A Real-Time Agent Architecture: Design, Implementation and Evaluation

J.H.M. Lee and L. Zhao

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong SAR, China
{jlee,lzhao}@cse.cuhk.edu.hk

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. 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.

1 Introduction

The task at hand is that of the design and implementation of real-time agents, such as those used in military training 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. Besides sharing all common characteristics of intelligent agents, real-time agents should possess also specific features for survival in real-time environments:

- Automation: Real-time agents are autonomous. It means we can realize real-time software agent with separate processes/threads.
- Reaction: Agents must be able to react to different events, expected or not. The more urgent a situation is, the more quickly the agent should respond to it.
- 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 [4].

- 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 cannot process all data in time (data glut). Agents must be able to select useful data (or data which agents think useful) from received data. Unprocessed data can remain in buffer, and can be flushed by new arriving data.

In this paper, we develop a real-time agent architecture from Ulric Neisser's human cognition model [12]. 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 [2], 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 paper is organized as following. In Section 2, we motivate and introduce the logical design of our real-time agent architecture. Section 3 explains the physical realization of our architecture on the QNX real-time platform. We also present a brief account of a multi-agent implementation of a real-time arcade game. Section 4 describes in details the task scheduling mechanism, theoretical analysis and experimental results, followed by related work in Section 5. Last but not least, Section 6 gives concluding remarks and shed light on possible direction of further work.

2 Logical Architecture

Neisser [12] views human cognition as a perpetual process, which keeps working as long as we are awake. Figure 1 illustrates different parts and their relations in human cognition. In this model, human acquires samples by exploring outer environment (Ex-

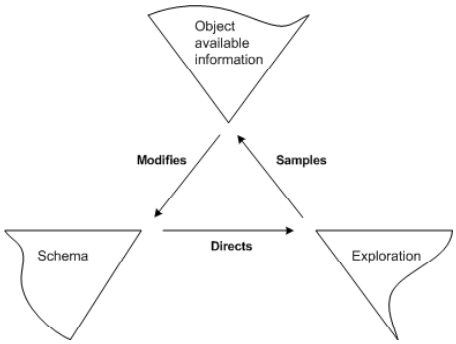

Fig. 1. The Perpetual Cycle

ploration). These samples bring useful information of the world (Object available information). By modifying the information, human makes decisions and plans (Schema), which guide us 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 since human is the best example of a real-time agent. In our architecture, a real-time agent 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. Figure 2 gives the overall structure and detailed implementation of our architecture.

2.1 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 [9]. 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.

2.2 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 [2], the

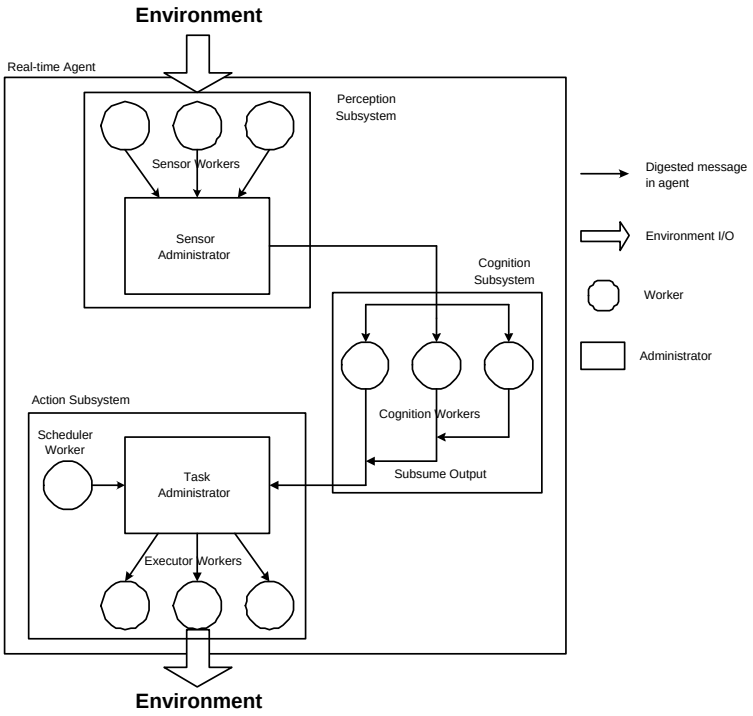

Fig. 2. Real-Time Agent Architecture

dynamic subsumption architecture [11], 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) [13].

2.3 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.

3 Physical Architecture

We have given a logical architecture of real-time agents. This architecture is composed of three collaborating subsystems, which can be implemented naturally as concurrently

running processes. While these subsystems have individual responsibilities and goals, they must cooperate to act as a collective whole. A good inter-process communication 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 form of message passing, which satisfies the above criteria, before giving a process structure design of an implementation of our real-time agent architecture. We conclude this section by presenting a brief account of a mult-agent implementation of the "Space Invader" real-time game.

3.1 Message Passing

Gentleman [5] designs a set of message passing primitives 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. This philosophy is subsequently adopted in the construction of the commercial real-time OS, QNX [7], which has been deployed in numerous mission critical and embedded system. There are three primitives:

- *Send()*: for sending messages from a sender process to other processes.
- *Receive()*: for receiving messages from other processes.
- *Reply()*: for replying to processes that have sent messages.

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.

When process A sends a message to process B, the following steps take place:

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.*
   (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.

Gentleman's message passing primitives enable us to define two kind of processes: *administrators* and *workers*. An administrator owns one or more workers. Administrator stores a set of jobs and workers perform them. Once a worker finishes a job, it sends a request to its administrator. Upon receiving the request, the administrator replies to the worker with a new job assignment. Administrators do only two thing repeatedly: receive task and job requests, and reply to workers with job assignments. Thus administrators are never blocked, since they never issue *Send()* messages, allowing administrators to attend 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 job allocations to her inferiors. On the other hand, lowly workers can only be either performing job duties or waiting for new assignments.

3.2 Overall Physical Architecture

We outline an implementation of our architecture on the QNX platform. 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 2 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.

3.3 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 preprocesses 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.

3.4 The Cognition Workers

The cognition workers are responsible for mapping events to tasks. Suppose we want to adopt Brooks's subsumption architecture [3] 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.

3.5 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.

3.6 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 illustrates the system 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.

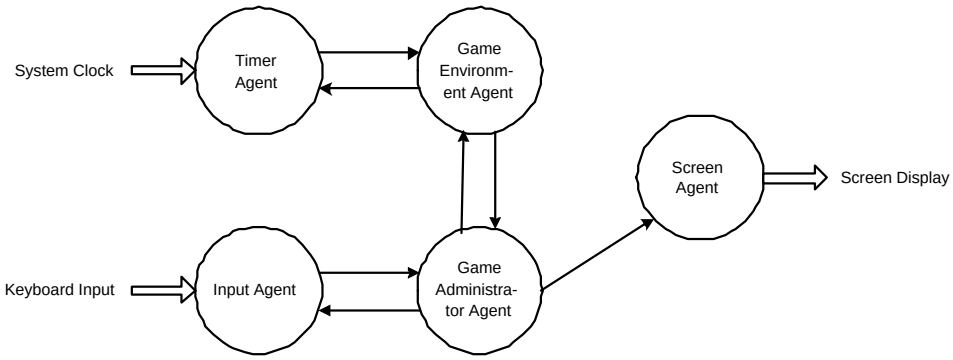

Fig. 3. Architecture of the Demonstration Game

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. For example, 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.*

Such a set 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.

4 Task Scheduling in Real-time Agents

As we have introduced in Section 3, we combine two different on-line scheduling algorithms in the action subsystem to schedule tasks. The greedy scheduling algorithm, usually simple and fast, used in the task administrator 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 scheduler worker opts for solution quality. An example is local search algorithm 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.

The task administrator maintains a task queue for undispatched tasks. Once a new task arrives, the administrator runs the greedy algorithm to insert this task into proper position of task queue, while preserving the results last sent by the scheduler worker. Once an executor worker finishes a task, the task administrator first checks if the result given by scheduler worker has any job for this executor. If there is no such task, the administrator runs the greedy algorithm to find the next task.

The scheduler worker maintains a task queue which is synchronized with the task queue in the task administrator. Once the scheduler worker finishes scheduling, it sends the scheduling result to the task administrator, which in turn replies with newly arrived tasks and other information. The scheduler worker uses these information to update its task queue, and runs the advanced algorithm again.

If tasks arrive sparsely, the scheduler worker would have enough time to complete executing its advanced algorithm, producing good quality results. Even if scheduler worker cannot return a result on time, the greedy algorithm in the task administrator can still provide substitute service temporarily. Therefore, the continuous functionality of the agent will not be disrupted.

4.1 Theoretical Analysis

We can model this combined 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 field is the executor worker set.

Task list T is composed of a sequence of tasks $T = (T_1, T_2, T_3, \ldots, T_n)$, stored stored in the task queue in the task administrator and the scheduler worker. Both the scheduler worker and the task administrator run scheduling algorithms to select tasks from T. We further partition T into T_G and T_A, where $T_G = T_{G_1}, T_{G_2}, \ldots, T_{G_{S_G}}$ is the task list selected by the greedy scheduling algorithm, and $T_A = T_{A_1}, T_{A_2}, \ldots, T_{A_{S_A}}$ is the task list selected by the advanced algorithm.

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. 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 ration*, which is a ratio between the off-line optimal solution and the on-line solution:

$$r = \frac{\text{Profit of On-line Algorithm Solution}}{\text{Profit of Optimal Solution}}$$

In our scheduling mechanism, we assume that the competitive ratio of the greedy algorithm and the advanced algorithm is $c.r.g$ and $c.r.a$ respectively. For task T_i, we

use p_i to denote the *profit* of T_i. There can be various notion of profits. Here, we are interested in the weighted (by priority) percentage of the tasks that can be completed before their respective deadline. The target of any scheduling algorithm is to achieve as high a profit as as possible. We can then define the competitive ratio of our mechanism $c.r.$ as follows:

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

We only discuss this formula when the system is in overload state. In non-overload state, even a greedy scheduling algorithm can give optimal results. For example, the EDF algorithm is optimal in non-overload state [8]. In overload state, the performance of the mechanism is determined by two parameters: task scheduling time $t_{scheduling}$ and task performing time $t_{performing}$, where $t_{scheduling}$ is the time that the scheduler worker used to run the advanced algorithm to give a result and $t_{performing}$ is the time that the task administrator used to finish tasks given by scheduler worker. If $t_{scheduling} \leq t_{performing}$, then the task administrator needs not run the greedy algorithm. In this case, the result quality of the combined mechanism is the same as that of the advanced scheduling algorithm. In the following analysis, we consider only the case when $t_{scheduling} > t_{performing}$.

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 $t_{i_{exe}}$ to be the execution time of task T_i.

$$\sum_{i=1}^{S_A} t_{A_{i_{exe}}} + \sum_{j=1}^{S_G} t_{G_{j_{exe}}} = \text{Scheduling Time of } T_A \quad (2)$$

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 $T_{A_1}, T_{A_2}, \ldots, T_{AS_A}$, then we can estimate the upperbound of the scheduling time as $S_A \log S_A$. Let $\alpha = \log S_A$, so we have:

$$\text{Scheduling Time of } T_A \leq S_A \alpha \quad (3)$$

Similarly, we can define α for any given scheduling algorithm and task list. So (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 (2) and (3), we have:

$$S_G \cdot t_{avgG} + S_A \cdot t_{avgA} \leq S_A \alpha \rightarrow S_G \leq S_A \frac{\alpha - t_{avgA}}{t_{avgG}} \quad (4)$$

Combining (1) and (4), we get

$$\begin{aligned} c.r. = \frac{S_A \cdot c.r.a + S_G \cdot c.r.g}{S_A + S_G} &= \frac{S_A(c.r.a + c.r.g\frac{\alpha - t_{avgA}}{t_{avgG}})}{S_A(1+\frac{\alpha - t_{avgA}}{t_{avgG}})} \\ &\leq \frac{c.r.a - c.r.g}{1+\frac{\alpha - t_{avgA}}{t_{avgG}}} + c.r.g = c.r.a - \frac{c.r.a - c.r.g}{1+\frac{t_{avgG}}{\alpha - t_{avgA}}} \end{aligned} \tag{5}$$

From (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 $t_{avgA} = t_{avgG} = t_{avg}$.

$$\begin{aligned} c.r. \leq c.r.a - \frac{c.r.a - c.r.g}{1+\frac{t_{avgG}}{\alpha - t_{avgA}}} &= c.r.a - (c.r.a. - c.r.g)\frac{\alpha - t_{avg}}{\alpha} \\ &= c.r.g + (c.r.a. - c.r.g)\frac{t_{avg}}{\alpha} \end{aligned} \tag{6}$$

4.2 Experimental Results

We have implemented a simulation system to test the combined scheduling mechanism against our theoretical prediction. The simulation system uses a random process task generator to generate different task lists. The generated tasks are stored and sent to the task administrator during simulation. This system is implemented on QNX.

In following experiments, we choose EDF of order $O(n)$ as the greedy algorithm and a *Ignore* algorithm [6] of order $O(n^2)$ as the advanced algorithm. As we mentioned before, we consider only the cases when the system is in overload state. We define a *overload factor* $f_{overload}$ to measure how overloaded a system is. For example, if $f_{overload} = 2$, that means the system has twice the amount of tasks that the system can handle. The higher the overload factor is, the longer time the algorithm takes to schedule the tasks.

Table table:time gives the average scheduling time (in ms) for different states. We

Table 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

can see that although the advanced algorithm can give better quality scheduling results in general, its efficiency is worsen 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 algorithm works well within the timing constraint; so is the combined algorithm.

Figure 4 compares the competitive ratio of the combined mechanism against those of the individual algorithms. The estimated theoretical upperbound of the mechanism,

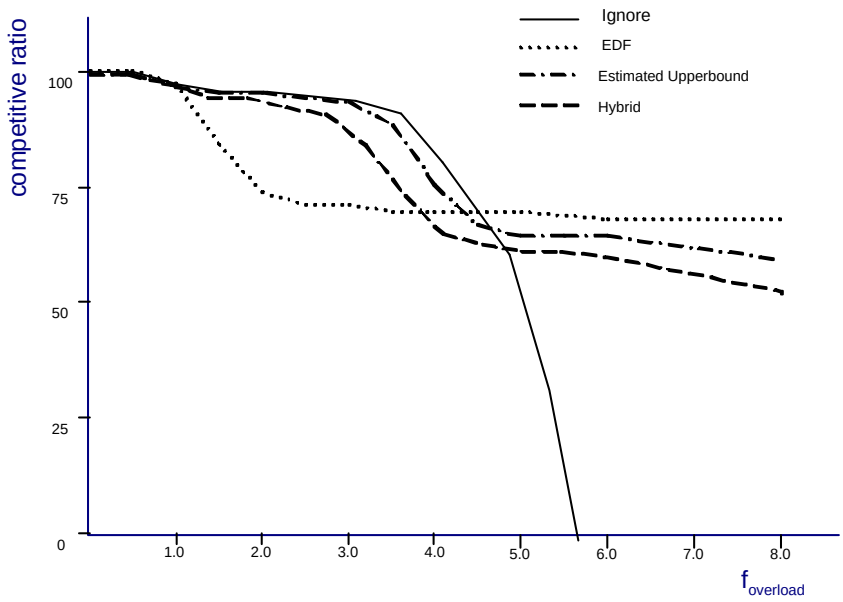

Fig. 4. Competitive ratio of different algorithms

according to (6), is also displayed in the figure.

When $f_{overload} \leq 1.0$, the system is in non-overload state, all algorithms are optimal. When $f_{overload} \geq 1.0$, the quality of the algorithms begins to drop, most noticeably that of the EDF algorithm as expected. When $f_{overload} \geq 3.8$, the performance of the combined mechanism is dropped under the performance of the greedy algorithm. When $f_{overload} \geq 4.9$, the performance of the advanced algorithm is lower than the combined mechanism and drops quickly. With the overload factor increases, the advanced algorithm cannot give any results in time. But even in this case, the performance of the combined mechanism is still similar with the greedy algorithm's advantage. It is important to note that the curve of the combined algorithm stays very close to that of the estimated theoretical upper bound, which is in turn very close to the curve of the *Ignore* algorithm. First, our theoretical analysis is fairly accurate. Second, the solution quality of the combined mechanism is very close to the sub-optimal returned by the *Ignore* algorithm.

5 Related Work

There have been other approaches towards real-time agents. Brooks's subsumption architecture [3] is composed of different layers in which higher layer can subsume lower layer functions. Lower layers have basic functions and higher layers have more human-like behavior. Through adding a new layer at the top, user can change the behavior of the system.

In the subsumption architecture, the relations among layers are fixed. The dynamic subsumption architecture [11], in which the layers are dynamically reconfigured during runtime, is more flexible than the original subsumption architecture.

Bonasso's 3T architecture [1] is another real-time robot control architecture. This architecture separates the general robot intelligence problem into three interacting layers or tiers (3T): a reactive skill layer, a sequencing layer, and a deliberation layer. 3T has a powerful planning mechanism, and ability to react to time-critical events.

Another real-time agent architecture is the InterRAP architecture described by Müller [10]. This architecture is a layered control architecture which is also composed of three layers: a behavior layer, a local planning layer, and a cooperative planning layer. The InterRAP architecture extends the planner-reactor architecture by adding a cooperation layer. The cooperative planning layer is specifically for multi-agent activities.

These various architectures generate tasks and actions, which are usually executed as soon as they are generated. Even when some architectures, such as InterRAP, rearrange the order of the tasks before execution, the main consideration is not on the satisfaction of timing constraints. In addition, each of the above architectures is designed for different real-time or robotic application with specific characteristic behavior. Our meta architecture can serve to enhance the responsiveness of these architectures, *especially in overload situations*, using our efficient hybrid on-line task scheduling algorithm. An important advantage is that our architecture works well in different real-time environments. The hybrid on-line scheduling mechanism guarantees our architecture's performance in different overload states. We can estimate the bounds of the performance through theoretical analysis.

6 Concluding Remarks

In summary, 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.

Another advantage of the architecture is flexibility. By changing the cognition methods in the cognition workers, 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 a particular kind of real-time agents.

We also study the task scheduling mechanism in our real-time agent in details. 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. Theoretical analysis and experimentation confirms that our combined mechanism inherits the best of both worlds: efficiency and good solution quality.

For future work, we suggest to try to give more performance analysis of different combination of scheduling algorithms and test them with different real world cases. It is also interesting to see construct a real-time agent builder platform based on our proposed architecture.

References

1. 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.
2. R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.
3. Rodney 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.
4. 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.
5. W. Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software-Practice and Experience*, 11:435–466, 1981.
6. 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.
7. 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.
8. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973.
9. Jane W.S. Liu, editor. *Real-Time Systems*. Prentice-Hall, 2000.
10. J.P. Muller. *The Design of Intelligent Agents: A Layered Approach*. (LNAI Volume 1177). Springer-Verlag: Berlin, Germany, 1997.
11. 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.
12. Ulric Neisser. *Cognition and Reality: Principles and Implications of Cognitive Psychology*. W.H. Freeman, 1976.
13. Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.