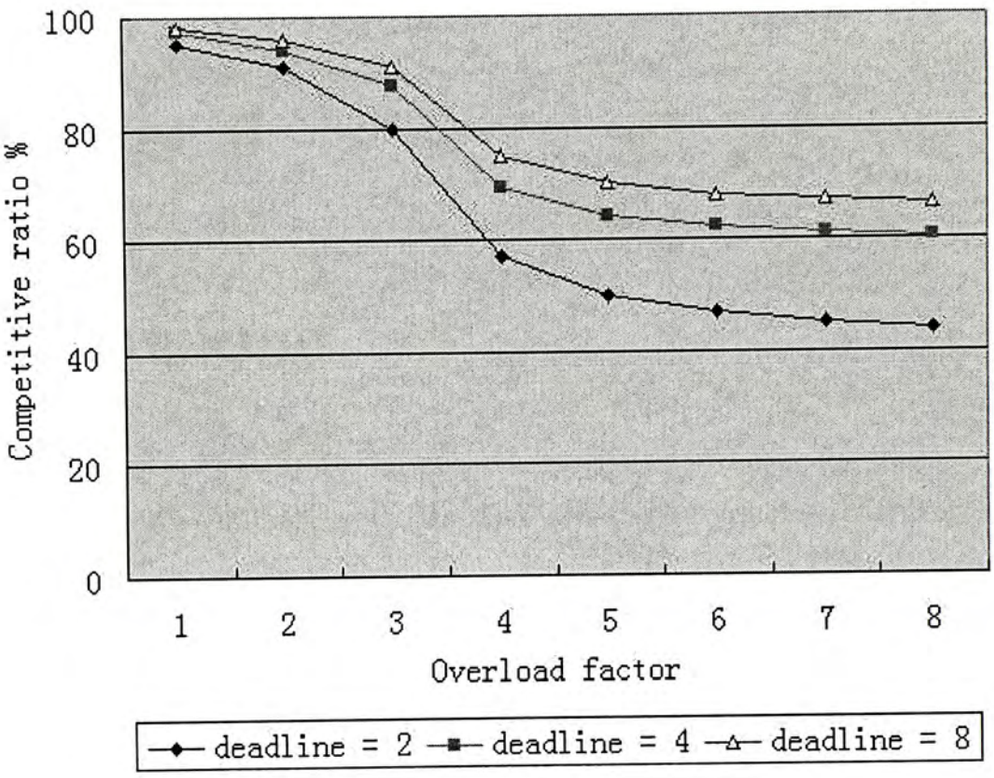

Figure 5.9: Effect of Different Deadline

The later the deadline is, the more time the system has to schedule and perform tasks. As we can see from the result, as the overload factor grows, the effect of deadline becomes more and more obvious. The deadline also determines if a task can be successfully finished (in our assumption, only successfully finished tasks can be counted in the system performance). A task with early deadline is likely to be not finished on time, especially when the overload factor is large. On the other hand, the effect of deadline decreases as the deadline becomes later.

Chapter 6

Conclusions

In this chapter, we summarize our contributions and discuss possible future work.

6.1 Summary of Contributions

Our real-time agent architecture contains a set of administrators and workers. These components rely on specially designed communication primitives to maintain inter-process communication and synchronization. The details of knowledge are hidden in individual processes, which communicate via a well-defined message interface. For example, if an agent wants to send a message to another agent, the cognition subsystem only needs to know the identifier of the recipient. The cognition worker do not need to know where the recipient is or how to send a message to it. This knowledge is maintained by the particular executor worker which will perform this task. Thus we can modify a component without changing another component, as long as the original functionality and communication interface are retained.

Our agent architecture has high flexibility. By changing the cognition methods in the cognition subsystem, we can realize different kinds of real-time agents. Since every component has its fixed function, it is possible to generate an agent

from a set of rules and data structures automatically. What we have essentially designed is a template for real-time agents. By instantiating the components with different algorithms and data, such as the scheduling algorithms, we can get real-time agents of particular characteristics.

We have introduced other approaches towards real-time agents in Chapter 2. The subsumption architecture consists of parallel layers which higher layer can subsume lower layer functions. User can change the behavior of the system by adding a new layer at the top without changing other layers. The subsumption architecture has a shortcoming: inflexibility. It does not allow users to rewrite a lower layer with out changing other layers. In our architecture, every component can be rewritten without affecting other components.

The 3T architecture and the InterRAP architecture are similar. Both of them use three layers: one layer for reaction, one layer for reasoning, and one layer for cooperation or higher level reasoning. They can be used to design agents which have both intelligence and reactivity. Their disadvantages are also similar. First, both of them are designed as robot control system, not for software real-time agents. Second, none of them provides task scheduling mechanisms, this lack will become more apparent when they work in heavy overloaded environments. Our architecture meets these problems well.

We also study the task scheduling mechanism in our real-time agent in details. The task scheduling problem in our architecture is similar to *on-line open shop scheduling* problem [12, 11]. In other words, the on-line open shop scheduling problem is a special case of our scheduling problem (if we define our agent has 8 executor workers, each task in the task list also has 8 actions in sequential relation, and these 8 action requires different executor workers one another, then this scheduling problem is equivalent to open shop scheduling problem with 8 machines). As far as we know, only a few approaches has been done about open shop scheduling with more than two machines [20]. We adopt the real-time AI

multiple method approach and combine two different scheduling algorithms: a greedy scheduling algorithm used in the task administrator and an advanced algorithm used in the scheduler worker. We give a scheduling model of the architecture, and present a theoretical analysis on the bounds of the competitive ratio of the proposed hybrid algorithm.

A simulated system is implemented to test the task scheduling mechanism. The validity of the analysis is verified empirically. Experimental results also confirm the robustness, efficiency, and quality of the proposed algorithm.

In summary, the contribution of this thesis is an architecture for real-time agents which can work well in real-time environments, even in heavy overload state. This architecture has high flexibility, and provides a pluggable template which supports to be initialized with different algorithms and data structures to acquire real-time agents with particular characteristics. Since we use a hybrid on-line task scheduling mechanism in our architecture, it works well under various conditions. This mechanism has been implemented in a test system which we have presented and evaluated in this thesis. We expect that this work should help us to understand problems in the design and implementation of real-time software agent systems. It is our hope that the research presented in this thesis will contribute to the goal of constructing robust, flexible, and efficient softwares for future's decentralized, large-scale computer-controlled or intelligent systems.

6.2 Future Work

Our research has provided a promising approach towards real-time software agents and on-line task scheduling issues, but more work around this topic still exists. Here we provide a list of possible extensions and future research topics.

- **Real-time agent demonstrations**

The real-time arcade game we described in Section 3.4 is simple. To verify our agent architecture, we need to apply it in more complicated real time demonstrations, such as *RoboCup Soccer* [39]. The real-time requirements in this application are more demanding than the arcade game we implemented. It will be challenging to evaluate whether our agent architecture actually performs well in this scenario or not.

- **Agent builder platform**

  The high flexibility of our architecture makes it is possible to build a real-time agent builder platform based on our architecture. As we introduce in this thesis, our architecture consists of a set of processes with fixed functions. Given a set of predefined rules and data structures, an agent can be generated automatically. Our goal is to design a platform which allows developers to quickly implement software agents and agent-based applications such as *AgentBuilder* [61].

- **More performance analysis**

  The theoretical analysis in this thesis only includes the most important parameters in the task scheduling problem. Parameters such as the action numbers per task, the relations among actions, the deadline distribution, task arriving distribution are not considered in the estimated bounds. Although the experimental results illuminate that these parameters may not be as important as the parameters we have discussed, they still affect the performance of the system to a certain extent. In the future, it is necessary to give a more detailed theoretical analysis which include these parameters and explain how these parameters affect the performance.

Bibliography

[1] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *In Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, 1987.

[2] J.L. Austin, editor. *How To Do Things With Words.* Oxford University Press, 1962.

[3] R.P. Bonasso, D. Kortenkamp, D. Miller, and M. Slack. Experiments with an architecture for intelligent, reactiveagents. *Intelligent Agents II, Lecture Notes in ArtificialIntelligence*, pages 187–202, 1995.

[4] A.H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann, 1988.

[5] M. E. Bratman, editor. *Intentions, Plans, and Practical Reason.* Harvard University Press, 1987.

[6] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[7] R.A. Brooks. Intelligence without reason. In Ray Myopoulos, John; Reiter, editor, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann.

[8] H. Burxhert and J. Muller. Ratman: Rational agents testbed for multi-agent networks. In *Decentralized A.I. 3 - Proceedings of MAAMAW'91*, pages 243–257, 1991.

[9] M.R. Carey and D.S. Johnson, editors. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W.H. Freeman and Co., San Francisco, Calif., 1979.

[10] C. Castelfranchi. Guarantees for autonomy in cognitive agent architecture. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: ECAI-94 Workshop on Agents Theories, Architectures, and Languages*, pages 56–70, Berlin, 1995. Springer-Verlag.

[11] B. Chen, P.A. Vestjens, and G.J. Woeginger. On-line scheduling of two-machine open shops where jobs arrive over time. *Journal of Combinatorial Optimization*, 1(4):355–365, 1998.

[12] B. Chen and G.J. Woeginger. A study of on-line scheduling two-stage shops. In D.Z. Du and P.M. Pardalos, editors, *Minimax and Applications*, pages 97–107. Kluwer Academic Publishers, 1995.

[13] S.E. Conry, K. Kuwabara, V.R. Lesser, and R.A. Meyer. Multistage negotiation for distributed constraint satisfaction. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1462–1477, 1991.

[14] V.G. Dabija. *Deciding Whether to Plan to React.* PhD thesis, Stanford University, Department of Computer Science, 1993.

[15] B. D'Ambrosio. Resource bounded-agents in an uncertain world. In *Proceedings of the Workshop on Real-Time Artificial Intelligence Problems (IJCAI-89, Detroit)*, 1989.

[16] R. Davis and R.G. Smith. Negotiation as a metaphor for distributed problem solving. In A.H. Bond L. Gasser, editor, *Readings in Distributed Artificial*

Intelligence, pages 333–356, San Mateo, CA, 1988. Morgan Kaufman Publishers.

[17] M. Drummond and J. Bresina. Anytime synthetic projection: Maximizing probability of goal satisfaction. In *Proc. of 8th National Conference on Artificial Intelligence (AAAI 90)*, pages 138–144, Boston, MA., 1990. AAAI Press / MIT Press.

[18] C. Elsaesser and M. Slack. Integrating deliberative planning in a robot architecture. In *the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS '94)*, pages 782–787, Houston, TX, 1994.

[19] I.A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents.* PhD thesis, UK, 1992.

[20] A. Fiat and G.J. Woeginger. *Online Algorithms: the State of Art.* Springer, 1998.

[21] T. Finin and R. Fritzson. Kqml - a language and protocol for knowledge and information exchange. In *the 13th International Distributed Artificial Intelligence Workshop*, pages 127–136, Seattle, WA, USA, 1994.

[22] R.J. Firby. Adaptive execution in complex dynamic worlds. Technical Report RR-672, 1989.

[23] R.J. Firby. Building symbolic primitives with continuous control routines. In *Proc. of the First Int. Conf. on AI Planning Systems*, pages 62–29, College Park, MD, 1992.

[24] K. Fischer, J.P. Muller, M. Pischel, and D. Schier. A model for cooperative transportation scheduling. In *Proceedings of the First International Conference on Multiagent Systems.*, pages 109–116, Menlo park, California, 1995. AAAI Press / MIT Press.

[25] A. Garvey and V. Lesser. Design to time real-time scheduling. In *COINS Technical Report 91-72, University of Massachusetts*, 1991.

[26] A. Garvey and V. Lesser. A survey of research in deliberative real-time artificial intelligence. In *Journal of Real-Time Systems*, 1994.

[27] L. Gasser and M.N. Huhns. Distributed artificial intelligence. In *Research Notes in Artificial Intelligence*, San Mateo, CA, 1989. Morgan Kaufmann.

[28] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *AAAI-92*, pages 809–815, 1992.

[29] M.R. Genesereth and R.E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, 1992.

[30] M.R. Genesereth and S.P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 1994.

[31] W.M. Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software-Practice and Experience*, 11:435–466, 1981.

[32] M. Grotschel, S.O. Krumke, J. Rambau, T. Winter, and U. Zimmermann. Combinatorial online optimization in real time. In Martin Grotschel, Sven O. Krumke, and Jörg Rambau, editors, *Online Optimization of Large Scale Systems—Collection of Results in the DFG-Schwerpunktprogramm Echtzeit-Optimierung groser Systeme (803 pages)*. Springer, 2001.

[33] J.Y. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379, 1992.

[34] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Worshop on Micro-Kernels & Other Kernel Architectures*, Seattle, U.S.A., April 1992.

[35] E. Hodys. A scheduling algorithm for a real-time multi-agent system. In *M.PHIL Thesis, Deaprtment of Computer Science, University of Rhode Island*, 2000.

[36] E. Huber and D. Kortenkamp. Using stereo vision to pursue moving agent with a mobile robot. In *Proceeding of IEEE International Conference on Robotics and Automation*, pages 2340–2346, 1995.

[37] M.N. Huhns and M.P. Singh. *Readings In Agents.* Morgan Kauffman Publishers, Inc., SanFrancisco, 1998.

[38] L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. In P. Maes, editor, *Designing autonomous agents*, pages 35–48, Cambridge, MA, 1990. MIT Press.

[39] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara. Robocup: A challenge problem for ai and robotics. In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I, volume 1395 of Lecture Notes in Artificial Intelligence*, pages 1–19. Springer-Verlag, Berlin, Heidelberg, New York, 1998.

[40] M. Klein. Supporting conflict resolution in cooperative design systems. *IEEE Transactions on Systems, Man, and Cybernetics (Special Section on DAI)*, 21(6), / 1991.

[41] S. Kraus, K.P. Sycara, and A. Evenchik. Reaching agreements through argumentation: A logical model and implementation. *Artificial Intelligence*, 104(1-2):1–69, 1998.

[42] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[43] J.W.S. Liu, editor. *Real-Time Systems.* Prentice-Hall, 2000.

[44] QNX Software Systems Ltd. *QNX Operating System: System Architecture.* 1993.

[45] D.M. Lyons and A.J. Hendriks. A practical approach to integrating reaction and deliberation. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, pages 153–162, 1992.

[46] P. Maes, editor. *Designing Autonomous Agents.* The MIT Press, 1990.

[47] F.V. Martial. Interactions among autonomous planning agents. In *Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 105–119, 1989.

[48] D.P. Miller and M.G. Slack. Increasing access with a low-cost robotic wheelchair. In *Proceedings of IROS'94*, pages 1663–1667, 1994.

[49] J.P. Muller, editor. *The Design of Intelligent Agents: A Layered Approach.* (LNAI Volume 1177). Springer-Verlag: Berlin, Germany, 1997.

[50] H. Nakashima and I. Noda. Dynamic subsumption architecture for programming intelligent agents. In *Proceedings of the International Conference on Multi-Agent Systems*, pages 190 – 197. AAAI Press, 1998.

[51] U. Neisser. *Cognition and Reality: Principles and Implications of Cognitive Psychology.* W.H. Freeman, 1976.

[52] A. Newell and H. Simon. Computer science as empirical enquiry: Symbols and search. *Communications of the Association for Computing Machinery*, 19:113–126, 1976.

[53] S. Phillips and J. Westbrook. On-line algorithms: Competitive analysis and beyond. In *Algorithms and Theory of Computation Handbook, CRC Press, 1999*. 1999.

[54] A.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[55] A.S. Rao and M.P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[56] J.S. Rosenschein. Rational interaction: Cooperation among intelligent agents. Phd thesis, Computer Science Department, Stanford University, 1985.

[57] J.S. Rosenschein and G. Zlotkin. *Rules of Encounter - Designing Conventions for Automated Negotiation Among Computers.* MIT Press, 1994.

[58] J.R. Searle. *Speech Acts.* Cambridge University Press, New York, 1969.

[59] A. Sloman and R. Poli. SIM AGENT: A toolkit for exploring agent designs. *Intelligent Agents Vol. II (ATAL-95)*, pages 392–407, 1996.

[60] A.S. Tanenbaum. *Distributed Operating System.* Prentice-Hall, Inc., New Jersey, 1995.

[61] A. Web. Agentbuilder - an integrated toolkit for constructing intelligence software agents, 1999.

[62] G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* The MIT Press, 1999.

[63] D.E. Wilkins, K.L. Myers, J.D. Lowrance, and L.P. Wesley. Planning and reacting in uncertain dynamic environments. In *Journal of Experimental an Theoretical AI, 7*, 1995.

[64] C. Wong, D. Kortenkamp, and M. Speich. A mobile robot that recognizes people. In *IEEE International Conference on Tools and Artificial intelligence*, 1995.

[65] M.J. Wooldridge. *On the Logical Modelling of Computational Multi-Agent Systems.* PhD thesis, Manchester, UK, 1992.

[66] M.J. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. In *Knowledge Engineering Review*, 1995.

[67] S. Yu, M. Slack, and D. Miller. A streamlined software environment for situated skills. In *the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS'94)*, 1994.

[68] G. Zlotkin and J.S. Rosenschein. A domain theory for task oriented negotiation. In Ruzena Bajcsy, editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 416–422, San Mateo, California, 1993. Morgan Kaufmann.