An Architecture For Multi-Agent Systems Operating In Soft Real-Time Environments With Unexpected Events

by

Christopher D. D. Micacchi

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Chris Micacchi

Abstract

In this thesis, we explore the topic of designing an architecture and processing algorithms for a multi-agent system, where agents need to address potential unexpected events in the environment, operating under soft real-time constraints. We first develop a classification of unexpected events into Opportunities, Barriers and Potential Causes of Failure, and outline the interaction required to support the allocation of tasks for these events. We then propose a hybrid architecture to provide for agent autonomy in the system, employing a central coordinating agent. Certain agents in the community operate autonomously, while others remain under the control of the coordinating agent. The coordinator is able to determine which agents should form teams to address unexpected events in a timely manner, and to oversee those agents as they perform their tasks. The proposed architecture avoids the overhead of negotiation amongst agent teams for the assignment of tasks, a benefit when operating under limited time and resource constraints. It also avoids the bottleneck of having one coordinating agent making all decisions before work can proceed in the community, by allowing some agents to work independently.

We illustrate the potential usefulness of the framework by describing an implementation of a simulator loosely based on that used for the RoboCup Rescue Simulation League contest. The implementation provides a set of simulated computers, each running a simple soft real-time operating system. On top of this basic simulation we implement the model described above and test it against two different search-and-rescue scenarios. From our experiments, we observe that our architecture is able to operate in dynamic and real-time environments, and can handle, in an appropriate and timely manner, any unexpected events that occur. We also comment on the value of our proposed approach for designing adjustable autonomy multi-agent systems and for specific environments such as robotics, where reducing the overall level of communication within the system is crucial.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Robin Cohen, for all the contributions and suggestions she has made, and for tirelessly working to help me finish this thesis. Without her there to push me along and keep me on track, I doubt I would have ever gotten to this point.

I would also like to thank Rob Szumlakowski for helping me determine the lists of goals and of unexpected events used in the simulation, and, along with Sean Munro, for pretending to be interested while I practised presenting my research. And of course, all of my friends who have helped me to complete this thesis, sometimes just by providing a much-needed distraction.

I am grateful to my mom and dad, my sister Lisa, and my brother Mike, as well as my grandparents, my nonna and nonno, and my many aunts, uncles, and cousins, all of whom have been very supportive of and interested in my research, some of whom have even endeavoured to read this thesis despite the technical jargon.

Also at Waterloo, I'd like to thank Richard Mann, who has always been willing to talk at length about anything, and Adam Milstein with his endless supply of Dilbert comics.

Finally, I'd like to thank my two readers, Peter van Beek and Chrysanne DiMarco, for their suggestions and corrections to this document.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Researchers in the field of artificial intelligence are designing agents as autonomous entities that solve problems on behalf of human users. In some cases, it is necessary for these agents to work together in what is referred to as a *multi-agent system*, to collectively solve the user's problem. One of the main challenges in designing an effective multi-agent system is determining the relationships between the various agents — that is, the architecture of the system. For example, one of the issues that must be decided is when to leave the agents fully autonomous within the system and when to have their actions directly controlled by some specially-designated controlling agent. Another issue is what kind of communication must be supported in order for tasks to be assigned to agents.

This challenge is made more complex when the system must operate in uncertain, dynamic environments where unexpected events may occur at any time. How unexpected events are handled is particularly important, because the system may need to take on new tasks as a result of these unexpected events. In fact, the system must be able to incorporate these new tasks into its current activities while still meeting its already existing commitments to previous tasks. Therefore, when specifying a multi-agent system that may encounter unexpected events, it is also especially important to correctly identify different classes of these events and to outline how each should be dealt with by the collection of agents.

Determining which agent performs a task and which agent assigns it that task must also be

determined for the system to correctly handle such events when they occur. Once more, this is a reflection of the autonomy of the agents in the society and the question of when that autonomy should be adjusted. For example, a system where the agents have minimal autonomy requires that a controlling agent perform more computation, but has the advantage of being able to more easily allocate the new tasks to agents, since the controlling agent already has access to all the information necessary to make such a determination.

If the group of agents is intended to work in an environment where the tasks that they must perform can have time constraints, or where the hardware the agents are embodied in, as well as the environment they must work in, can impose resource constraints (such as limited computing power or communications bandwidth), additional strategies must be integrated into the proposed architecture for the multi-agent system.

Moreover, any algorithms used in the system (for example, those used for communication, problem solving and planning, or task allocation) must be aware of these constraints and must be adjusted accordingly. Systems with these characteristics are generally called *real-time systems*. In particular, if the time or resource constraints imposed by tasks are very strict and cannot be violated, such a system is called a *hard real-time system*. However, if some of the constraints may be applied only loosely, or even completely ignored, the system is called a *soft real-time system*.

The aim of this thesis is to develop an architecture for multi-agent systems operating in soft real-time environments with unexpected events. Our focus is on how to use the structure of the system to affect the autonomy of the agents and how to best support communication within the system, to respect the limitations of time and resources.

1.2 Problem Definition

Consider the following scenario for a multi-agent system:

- There is a task/problem to be solved by the system of agents;
- Agents are able to reason about what they are doing;
- Agents are allowed to communicate with each other and with a human;

- The human provides the initial goal and specifications;
- Agents are allowed to sense their environment dynamically;
- Agents carry out actions and contribute to the overall task (the mission);
- There are real-time issues that need to be addressed, such as a deadline for mission or subtask completion;
- The environment the agents are working in can change unexpectedly.

In this research, we present a framework for designing such a multi-agent system. We discuss how the need to accommodate real-time constraints makes it valuable to support, within the same community of agents, some agents that are fully autonomous and some agents whose actions are specifically controlled. We then elaborate on the communication necessary to provide the appropriate level of autonomy for these agents, outlining the messages required between the agents and the controller. We discuss how this is especially important in scenarios where there are unexpected events in the environment, classifying these events into three main types, distinguishing when these events arise and, in each case, what kind of response is required.

While other researchers have looked at the problem of allocating tasks to agents in agent societies under unexpected events (e.g. [15]), these solutions have not additionally addressed the concern of operating in a soft real-time environment. Moreover, the proposed combination of fully autonomous and controlled agents that co-exist within the society is a novel feature of our design. This hybrid architecture contrasts with other models that are fully distributed or completely centralized.

In addition to developing a framework for designing effective multi-agent systems, we have designed an implementation to demonstrate the effectiveness of this model and have used this implementation to run different scenarios in a simplified variation of the RoboCup Search and Rescue simulation environment [20], to validate the usefulness of our proposed approach. The results of our tests also allow us to discuss the benefits of our architecture, in comparison with other related work.

Chapter 2 discusses in detail the requirements of a real-time multi-agent system, and describes background research related to this topic.

Chapter 3 presents the proposed model, with some brief examples.

Chapter 4 presents a more thorough illustration of the model with the use of examples.

Chapter 5 describes our implementation of the model in a software simulation environment.

Chapter 6 examines the results of experiments running the software simulation on various scenarios.

Chapter 7 discusses the value of the research in comparison with the related work, and records some topics for future work.

Chapter 8 presents the conclusions.

Appendix A provides additional documentation on the implementation, and includes information necessary to create agents that will operate with our implementation.

In summary, this thesis provides a useful starting point for the design of any multi-agent system that must address unexpected events in a time-sensitive manner. We feel our architecture will be especially valuable to applications such as robotics, since in this environment it is expensive to include a large amount of computing power on every agent, so that having one coordinator performing the more complex reasoning is desirable. In addition, our research suggests how to use the structure of the system to control the level of autonomy of the agents within the society and as such contributes to our understanding of the construction of adjustable autonomy multi-agent systems.

Chapter 2

Background

2.1 Introduction

To design an architecture for constructing multi-agent systems that can address unexpected events and real-time constraints, it is important to first clarify what we mean by the terms *agent*, *multi-agent system*, and *real-time constraints*. We then examine some existing research in the design of multi-agent systems, discussing briefly how solutions to other problems for multi-agent systems relate to our proposed design. Once we have presented our model in Chapter 3, we provide additional comparison with related work in Chapter 7.

2.2 Agents and Multi-Agent Systems

2.2.1 Agents

An *agent* is an autonomous software program capable of reasoning about and performing tasks and interacting with its environment [23]. Each agent has a *state*, which is simply the union of all the variables controllable and observable by the agent.

We define a *goal* to be the target state that the agent wishes to achieve. Agents achieve goals by formulating plans. The highest-level goal that an agent must achieve is its *mission*. The mission also includes any constraints or limitations on the agent's behaviour.

Term	Definition
Goal	A target state that the agent wishes to achieve. Agents achieve goals by formulating plans.
Mission	The highest-level goal that an agent must achieve. In addition, the mission also includes any constraints or limitations on the agent's behaviour.
Plan	The series of actions that the agent must take to get from its current state to the goal state.
Task	A goal and a plan, where the plan has been fully generated and is designed to accomplish the goal it is attached to.

Table 2.1: Summary of Terms and Definitions

A *plan* is the series of actions that the agent must take to get from its current state to the goal state.

2.2.2 Multi-Agent Systems

A *multi-agent system* (*MAS*) is a society of agents, possibly working together towards, or competing with each other for, a goal or goals [23].

Like agents, multi-agent systems can also have their own goals and plans. These goals and plans describe the motivations and actions of the society as a whole; agents within the society may still have their own independent goals and plans. However, when an agent is operating within a society, some issues commonly arise.

How do we assign tasks to the agents in the system?

Common solutions to this issue are to let individual agents make independent choices regarding when to accept new tasks, to hold auctions or votes to decide which agent is best suited to handle the task, and to designate an agent as a "controlling agent," responsible for assigning tasks to other agents [23, 14].

How do we coordinate agents that are executing different parts of the same plan?

For example, agents can individually negotiate with each other to coordinate their actions, agents can decide to form temporary "coalitions" to accomplish a task jointly, or another third-party agent can direct the interacting agents [23, 4].

How much autonomy does each agent have? That is, how much freedom do individual agents have to make their own decisions?

In particular, how much does an agent consider the goals and plans of the society as a whole when deciding what it should do? Common responses to this problem are to allow all agents to be fully autonomous, to have all agents directly controlled by an agent with a higher authority, or to have agents that can change their autonomy based on the circumstances [23, 10].

How much information does each agent need to send to other agents? How much information does each agent need to request from other agents?

Some solutions to the above problems require more communication between agents than others do. In some scenarios communication costs are essentially zero, and thus there is no price to pay for sending large messages between agents. In other circumstances communication costs are very high and therefore the system must justify every message it sends [23, 2].

2.3 Real-Time Systems

2.3.1 Overview

A *real-time system* [3] is a system in which tasks can have a deadline, by which the task must be completed. To deal with such deadlines, a real-time system includes a *scheduler*, responsible for executing tasks at their appropriate times. The list of all upcoming tasks, along with their times of execution, is called a *schedule*.

Different systems require different degrees of "softness" in their real-time components. Generally, systems where deadlines must be met precisely with no slipped or missed goals are called *hard real-time* systems, whereas systems in which it is permissible for deadlines to slip or be missed are called *soft real-time* systems. However, even in a soft real-time system, slipped deadlines can be problematic and the system generally makes a "best-effort" to meet its deadlines.

An example of a soft real-time system is a mass transit system. In such a system, it is acceptable for the bus or train to arrive slightly later or earlier than the scheduled arrival time. However, arriving very late or very early will inconvenience passengers and can generate a backlog in other parts of the system.

For all real-time systems, missing a deadline is considered a system failure. The softness of

the real-time system determines the consequences for failure. If a bus slips its deadline by five minutes, the consequences are very small or non-existent. However, in a hard real-time system, because a single failure can have severe effects, such systems usually have multiple redundant systems that can take over should the primary system fail.

2.3.2 Designing Real-Time Systems

Real-time systems have been well-studied in computer science, and there are now a number of common methodologies that are used to design them [3]. One such methodology is a *process-based* design, in which each component of the system is encapsulated inside a separate process running on a *real-time operating system*. In a real-time operating system, each process is assigned a priority, and is scheduled to run by a real-time scheduler based on that priority.

Processes in this design can perform any task necessary; however, a number of common roles have been recognized.

A *Worker* is a process that performs a specific function. Other processes send their requests to the Worker, and the Worker eventually replies to them with the results.

An *Administrator* is a process that provides a service to other processes in the system. Administrators act as a central contact point and coordinator for their Workers. If a process wants a certain task to be performed, it contacts the appropriate Administrator. The Administrator then farms the work out to one of its Workers. Once the task is complete, the Worker sends the results to the requesting process (possibly through the Worker's Administrator).

A *Notifier* is a process that waits for a specific event to occur (such as waiting for a timer to go off). Once the event has occurred, the Notifier sends a message to its Administrator and begins waiting for the event once more.

A *Courier* is a process that carries messages between Administrators. Couriers are used to ensure that no Administrator even needs to wait for another process to acknowledge its messages: the Courier does the waiting for it.

We have used these concepts from general real-time systems programming to design our architecture for a real-time multi-agent system.

2.4 Real-Time Multi-Agent Systems

We can generate the necessary properties of a soft real-time multi-agent system (RTMAS) from the definitions given above[1]. In particular, a RTMAS must meet these criteria:

- A RTMAS must be able to form plans that meet temporal constraints,
- The RTMAS planner must produce such plans within its own temporal constraints.
- The RTMAS must have a mechanism to ensure that the steps of a plan are executed at the appropriate times, and that any possible problems are dealt with such that the plan schedule is met with a best-effort attempt.
- No operation of the RTMAS must take an unbounded amount of time to complete.

To meet these requirements, we propose borrowing software design principles from (non-AI) real-time systems programming to ensure that all parts of the agent meet the required real-time constraints. For example, in a real-time system, algorithms must usually complete in $O(1)$ time. Failing this, the algorithm must be anytime; that is, they must be interruptible at any time and still produce a valid solution.

These principles imply that such things as planners and communications infrastructures must be structured in such a way that any time constraints present in the plans can be met. They suggest that, to create a MAS that must operate in a real-time environment, the MAS must be designed from the start with real-time principles in mind.

2.5 An Example

This example will help to clarify the terms used in this thesis.

Suppose a robotic agent wishes to travel from one point (say, A) to another, (B). It thus has a goal, "Set $Position = B$." To achieve the goal, it formulates a plan that looks like this:

1. Compute distance d and direction α to point B.

[1]Note that our focus in this thesis is on soft real-time multi-agent systems; we do not discuss a hard real-time system.

2. Turn α degrees, to face B.

3. Move forward a distance of d.

4. Stop.

However, to execute each of these steps the agent must break them down into further steps. For example, to achieve task 3, the agent needs to engage some sort of drive system, which may require that it first perform another set of tasks to, for example, warm up the drive system. It also must have a method for monitoring how far it has driven so that it may stop at the appropriate point.

Because the drive system of the robot is a mechanical system, it must work within a number of constraints (such as the laws of physics) that most software systems do not need to worry about. Therefore, the software that controls this system must also understand and cope with those constraints, and so must any software that depends on this control software. The result of this "bubble-up" effect is that even the highest-level planning software in the system must have some method for generating plans that involve temporal constraints. If the drive system always requires five seconds to engage, and 10 more seconds to achieve full speed, any plans that assume that the drive system is engaged immediately upon its activation will likely fail.

Similarly, once the drive system is engaged and the robot begins to move, it must have some mechanism for determining when it has reached its destination. A naive solution to this problem would compute the distance needed to travel, monitor the speed of the robot, and issue a "stop" command to the drive system once the distance has been traversed. However, the robot cannot stop instantaneously under its own power because its inertia will tend to keep it moving. Therefore, for the robot to arrive at the correct point, the "stop" command must be issued to the drive system *before* the robot arrives at B. The time at which this command must be issued can generally be predicted in advance by the robot, but occasionally this time must be checked as the robot moves to ensure its accuracy.

To facilitate the periodic checks of the time of the "stop" command, as well as the sending of the command itself, the robot needs a *real-time scheduler*. This scheduler works with the planner and other subsystems to allow the execution of a task at the exact time that task is required to be executed. The scheduler can also be used to determine if execution of a task is possible (by comparing the time limits of the task against its current schedule and looking for conflicts).

Clearly, even the relatively simple-seeming goal of driving from one point to another requires a number of systems to function correctly to be achieved. If this agent were working with other agents in a MAS, this simple scenario would also have to include the agent's interactions with the other members of the system.

2.6 Real-Time Planning

2.6.1 TAEMS

Vincent, Horling, Lesser, and Wagner [21] describe an implementation of a framework called TAEMS. TAEMS is a model for representing the interdependencies between agents that are planning in a real-time system.

They use an augmented plan library as part of their planner. Instead of storing specific plans, their TAEMS-based system stores plan templates. Each template provides multiple ways of achieving a goal, each with an associated quality. The template also stores the dependency order of the individual steps needed to reach the goal (e.g., step A must complete before step C can be started).

When the system is given a new goal, it selects those templates from the plan library that accomplish that goal and passes them to the planner. The planner has a fixed time interval in which to complete its job; if the interval elapses before a plan is finalized, the current-best plan will be used instead.

The planner's first action is to relax all the soft constraints imposed on the goal, leaving only the hard constraints. It discards any plans that do not meed the hard constraints, re-imposes the soft constraints, and begins iteratively searching the plan-space for better plans. At the end of each iteration, the current-best plan is recorded. Once a final plan has been found (either because the entire plan-space was searched or because the interval expired), the planner gives the final plan to a partial order scheduler.

The partial order scheduler's job is to arrange the steps of the plan in an order that satisfies all the time and resource limits and any dependencies on other steps, but allows unrelated steps to execute concurrently. At this point the plan schedule can be executed by the agent, or it can be further optimized by one or more of the three remaining modules.

The Resource Modeller adjusts the schedule to accommodate any resource usage limitations. For example, if two current plans both require access to the same device, one of them will have to wait for the other to finish. Any new schedule produced by this module must still meet the hard deadlines imposed on the initial goal.

The Task Merger looks for tasks that are common to all currently executing plans, and merges them to avoid duplication of work. For example, if two plans both depend on task A to complete before they can proceed, the Task Merger will reorder the schedules so that only one task A is ever executed. Both plans will wait for the completion of the same task A.

The Conflict Resolver deals with any conflicts that could not be resolved by the other two modules. Its response to a conflict between two plans is to simply drop (or ”decommit to”) the lower priority conflicting plan to allow the higher priority one to proceed.

Because the time that each step of a plan takes may be uncertain, the planning and scheduling parts of the system operate using the expected time taken for each step. Thus, the agent must monitor the execution of the plan, and quickly deal with any missed deadlines. Because the scheduler can schedule an individual plan's steps relatively independently, if a plan begins to fall behind its schedule, the scheduler can reorganize the schedule to compensate, avoiding a full re-planning of all affected tasks. The various modules, including the conflict manager, are again invoked on the new schedule.

This same behaviour allows easy integration of new goals and plans into the existing schedule. When a new plan is formulated, the scheduler can insert the individual steps of the new plan into the existing scheduler without forcing all existing plans to be recomputed.

In experiments, the authors set up a four-sensor network to track a small moving target. Each sensor was connected to a PC and could communicate with the others via a radio link, allowing the sensors to synchronize with each other when taking measurements. In their experiments, their scheduling system was able to avoid a full re-planning of all tasks 43% of the time. As well, the four agents were able to synchronize their sensor measurements to within 58 ms of each other, which the authors state is well below their desired threshold of one second.

This research provides a soft real-time planner and scheduler that could be used with our model in that capacity, as illustrated in the example in Section 2.5.

2.6.2 CIRCA

Atkins, Abdelzaher, Shin, and Durfee extend an architecture known as the Cooperative Intelligent Real-time Control Architecture (CIRCA) [1], described originally in [12], to include greater fault-tolerance and the ability to plan with multiple types of resources.

CIRCA is composed of two loosely coupled subsystems: a real-time control subsystem and an artificial intelligence subsystem. The real-time control subsystem provides monitoring and execution of individual real-time tasks, while the artificial intelligence subsystem is responsible for determining which tasks should be executed.

To provide a real-time response even in the artificial intelligence subsystem, CIRCA relies on an extensive plan library and a reactive planning system. A reactive planning system is a planning system in which each plan is composed of test-action pairs (TAPs); that is, each plan consists of a set of state requirements and a set of actions to be performed when those requirements are met. The real-time scheduler can then perform these tests at the appropriate times, and if the state requirements of the plan match the observed state, the scheduler executes the action associated with the test. If the test fails, the schedule can execute a remedy action, which is designed to correct the inconsistencies between the expected and actual states.

CIRCA's real-time guarantees come from the planner's ability to generate TAPs with a scheduled testing time sufficiently in advance of the action's deadline that the remedy action (taken should the test fail) will complete by the deadline. For example, if the landing gear on an autonomous unmanned aerial vehicle (UAV) must be extended by time T, and it takes t_l time to extend the landing gear, the latest the TAP must be executed is time $T - t_l$. In addition, any remedies that would need to be executed should the landing gear fail to extend (such as retracting and retrying) need also be incorporated into this time. That is, the maximum time to execute the original TAP is:

$$T - \sum_{\substack{all \\ remedies}} t_{remedy}$$

CIRCA provides a hard real-time planner and scheduler that could similarly be used with our model in that capacity, as illustrated in the example in Section 2.5.

2.7 Task Allocation

In a multi-agent system, individual agents rarely have complete, or even good, knowledge of what other agents in the system are doing. This presents a challenge when one agent needs to assign a task to another. Various solutions to this problem have been proposed for the general problem of assigning tasks to agents in a multi-agent system. However, the special requirements of a real-time multi-agent system means that many of these proposed solutions are infeasible, as they assume capabilities, such as unlimited time or communication ability, that are not available in a real-time environment.

Part of our framework requires a mechanism for assigning tasks to agents. We present some research related to this requirement here.

2.7.1 Ortiz and Rauenbush

Ortiz and Rauenbush [15] describe a system capable of allocating tasks amongst distributed agents by reasoning about temporal and spatial constraints. As well, their solution can handle situations where the set of tasks and the set of agents can change unexpectedly. Their system uses an anytime auction-based algorithm called the Mediation Algorithm. Because the algorithm is anytime, it is suitable for use in time-sensitive environments such as a RTMAS.

Whenever a new task needs to be assigned, a mediator agent is chosen. This mediator then announces a proposed assignment for the new task to a set of bidders (agents who could accomplish the task). The candidates then produce bids that contain information about their current task (if any), their bid on the new task, and a potential bid on the task if the agent weren't otherwise occupied. These *enriched bids* allow the mediator to modify the proposal and adjust the list of candidates for a task. The mediator then issues the revised proposal to the current set of candidates. The process repeats until the algorithm's time limit is exceeded or until every task has been satisfactorily assigned.

The mediator can modify the proposal by either dropping a task that is to be assigned or by adding either an already assigned task or a new task into the auction. Adding a task into the auction provides the mediator with a way to resolve conflicts between two tasks. If a task t_0 must by done by agent A_1, but A_1 is already assigned to task t_1, the mediator can reintroduce t_1 into the proposal, freeing A_1 and allowing it to be assigned to t_0 instead, if necessary. t_1 can then be

reassigned to a different agent instead.

Similarly, the mediator can choose to drop a task from the proposal, if the task has been completed, is no longer relevant, or conflicts with a higher-priority task. In this case, the task is discarded, at least temporarily, and is not assigned to any agent.

Likewise, the mediator can add and remove agents from its list of candidates for a proposal. If a task must be executed at a specific location, the mediator might add an agent that is further from that location if all closer agents are occupied. The mediator might also elect to remove an agent that has become disabled (for example, due to a malfunction) from the list of candidates to prevent the agent from being assigned tasks.

In experiments, the authors found that their dynamic mediation method allowed for a better allocation of tasks to agents.

This task allocation mechanism is designed to assign tasks to agents in a distributed manner. We will contrast this approach with our approach, which integrates a central coordinating agent, in Chapter 7.

2.8 Adjustable Autonomy Systems

There are a growing number of researchers investigating the topic of adjustable autonomy in multi-agent systems (e.g. [8, 7, 10]; see also [6, 13, 17, 18, 19]). The challenge is to design a multi-agent system in such a way that either the user or the agents themselves can take the initiative to have the autonomy of the agents adjusted. For example, initially agents may be fully autonomous to carry out certain designated tasks. However, in the face of great uncertainty, these agents may reason that it is better for them to forgo some of their autonomy and have another party dictate the actions to be followed (e.g. [6, 17]). In other cases, it may be useful for a human user to establish, ahead of time, certain policies that allow an autonomous agent to return to the user for further direction, giving up some of its autonomy. The former strategy is known as *agent-directed adjustable autonomy*, while the latter is referred to as *user-directed adjustable autonomy* [10].

Some of the research questions that have arisen in the adjustable autonomy community include when and how to monitor other parties, how to model reliability and trust to ensure effective delegation of responsibilities, and how to design a software architecture to facilitate adjustment

of autonomy. In general, then, it is useful to be able to stipulate the conditions under which an agent may elect to forgo some of its autonomy for the improvement of the overall problem solving.

In this thesis we propose a specific level of autonomy for the agents in the multi-agent system, allowing some of them to be fully autonomous, while others have their actions dictated by a controller agent. This enables more effective "direction" of the agents that are controlled, while allowing a segment of the society to freely explore, enjoying the benefits of being autonomous, such as the ability to acquire new goals and perform actions based on local circumstances.

Chapter 3

The Model

3.1 Introduction

Recall the scenario for a multi-agent system, described in Chapter 1:

- There is a task/problem to be solved by the system of agents;
- Agents are able to reason about what they are doing;
- Agents are allowed to communicate with each other and with a human;
- The human provides the initial goal and specifications;
- Agents are allowed to sense their environment dynamically;
- Agents carry out actions and contribute to the overall task (the mission);
- There are real-time issues that need to be addressed, such as a deadline for mission or subtask completion;
- The environment the agents are working in can change unexpectedly.

In this chapter we present an architecture that is designed to support such a system. The proposed architecture is illustrated with examples, and in particular a running example of a lander and its group of four robotic explorers (rovers) on the planet Mars.

3.2 Structure

Our research aims to provide both an architecture for designing a multi-agent system and a specification for communication within the system.

Our architecture has two types of agents: the *worker* agents do tasks and make observations, while the *coordinator* agent oversees the workers and ensures that the system's mission is completed successfully.

The coordinator is the system's interface to the outside world, including any human overseers and possibly the coordinators of other systems. All communication to the workers must come from the coordinator, and all communication from the workers must go to the coordinator. The coordinator is also responsible for assigning a mission (i.e., a top-level goal) to each individual worker, and the coordinator is the only agent in the system that can modify a worker's mission. The coordinator receives the system's mission from an overseer (human, or another agent), and cannot modify its own mission.

In our Mars example, the lander will take on the role of the coordinator, C, and the rovers will be worker agents $W_0, \cdots, W_3$. The mission given to C by its human overseers on Earth is "Explore the terrain within 2 km of the lander." Upon receipt of this mission from the ground, C generates a plan to use its agents to explore the area. The plan is (in general terms), "W_0 explores north, W_1 east, W_2 south, and W_3 west." Finally, C gives the workers their individual missions using the plan it has generated. The missions are:

- W_0: "Explore the terrain within a 90 degree arc up to 2 km north of the lander."
- W_1: "Explore the terrain within a 90 degree arc up to 2 km east of the lander."
- W_2: "Explore the terrain within a 90 degree arc up to 2 km south of the lander."
- W_3: "Explore the terrain within a 90 degree arc up to 2 km west of the lander."

$W_0, \cdots, W_3$ are now free to execute their missions however they see fit.

3.3 Worker Agent Behaviour

One research problem we aim to address is what level of autonomy should be given to the agents in the community. As will be outlined below, our approach is to support different levels of

autonomy within the overall system. A subset of worker agents will be fully autonomous, while others will be under the direct control of the coordinator.

To reduce the need for communication, each worker is initially given a high level of autonomy, and while workers are aware of the existence of other workers, they do not communicate with any of their fellows. Coordination amongst workers is the responsibility of the coordinator agent, and is described in the next section.

When a worker is fully autonomous in our system it generates plans that will enable it to accomplish its mission. In addition, an autonomous worker can generate and assign itself new tasks and goals, subject to the constraints of its mission. When a worker encounters an unexpected condition or situation it may choose to cede some of its autonomy to the coordinator. We describe three types of events that workers can encounter.

3.3.1 Opportunities

An **Opportunity** is a condition or situation that, while unexpected, is not detrimental to the worker and is within the scope of the worker's current mission. When worker agent W_i encounters an Opportunity it can autonomously take advantage of it without first consulting the coordinator. That is, W_i can generate any new tasks or goals necessary to further its mission in the context of the Opportunity. Note that not all Opportunities need be exploited; when to generate new goals and plans is a domain-specific issue. See, e.g., [5].

If W_i decides to take advantage of the Opportunity, it contacts its coordinator, C, and reports the Opportunity and its new self-assigned tasks. This message is purely for informational purposes: W_i does not wait for or expect a response from C. Thus, W_i retains full autonomy after an Opportunity event has occurred. If W_i cannot take advantage of an Opportunity (for example, due to other, higher-priority tasks), it sends a message to C describing the Opportunity to allow C to possibly exploit the Opportunity using a different worker.

In the Mars example described earlier, we can suppose worker W_0 is travelling north when it comes upon a large rock. W_0 performs a preliminary evaluation of the rock and determines that it is worth further study. The robot generates a new task, "Study this interesting rock," and sends it, along with the location of the rock, to C as a Notification message. W_0 starts studying the rock immediately. Once its study is complete it will report its results and resume its mission.

3.3.2 Barriers

A **Barrier** is a condition or situation that prevents a worker from progressing towards its goals. When worker W_i encounters a Barrier, it attempts to generate a plan to overcome the Barrier. If a plan can be found, W_i informs the coordinator of its situation and its solution and proceeds to execute its plan. In this case the worker does not lose its autonomy.

If W_i cannot find a plan to solve the problem it contacts its coordinator and reports a description of the problem. W_i completely surrenders its autonomy to C in order to solve the problem (that is, W_i stops working on its tasks and waits for instructions from C). The coordinator is responsible for determining the solution to W_i's problem (the details of the coordinator's decision making are described in the next section), which may require that W_i perform some actions. W_i does not regain its autonomy until it is released by C.

In our example, suppose worker W_0 is travelling north again when it detects a sharp change in elevation ahead: the edge of a pit. The pit is not small, and its edge extends beyond the range of W_0's sensors. An attempt to find a plan to overcome the obstacle fails, so W_0 contacts C to request help navigating around the pit and waits for a solution.

Because C has access to a map provided by a satellite, it is able to formulate a plan to see W_0 safely around the pit. C responds to W_0's request with a command for W_0 to travel 40 metres west of its current position, followed by a command to travel 10 metres north. W_0 executes the commands and travels around the pit. C relinquishes command over W_0 once it finishes executing the plan, allowing it to become fully autonomous once again.

3.3.3 Potential Causes of Failure

A **Potential Cause of Failure (PCF)** is a condition or situation that, if not corrected, will cause failure in the *future*. The worker can still make progress towards its goals up until the time of the failure. When worker W_i believes that an event in the future will endanger one of its goals, it attempts to generate a plan that can avert the failure. If the attempt succeeds, the plan is executed and W_i informs the coordinator of its new state.

If a plan cannot be found that will avoid the failure, W_i contacts the coordinator (C) and sends a description of the problem, as with a Barrier. However, W_i continues working on its tasks, since, unlike a Barrier, there is no immediate impediment to their accomplishment. Thus,

W_i gives up only part of its autonomy to C: W_i can still generate its own tasks and plans until C takes control (when it finds a solution). W_i remains under the control of C until it is released by C.

For an example of a worker's response to a PCF, see the RoboCup Rescue example in the next chapter.

3.4 Coordinator Behaviour

3.4.1 Worker Message Types

One challenge in designing our processing algorithms for agents is determining the conditions for communication within the community of agents. We propose that communication is always channelled through the coordinator, using the framework described below. Through this communication, the coordinator continuously tracks the state of the worker agents in the community and maintains a global state representation of the environment. Workers can send three types of messages to the coordinator.

Heartbeat messages are sent by workers periodically simply to keep the coordinator up-to-date on the worker's state. The frequency of the Heartbeats is determined by the communications and processing limitations of the coordinator, as well as the requirements of the problem domain. Heartbeat messages do not require a response from the coordinator.

Notification messages are sent whenever an agent exploits an Opportunity, when an agent generates a plan to solve a problem (a Barrier or PCF), or when an agent completes a task assigned to it by the coordinator. Notification messages are also used to report the results of a task or goal completion (such as the results of a scientific instrument) to the coordinator. Like Heartbeats, Notification messages do not require a response.

Panic messages are sent when a worker (or possibly even the coordinator for another system) requires the assistance or direction of the coordinator. Panic messages are sent by workers in response to Barrier and PCF events that the worker cannot solve, and will result in the coordinator taking an active role in the affected agent's problem.

In the situation where all workers are proceeding towards their mission goals, the coordinator will have relatively little to do. In this scenario, the only types of messages the coordinator will receive are Notifications and Heartbeats. The coordinator's processing will be limited to incorporating the information from these messages into its global state representation. However, if the coordinator receives a Panic message from worker W_i it will have to formulate and implement a solution, possibly by a deadline.

3.4.2 Coordinator Messaging

The messages the coordinator sends to individual workers do not fall into the three categories described above. Because each coordinator message is essentially a command, instructing a worker to start or stop performing a task, there is no need to prioritize between messages; all coordinator messages are important.

Though the specifics of the coordinator's messages are domain-dependent, there are certain messages that will be present in any application of our system.

New Mission messages are sent by the coordinator to assign or reassign a worker's top-level goal. A worker must first receive this message from the coordinator before it can begin to perform any tasks. The coordinator can also use this type of message to reassign a worker to a new task without revoking its autonomy. For example, if worker W_i encounters a Barrier that it cannot itself handle, the coordinator may elect to give W_i a new mission rather than to revoke W_i's autonomy.

This message may also include constraints the coordinator wishes to place on the worker.

Revoke Autonomy messages are sent to a worker when the coordinator wishes to take control of the worker. The message body should include a new plan or goal for the controlled worker to begin working on.

Restore Autonomy messages release a worker from the coordinator's control. This message complements the *Revoke Autonomy* message, above.

New Task messages reassign an already-controlled worker to a new task. The coordinator can send such a message to a worker if it must use the worker to perform a new task before the first task it assigned the worker has been completed.

See Section 5.7.2 for a discussion of how these messages could be used in an implementation based on our model.

3.4.3 Task Assignment and Reassignment

The coordinator can receive new tasks at any time. From the discussion above we see three sources of new tasks in the system:

1. **From a worker**: These new tasks are the result of an Opportunity being exercised, or of a worker solving a problem (Barrier or PCF) on its own. No action on the part of the coordinator is needed: the worker has already self-assigned the new task.

2. **From the coordinator itself**: These new tasks are part of the solution to a Panic message (a Barrier or PCF event), and need to be assigned to one or more workers for execution.

3. **From outside the system**: These new tasks come from a human overseer, or from the coordinator of another system. The new task needs to be assigned to one or more workers for execution.

The coordinator is responsible for selecting the workers to assign to each new task. Because the coordinator tracks the system's global state by receiving Heartbeat and Notification messages from its workers, it is able to select the 'best' workers to assign to the task. Selection of a worker for a task in this way forcibly decreases a worker's autonomy.

Once a worker completes all the tasks assigned to it by C, its autonomy is automatically restored and the worker may continue working on its autonomous goals.

3.5 Real-Time

We have designed our system from the ground up to support real-time decision making and response. In particular, our architecture is based on a model commonly found in real-time control systems, in which there is a central Administrator process and a number of Worker processes. The Administrator is responsible for dispatching Worker processes, for handling interactions between Workers, and for facilitating communication between Workers and other processes in the

system. We have adapted this model to allow the Workers and Administrator to be independent autonomous agents.

Our system is also designed to support real-time application domains and real-time failures. By their nature, PCF events have a deadline by which they must be resolved (recall that a PCF is a problem that will cause failure in the future). In this manner, our system can incorporate application domain-imposed real-time constraints. We treat such deadlines as PCF events: a worker will generate a PCF event if it comes to believe that the task with the time constraint will not be completed by its deadline. To detect potential problems, the worker agent must be able to predict the time of failure. Thus, PCF events are real-time deadlines imposed on the coordinator: the coordinator must determine a solution and ensure its execution is completed before the failure occurs.

When operating under real-time constraints it is especially important to minimize response time. In our architecture, workers are able to solve problems locally if possible, eliminating the communication delay of contacting the coordinator. The extent to which a worker may solve problems on its own is dependent on the problem domain, and is dictated by the worker's mission. For example, the mission might explicitly specify that a worker travelling to a destination must keep its speed below a certain threshold. If the worker determines that it cannot arrive at the destination by the required time, it is allowed to increase its speed. However, if the increase in speed exceeds the threshold, the worker will instead be required to request help from the coordinator (in the form of a PCF). The coordinator can then instruct the worker to adjust its speed to a value higher than the threshold, or even to take a different path.

3.6 Implementing The System

In this section, we discuss in more detail some elements of the design that must be integrated into an implementation of a multi-agent system based on our proposed architecture.

Failure of some parts of the system, including missed deadlines, will cause the whole system to fail. To avoid missing deadlines, it is important to know the minimum time guarantee that can be made by the system. This minimum is bounded by the length of time it takes to execute each code path through the system. The time required to execute the longest code path determines the minimum response time for the system, regardless of the likelihood of its execution. For

example, if the longest code path in the system takes 700 ms to execute, but is expected to execute only very rarely, the minimum time guarantee that can be made by the system is still 700 ms since in the worst case the system will have to execute the code on that path.

We assume the following components are available.

A Real-Time Planner And Scheduler: A real-time planner and scheduler such as those described in Chapter 2 are needed.

A State Evaluator: A mechanism for detecting Opportunities, and for detecting plan failures (Barriers) and potential failures (PCFs). For PCFs, the state evaluator needs to also determine the time of the potential failure.

An Opportunity Evaluator: A mechanism for determining the utility of an Opportunity, such as is described in [5].

A Task Allocator: A mechanism to allow the coordinator to select workers to perform tasks it needs to perform.

In addition, because the run-time of each of the above components has an effect on the minimum time guarantee, all of the algorithms used in the time-sensitive portions of these components must be either constant-time or anytime algorithms.

3.6.1 Representing The Environment

Each worker keeps a model of its local environment (that which it can directly observe). The model is updated as new information is received from the worker's sensors. The information in the state tables is used by the worker's planner for evaluating possible plans, and by its state evaluator to determine if an Opportunity, Barrier, or PCF event has occurred.

While much of the information in the local and global state tables depends on the problem domain characteristics, there are some fields that are necessary to manage the system. Each worker's state includes information such as its current mission, its schedule of actions, and its *state history*.

Whenever a worker sends a message (Heartbeat, Notification, or Panic) to the coordinator, it includes local state updates. The coordinator merges the local state updates into its global state

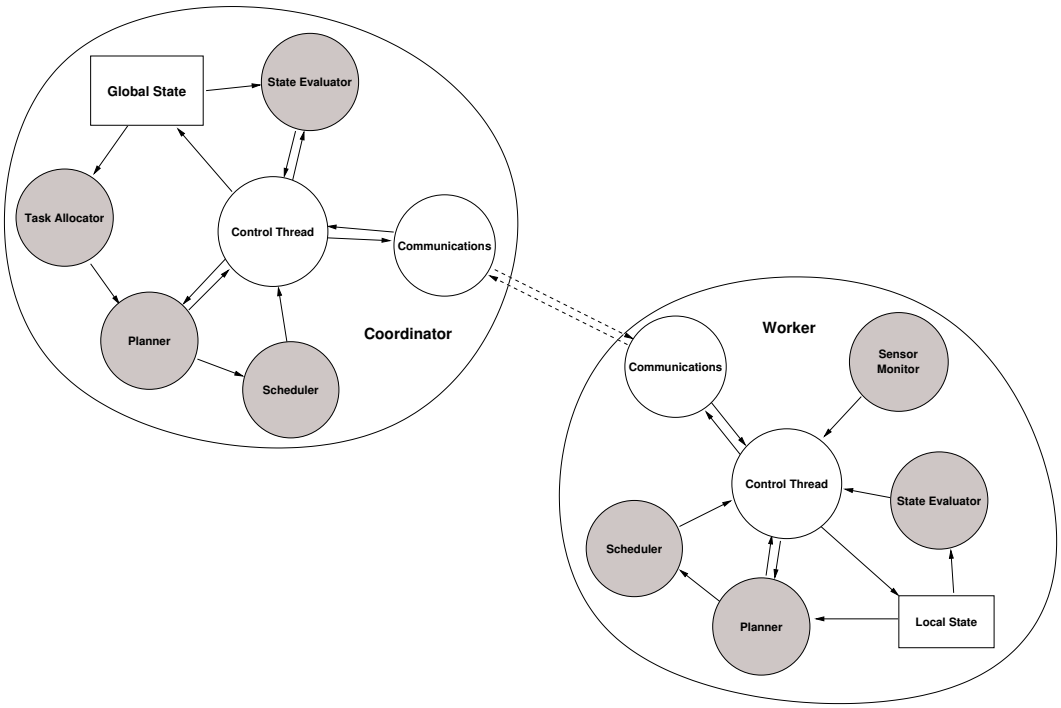

Figure 3.1: Coordinator and worker process interaction. Grey circles indicate "plug-in" components.

tables (Figure 3.2), which it uses to generate and execute plans, coordinate worker interactions, and deal with failures. The frequency at which these messages are sent to the coordinator affects the coordinator's ability to reason about its workers. When tailoring our system for a particular problem, consideration must be given to the effect that stale information has on the coordinator's ability to reason.

Heartbeat messages provide a mechanism for controlling this frequency by defining a maximum time a worker can go before it sends an update to the coordinator. However, even with frequent Heartbeat messages, the information in the coordinator's state model is updated only at specific points in time and therefore may be out-of date. Thus the coordinator stores a timestamp with the data whenever the state is updated to indicate when the state was last known to be true. Each worker's state table includes a *state history* that stores the complete worker state for a number of previous timestamps, allowing the worker to monitor trends in the data as well.

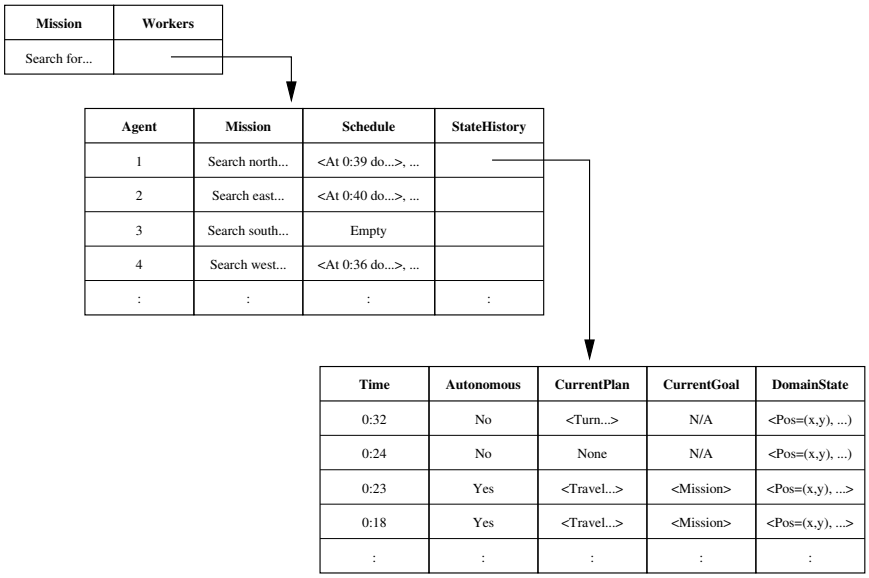

Figure 3.2: Example global state representation. The system's mission is in the top-level table. Each row in the middle table corresponds to an individual worker, while each row in the bottom table contains a snapshot of the worker's state at various times in the past.

For example, the bottom chart in Figure 3.2 shows Agent1's state history.

3.6.2 Worker Details

Each worker is composed of its local state, a stack of its goals, and a number of threads of execution. The local state is described in the previous section. The goal stack is used to track the worker's current goal. At the bottom of the goal stack is the worker's mission as given to it by the coordinator. As the worker generates new goals they are pushed onto the stack, and as the worker accomplishes them they are popped off it. The goal stack is only used when the worker is autonomous.

The core of each worker is its control thread. A worker's control thread monitors all of its other threads and contains the logic that manages the worker's reasoning and planning capabilities. There are also planner and scheduler threads responsible for generating and executing the worker's plans. Other threads in the worker monitor its sensors and, in the case of agents that are

actually robots, control its physical machinery.

The worker's control thread must iterate quickly to allow fast response to any problems. The following will illustrate how Barriers, Opportunities, and Potential Causes of Failure are handled by the worker:

0. Initially the worker is autonomous and the goal stack is empty. The worker cannot begin generating new goals and plans until it receives its mission from the coordinator via a *New Mission* message.

1. When sensor threads receive new data from their sensors they send a message to the control thread with the new data. The control thread incorporates this new information into its local state, then checks whether the new information presents an Opportunity, Barrier, or PCF event.

2. If the coordinator sends the worker a *New Mission* message, the worker stops executing its schedule, clears all its current plans and the goal stack, and pushes the new mission onto the goal stack. It then generates a new plan to accomplish the mission and begins executing it.

3. If the coordinator sends the worker a *Revoke Autonomy* message, the worker stops executing its own schedule and begins executing the plans sent to it by the coordinator. As the worker completes each plan's execution, it sends a *Task Completion* Notification message to the coordinator requesting the next task for it to do. The coordinator may reply with a *New Task* message, or with a *Restore Autonomy* message.

4. If the coordinator sends a *Restore Autonomy* message, the worker reexamines the goals on its goal stack, removing those that can no longer be performed (for example, due to changes in the environment). It then generates a new plan for the goal at the top of the stack and resumes its autonomous execution.

5. If the coordinator sends the worker a *New Task* message, the worker simply takes the new task and begins executing it.

6. If the worker is autonomous and a goal has been accomplished it is removed from the goal stack.

7. If an Opportunity has been detected and the worker is autonomous, the control thread evaluates the utility of exploiting the Opportunity. If the Opportunity is not sufficiently interesting or useful, it is ignored. Otherwise, the control thread instructs the planner to search for a plan that can take advantage of the Opportunity. The planner must find a plan that achieves the goal of exploiting the Opportunity while satisfying the mission constraints.

   If the planner finds a plan, the control thread begins executing it and the Opportunity is thus exploited. A Notification message is sent to the coordinator informing it of the worker's new goal, which is also pushed onto the goal stack. If the planner fails to find a plan (perhaps due to conflicting time or mission constraints), the control thread informs the coordinator of the unexploited Opportunity with a different Notification message.

8. If a Barrier or PCF has been detected and the worker is autonomous, the control thread again instructs the planner to search for a plan that can remove the Barrier or avoid the PCF. If the planner finds such a plan, it is inserted into the scheduler and a Notification message is sent to the coordinator with the new goal (which is to solve the Barrier/PCF). The new goal is pushed onto the goal stack.

   If the planner cannot find such a plan, or if the worker is not operating autonomously, the control thread sends a Panic message to the coordinator informing it of the problem. In the case of a Barrier, the worker suspends its activity, while in the case of a PCF the worker attempts to estimate the PCF's deadline but continues working on its previous tasks. The coordinator may respond with a *Revoke Autonomy* or *New Mission* message when it finds a solution.

9. If the worker completes all of the above actions without sending a message to the coordinator, it may send a Heartbeat message to the coordinator.

10. Repeat steps 1 to 10.

3.6.3 Coordinator Details

The coordinator's control thread is as follows:

0. The coordinator starts when it receives its mission from its overseers (a human or another agent). The coordinator sends *New Mission* messages to each of its workers, and sets their initial state in its global state table to be autonomous.

1. If the coordinator receives a Heartbeat message from a worker, it takes the updated state information contained in the message and merges it into its global state.

2. If the coordinator receives a *Task Completion* Notification message from a worker and the worker is currently controlled by the coordinator, it examines its schedule for that worker. If there are still tasks on the schedule, the next task is sent to the worker as a *New Task* message.

   If there are no more tasks for the worker to perform, the coordinator restores the worker's autonomy by sending the agent a *Restore Autonomy* message and updating its state tables.

   If the worker is not controlled by the coordinator, the coordinator just updates its global state to reflect the worker's changed goal.

3. If the coordinator receives Notification of an exploited Opportunity from a worker, it simply updates its state tables to include the worker's new state.

   If the Notification is of an unexploited Opportunity, the coordinator evaluates the utility of exploiting the Opportunity. If the Opportunity is sufficiently worthwhile, it generates a plan to exploit the Opportunity and selects an appropriate worker to perform the task. The worker is sent a *Revoke Autonomy* or *New Task* message containing the plan. If the Opportunity is not sufficiently interesting, or if no plan can be found to exploit it, it is ignored.

4. If the coordinator receives a Notification message of a Barrier or PCF from a worker, it simply updates its state tables to include the worker's new state.

5. If the coordinator receives a Panic message of a Barrier from a worker, it generates a plan to overcome the Barrier and selects one or more workers to execute the plan. The worker(s) selected may or may not include the worker that originally encountered the Barrier. Those selected are sent *Revoke Autonomy* or *New Task* messages containing the new tasks they are to perform.

6. If the coordinator receives a Panic message of a PCF from a worker, it verifies the deadline provided by the worker and then generates a plan to overcome the PCF within the deadline. The coordinator then selects one or more workers to execute the plan. The worker(s) selected may or may not include the worker that originally encountered the PCF. Those selected are sent *Revoke Autonomy* or *New Task* messages containing the new tasks they are to perform.

7. Repeat steps 1 to 7.

Chapter 4

Examples

4.1 RoboCup Rescue

RoboCup Rescue [20] is an annual competition designed to foster research into intelligent robotics for search-and-rescue. It is related to the more well-known RoboCup Soccer competitions. Currently, it has two leagues: robot and simulation. In this chapter we focus on the simulation league, which is centred around multi-agent planning and coordination.

In the simulation league, rescue agents are designated as one of Police, Fire, or Ambulance agents. Each agent type has its own special ability: only fire agents can extinguish fires, only ambulance agents can rescue civilians, and only police agents can clear blocked roads.

Each rescue agent type has one "dispatch centre" agent. Agents of one type can send messages to all other agents of that type, including the dispatch centre for that type, by using their radio. The three dispatch centres can send messages to each other as well as to their own agents. There are also Civilian agents that are controlled by the simulator and represent people to be rescued. Civilians can be trapped under rubble, which affects the time needed to rescue them.

The simulation begins with a simulated disaster (generally an earthquake). The disaster simulators begin simulating collapsed buildings, fires, and blocked roads. The civilian simulator tracks the health of each civilian, which deteriorates over time if the civilian is trapped or on fire. The goal of the simulation is to minimize both civilian casualties and property damage (due to fires, which spread quickly).

This structure lends itself well to our model, as follows:

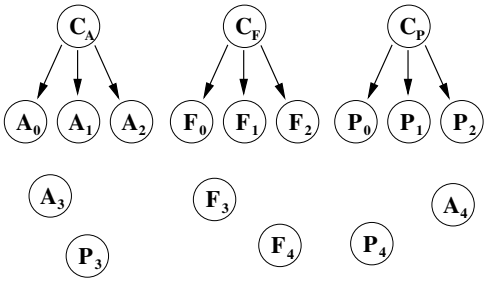

Figure 4.1: Our system can be mapped to the RoboCup Rescue simulation scenario by using three coordinators, one for each service type.

- The individual rescue agents are the workers.
- The centre agent for each service is the coordinator for all the agents in that service.
- The centre agent acts as the contact point for the other two services.

4.1.1 Example

Suppose we have a RoboCup Rescue simulation with these agents (Figure 4.1):

- Ambulance agents A_i, with dispatch centre C_A.
- Police agents P_i, with dispatch centre C_P.
- Fire agents F_i, with dispatch centre C_F.

The mission given to the ambulance agent A_3 is "Rescue trapped civilians." At some point in time, A_3 finds a trapped civilian and generates a new sub-goal, "Rescue this trapped civilian." A_3 expects this task to take 15 minutes. This is an Opportunity event, so A_3 sends a notification message to C_A informing it of the location and projected time to rescue for the civilian.

As A_3 proceeds, it detects that a nearby fire is spreading towards its location. Using its state evaluator, A_3 projects that the fire will reach the civilian in 8 minutes, and is therefore a PCF. It immediately sends a Panic message to C_A, but continues to rescue the civilian.

C_A's response depends on the global situation, which is not observable by A_3.

- The ideal solution is for C_A to request that C_F, the fire centre agent, send fire agents to extinguish the fire before it reaches the civilian. This solution would result in C_F generating new tasks and assigning them to some subset of fire agents.
- C_A might also dispatch more ambulance agents to help A_3, reassigning them from their current tasks if necessary. These helper agents would lose their autonomy (as would A_3) until the civilian is rescued.
- C_A can elect to implement both of the above solutions to maximize the chances that the civilian will be rescued.
- Another possibility is that, unknown to A_3, fire agents may already be fighting the fire in question and expect it to be extinguished within 6 minutes. If this is the case, C_A (or C_F) will recognize this condition and inform A_3 that the fire should be out before it reaches the civilian (if C_F is the agent to recognize the condition, it will inform C_A, which will inform A_3). Note that if the fire is not extinguished within the 6 minute target, A_3 can re-issue the PCF.
- Finally, C_A may recall A_3 if there is no chance that the civilian will be rescued before the fire reaches A_3's location, to avoid losing A_3 as well as the civilian.

4.2 Worker-Coordinator Interaction

This example steps through the interactions that would occur between a worker, W_0, robot and its coordinator, C, as the worker explores the surface of Mars. Heartbeats ensure a message is sent at least once every 5 "ticks". The worker's state is shown for both the worker and the coordinator's state tables (see Figure 3.2) as lines such as:

(A = *true* , CP = *empty* , CG = *empty*)

The fields used in this example are:

- **A**: Whether the worker is currently autonomous.

- **CP**: The worker's current plan. This field is *empty* if the worker is idle (such as when the worker is waiting for a message from the coordinator).
- **CG**: The goal at the top of the worker's goal stack. This field is *empty* if the worker has no mission or if the worker is not autonomous.

We suppose that W_0 is able to sense the terrain and atmosphere in its immediate vicinity (say, within 10 metres) using its long-range sensors, and able to navigate the surface of the planet using this information. We also suppose the worker has a number of instruments that it can use to perform detailed analyses of rocks, but that it must be adjacent to any rock to use them. W_0 can communicate with C via a radio link. Finally, W_0 can also monitor its own internal status (such as its power level) using its internal sensors.

4.2.1 Worker Startup And Plan Generation

This section illustrates the communication that takes place during the system's initialization.

Time	**Worker** W_0	**Coordinator** C
1	(A = *true* , CP = *empty* , CG = *empty*)	(A = *true* , CP = *empty* , CG = *empty*)
2	(A = *true* , CP = *empty* , CG = *empty*)	(A = *true* , CP = *empty* , CG = <*Explore...*>) Generate and send mission to W_0 <*Explore region north of lander*>.
3	(A = *true* , CP = *empty* , CG = <*Explore...*>) Receives mission, pushed onto goal stack.	(A = *true* , CP = *empty* , CG = <*Explore...*>)
4	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Generates a plan: <*Travel north, scan for interesting features...*> based on the mission goal.	(A = *true* , CP = *empty* , CG = <*Explore...*>)
5	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Scheduler begins executing plan. W_0 begins travelling north, scanning for interesting features of the terrain.	(A = *true* , CP = *empty* , CG = <*Explore...*>)
⋮	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Occasional Heartbeats sent.	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Heartbeats received.

4.2.2 A Barrier

This section illustrates the communication that takes place when the worker's long-range sensors detect a sharp change in elevation in front of the robot (a large, deep pit).

Time	**Worker** W_0	**Coordinator** C
16	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
17	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Sensors detect a steep change in elevation ahead. W_0's domain state is updated.	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
18	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) State evaluator detects a Barrier event. Cause is determined as <*elevation change in path exceeds maximum allowed.*>	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
19	(A = *true* , CP = , CG = <*Explore...*>) Attempts to generate a plan, no valid plans found. Sends a Panic message to C indicating the cause of the problem and stops executing any plans.	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
20	(A = *true* , CP = , CG = <*Explore...*>)	(A = *true* , CP = , CG = <*Explore...*>) Receives Panic message with Barrier. Cause of failure determined: elevation change is due to a large pit.
21	(A = *true* , CP = , CG = <*Explore...*>)	(A = *true* , CP = , CG = <*Explore...*>) Plan found: <*Travel west 40 meters, then travel north 10 meters.*> Worker selected for this plan is W_0 (the worker that encountered the Barrier).
22	(A = *true* , CP = , CG = <*Explore...*>)	(A = *false* , CP = <*Travel west...*>, CG = *empty*) <*Revoke Autonomy*> message sent to W_0. First part of plan is sent: <*Travel west 40 meters*>.
23	(A = *false* , CP = <*Travel west...*>, CG = *empty*) Receive <*Revoke Autonomy*> message.	(A = *false* , CP = <*Travel west...*>, CG = *empty*)
24	(A = *false* , CP = <*Travel west...*>, CG = *empty*) Scheduler begins executing new plan. Heartbeat.	(A = *false* , CP = <*Travel west...*>, CG = *empty*)
25	(A = *false* , CP = <*Travel west...*>, CG = *empty*)	Heartbeat received. Updates worker state. (A = *false* , CP = <*Travel west...*>, CG = *empty*)
⋮	(A = *false* , CP = <*Travel west...*>, CG = *empty*) Occasional Heartbeats sent.	(A = *false* , CP = <*Travel west...*>, CG = *empty*) Heartbeats received.
36	(A = *false* , CP = *empty* , CG = *empty*) First task complete. W_0 detects completion and sends <*Task Completion*> message to coordinator.	(A = *false* , CP = <*Travel west...*>, CG = *empty*)
37	(A = *false* , CP = *empty* , CG = *empty*)	(A = *false* , CP = *empty* , CG = *empty*) <*Task Completion*> message received.
38	(A = *false* , CP = *empty* , CG = *empty*)	(A = *false* , CP = <*Travel north...*>, CG = *empty*) There are still tasks in the schedule. Send a <*New Task*> message to W_0 with the next task (<*Travel north 10 meters*>).
39	(A = *false* , CP = <*Travel north...*>, CG = *empty*) Receives the <*New Task*> message	(A = *false* , CP = <*Travel north...*>, CG = *empty*)
40	(A = *false* , CP = <*Travel north...*>, CG = *empty*) Scheduler begins executing new plan.	(A = *false* , CP = <*Travel north...*>, CG = *empty*)

Time	**Worker** W_0	**Coordinator** C
⋮	(A = *false* , CP = <*Travel north...*>, CG = *empty*) Occasional Heartbeats sent.	(A = *false* , CP = <*Travel north...*>, CG = *empty*) Heartbeats received.
45	(A = *false* , CP = , CG = *empty*) Second task complete. W_0 detects completion and sends <*Task Completion*> message to C.	(A = *false* , CP = <*Travel north...*>, CG = *empty*)
46	(A = *false* , CP = , CG = *empty*)	(A = *false* , CP = , CG = *empty*) <*Task Completion*> message received.
47	(A = *false* , CP = , CG = *empty*)	(A = *true* , CP = , CG = *empty*) No further tasks to be done. Send <*Restore Autonomy*> message to W_0.
48	(A = *true* , CP = , CG = <*Explore...*>) Receives <*Restore Autonomy.*> Clean up and restore goal stack.	(A = *true* , CP = , CG = *empty*)
49	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Find a new plan for the current goal and resume execution. Scheduler begins executing new plan. Heartbeat	(A = *true* , CP = , CG = *empty*)
50	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Heartbeat received. Updates worker state for W_0.

4.2.3 An Opportunity

This example steps through the interactions that occur when the worker robot's long-range sensors detect a potentially interesting rock during the course of its exploration.

Time	**Worker** W_0	**Coordinator** C
51	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
52	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Sensors detect a large rock nearby containing a rare mineral. W_0's domain state is updated.	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
53	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) State evaluator detects an Opportunity event.	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
54	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) W_0 searches for a plan to exploit the Opportunity. Heartbeat	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
55	(A = *true* , CP = <*Travel...*>, CG = <*Investigate...*>) Plan found: <*Go to rock, analyse mineral, report results.*> The Opportunity generates a new goal, <*Investigate the interesting rock.*> W_0 sends a <*Notification of Opportunity*> message to C	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Heartbeat received. Updates worker state for W_0.
56	(A = *true* , CP = <*Go to...*>, CG = <*Investigate...*>) Scheduler begins executing new plan.	(A = *true* , CP = <*Go to...*>, CG = <*Investigate...*>) <*Notification of Opportunity*> message received from W_0, worker state is updated

Time	**Worker** W_0	**Coordinator** C
⋮	(A = *true* , CP = <*Go to...*>, CG = <*Investigate...*>) Occasional Heartbeats sent.	(A = *true* , CP = <*Go to...*>, CG = <*Investigate...*>) Heartbeats received.
67	(A = *true* , CP = *empty* , CG = <*Explore...*>) Plan execution complete. Send <*Task Completion*> message to C with results of analysis. Opportunity goal is popped from goal stack, new goal is <*Explore region north of lander.*>	(A = *true* , CP = <*Go to...*>, CG = <*Investigate...*>)
68	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Generate plan to achieve current goal, as before.	(A = *true* , CP = *empty* , CG = <*Explore...*>) <*Task Completion*> message received, W_0's state updated and data is archived for transmission to Earth.
⋮	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Occasional Heartbeats sent.	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Heartbeats received.

4.2.4 A Potential Cause Of Failure

This example steps through the interactions that occur when the worker robot believes its solar panel power input is low (the Sun is setting).

Time	**Worker** W_0	**Coordinator** C
79	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
80	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Internal sensors detect that W_0's power input is low. W_0's domain state is updated.	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
81	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) State evaluator detects a PCF event. Insufficient power to continue operation is estimated to occur in 25 minutes.	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
82	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) W_0 searches for a plan to prevent the PCF. Heartbeat	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>)
83	(A = *true* , CP = <*Travel...*>, CG = <*Begin...*>) Plan found: <*Prepare worker hardware for nighttime hibernation.*> This plan is expected to take 20 minutes to complete. The PCF generates a new goal, <*Begin nighttime hibernation.*> W_0 sends a <*Notification of PCF*> message to C	(A = *true* , CP = <*Travel...*>, CG = <*Explore...*>) Heartbeat received. Updates worker state for W_0.
84	(A = *true* , CP = <*Prepare...*>, CG = <*Begin...*>) Scheduler begins executing new plan.	(A = *true* , CP = <*Prepare...*>, CG = <*Begin...*>) <*Notification of PCF*> message received from W_0, worker state is updated
⋮	(A = *true* , CP = <*Prepare...*>, CG = <*Begin...*>) Occasional Heartbeats sent.	(A = *true* , CP = <*Prepare...*>, CG = <*Begin...*>) Heartbeats received.

Time	**Worker** W_0	**Coordinator** C
185	(A = *true* , CP = *empty* , CG = $<$*Begin*...$>$) Plan execution complete, took 21 minutes. Send $<$*Task Completion*$>$ message to C with results of analysis. PCF goal is popped from goal stack, worker goes into hibernation for the night.	(A = *true* , CP = $<$*Prepare*...$>$, CG = $<$*Begin*...$>$)
186	(A = *true* , CP = *empty* , CG = *empty*) Agent is asleep.	(A = *true* , CP = *empty* , CG = $<$*Begin*...$>$) $<$*Task Completion*$>$ message received, W_0's state updated.

Chapter 5

Implementation

5.1 Introduction

5.1.1 Simulator Basics

We have implemented a software simulation of a rescue scenario that we have used to test our model and evaluate its performance. Our simulation is based on the RoboCup Rescue simulator described previously, though with some differences.

A number of properties of the simulation are represented by integer values; such properties are commonly referred to as *points*. For example, civilians start the simulation with a certain number of *hit points*, which indicate the level of injury the civilian has sustained[1]. As well, each civilian has a number of *rescue points*, which count the amount of effort applied by the agents in the system towards rescuing that civilian. Rescue points are described further below.

Fires in the simulation have an associated number of *extinguish points*, which count the amount of effort applied towards extinguishing the specific fire. Extinguish points are described below.

The simulated world is divided into a number of *cells*. The size of a single cell is 0.5 metres by 0.5 metres. Cells are the smallest unit of position in the simulation, and fractional positions are not permitted. A single cell contains information about the terrain at that location (such as

[1]Note that hit points decrease as civilians become more injured, and an overall goal is to reduce the total loss of hit points by all civilians.

elevation), as well as a list of all the objects (agents, civilians, and fires) present in the cell.

The specifics of the simulation are given in a *scenario*. A scenario describes the starting conditions of the simulation, including the number, type, and properties of both agents and civilians. The rules of the simulation (such as the cost of communication or the number of rescue points required to complete a rescue) are also specified in the scenario. Appendix A provides details on running the simulator and creating new scenarios. Table A.2 lists all the configuration options available.

5.1.2 Agents and Actions

Our experiments include only one type of agent[2]. The agent can perform certain actions, described below.

Rescue actions allow an agent to spend its time rescuing a trapped civilian. The rescuing agent must be in a cell adjacent to the target civilian. Performing a Rescue action requires a certain amount of time and adds a certain number of rescue points to the civilian's rescue counter (as specified by the `RESCUE_COST` and `RESCUE_POINTS` configuration options). The scenario specification includes a rescue target for each civilian, which is the number of rescue points required for the civilian to be successfully rescued. If multiple agents perform Rescue actions on a single civilian simultaneously, their efforts are combined. Thus, collaboration will reduce the time required to complete the rescue.

Extinguish actions allow an agent to spend its time putting out fires. Agents performing this action can extinguish a small area of fire a number of cells away. The exact size and range of the extinguished area are specified by the `EXTINGUISH_RADIUS` and `EXTINGUISH_PROXIMITY` configuration options. Extinguish actions are similar to Rescue action: each action increments the target fire's extinguish points until the number of extinguish points exceeds a specific threshold, at which point the fire has been put out. Also like Rescue actions, multiple agents extinguishing the same cells will combine their efforts, causing the fire to be put out faster.

Move actions allow the agent to travel to an adjacent cell, provided the movement is legal. Illegal movement includes attempts to move outside of the world map's boundaries, to move into a cell that is on fire, or to move up or down a slope that exceeds the maximum slope allowed

[2]We made this simplification to allow more cases where the agent detecting the unexpected event ends up being the agent responsible for handling that event.

by the `MAX_SLOPE` option (this includes attempts to move through walls). The time required to complete a Move action depends on the specific terrain being moved through: travelling uphill requires more time than travelling across flat ground.

Only one of the above actions can be performed by an agent at a time. An agent cannot simultaneously move around a fire and attempt to extinguish the fire; however, the agent could alternate between performing single Move and Extinguish actions to achieve a similar effect.

5.2 Implementing the Simulator

5.2.1 Overview

The simulator is written in the Perl programming language. It is broken into a number of classes and modules, each of which performs a specific function within the simulation. The main components of the system are as follows.

1. The Real-Time Operating System and CPU simulators (RTOS/CPU simulators).
2. The World simulator.
3. The Agent simulators.

Each of these components is discussed in greater detail below. At the end of this chapter more detail is given on the specifics of the system.

5.2.2 Terminology

Because our implementation involves a simulation of the real world, it can become unclear whether a term is referring to a real-world concept or a simulated-world concept. In this chapter, if a term could apply to both worlds in a confusing manner, the word *simulation* will be prepended to indicate the concept should be considered only with respect to the simulated world, and the word *actual*[3] will be prepended to indicate the concept should be considered only for the real world.

[3]The word *real* could be used here as well, but might lead to confusion when used to differentiate simulation time from actual time (“real time”).

5.3 The Real-Time Core

The most fundamental part of the real-time simulation is the real-time core, which simulates the actual hardware and the real-time operating system that would be running on top of it. The modules and classes described in this section comprise this part of the system.

The `RT` module is a Perl module that maintains the simulation clock. It is also responsible for keeping a list of all the schedulers in the system (see below), and for telling each scheduler to execute any pending actions whenever the clock advances. Time in the simulation is measured in *ticks*, starting from 0. The number of ticks per simulation second is configurable.

As the simulation advances, the real-time core will advance the simulation clock automatically to the time of the next scheduled action, according to the schedulers the core is managing. That is, if the time is currently $t = 0$, and the next scheduled action on any scheduler is at $t = 435$, the core will advance the clock to time 435 and then sequentially execute each scheduler's scheduled actions for that time.

5.3.1 Classes

The RTOS and CPU are simulated using five classes. These classes simulate one CPU, its hardware, and a basic RTOS running on it, forming a *simulated computer*.

`RT::Scheduler`

This class simulates a real-time scheduler, scheduling tasks for a single CPU. The schedule consists of a list of actions to perform, accompanied by the time at which they are to be performed. An action in this context is an object reference and the name of the method to be called on that object.

To schedule an action, a call is made to the scheduler's `Schedule` or `ScheduleOn` methods. The first method takes a function name and a time, and inserts an entry into the schedule to cause that function to be called when the simulation clock is equal to that time. The second method takes an object and a method, as well as a time, and inserts an entry into the schedule to cause that object's method to be called when the simulation clock is equal to that time.

Once a scheduled action has been performed, the item is removed from the schedule.

`RT::Thread`

This class provides a mechanism for automatically rescheduling an action so that it runs repeatedly, and for accounting for the simulated time that action requires.

Because the program is executing in actual time, each action that an agent or another simulator takes must estimate how much simulation time would be required to perform that action. This time is used to delay the execution of the simulated thread, to account for the time that would be needed to complete the action.

Multiple threads can be created and attached to a single scheduler. To differentiate between them, each thread can be assigned a name when it is first created.

`RT::MASMessage`

This class simulates a wireless link between two simulated computers. It provides two methods, `Send`, and `Receive`.

`Send` sends a message to another simulated computer by first estimating the time required to send the message over the wireless link. If the destination computer is already receiving a message from another computer, the time needed is added to the delay caused by the messages already being sent. The result is that a message sent to another simulated computer will be delayed according to how long the message is and how many other messages are being sent to the other computer. Once the delay has been calculated a new action is scheduled on the receiving computer's scheduler to copy the message into the receiver's buffer after the computed delay.

`Receive` checks to see if a message is pending (waiting to be read) in the receiving computer's buffer. If there is a message, it is removed and returned. Otherwise the call returns an error that the caller can use to simulate blocking until a message is received.

`RT::Notifier`

This class simulates asynchronous events (such as hardware interrupts) and allows threads to check if a specific event has occurred. Asynchronous events are signalled with the `Signal` method, and their state (whether they have been signalled or not) can be checked with the `Wait` method. This class can track an arbitrary number of events; to differentiate between them, each event is assigned a name when it is first signalled. A thread wishing to check if a specific asyn-

chronous event has occurred simply calls `Wait` with the event's name as an argument. Calling `Wait` on a signalled event resets the event's state to "not signalled."

`RT::IPC`

This class simulates a mechanism for inter-process communication. Messages can only be sent between threads running on the same scheduler with this mechanism; the `RT::MASMessage` class handles messages sent to a thread on another scheduler.

The `RT::IPC` class provides two methods, `Send` and `Receive`, to facilitate this communication. Each instance of the class maintains a buffer for each thread in the simulated computer. Each buffer holds the messages that have been sent to its thread (via the `Send` method), but have not yet been received. A thread can call `Receive` to retrieve the next unread message. If there are no messages waiting to be received, the call returns an error that the caller can use to simulate blocking until a message is received.

Unlike the MASMessage messaging system, IPC messages take no time to deliver. However, this class provides a mechanism to delay the receipt of a message by the receiver. This ability is used, for example, when a plan has been generated. The generation of a plan takes a certain amount of simulated time: using this mechanism, the results of the planning can be delayed appropriately before they are received by the thread responsible for executing the plan.

5.4 World Simulator

The next major component of the simulation is the world simulator. This simulator comprises the map of the world, the list of all objects (agents, fires, etc.) in the world, the fire simulator, and the path planner.

5.4.1 The World Class

The World class maintains data structures that allow lookup of objects by name or location, maintains the *world map*, and is also responsible for drawing the map to the user interface. The world map is a data structure composed of a grid of cells that together represent the physical environment, including the present locations of all objects. The World class contains an

`RT::Scheduler` object (but none of the other RTOS classes) to provide a mechanism for scheduling upcoming world events.

This class also loads the scenario files specified in the configuration file, and creates the world and the objects (including agents) within it.

Scenarios

A scenario is composed of three parts. The first part is a configuration file that specifies all the configurable aspects of the simulation. Table A.2 describes the available configuration parameters.

The second part is a terrain map of the world, specified as a grey scale PNG file. Terrain maps are discussed in detail in Section A.3.3.

The final part of a scenario is the object specification file, which describes all the objects in the scenario. There are five types of objects available, each with its own properties. The object specification file syntax is described in Appendix A.

5.4.2 The Fire Simulator

The fire simulator is implemented in the `World::Fire` module. It simulates the spreading and extinguishing of fires, and is also responsible for detecting when an agent has caught on fire.

Fire in the simulation is modelled on a cell-by-cell basis. Periodically (based on the `BURNINATE_INTERVAL` option), the World object's scheduler executes a function in the Fire module. The function examines each cell that is currently in the "on fire" state and, for each of its neighbours that aren't in the "on fire" state, it adds the value of `BURNINATE_POINTS` to that cell's combustion level. Once a cell's combustion level exceeds the limit specified by `COMBUST_TARGET`, the Fire module marks it as "on fire." Any objects currently located within that cell are then notified of the state change, and can take any necessary actions. For example, a Civilian object will begin to lose hit points much more rapidly when it is on fire.

The fire simulation also handles Extinguish actions made by agents to put out fires. Cells subject to an extinguish action have their extinguish level increased by `EXTINGUISH_POINTS` for each Extinguish action. Once this level exceeds the `EXTINGUISH_TARGET` value, the cell's "on fire" state is cleared.

5.4.3 The Path Planner

The world simulator also contains a simple path planner. Given a map and starting and ending locations, the path planner will attempt to produce a path from the start to the end. If such a path exists it will return the sequence of cells to traverse along the path and the estimated cost of the movement at each step.

Each ModelWorker agent (described below) contains its own map of the world, which consists of all the information it has observed through its sensors. When a ModelWorker agent generates a path it does not have access to the true state of the world, only to its (possibly out-of-date) model. The path planner must therefore be able to generate paths using this outdated information.

5.5 Agent Simulators

The main portion of the simulation resides in the agent simulators. Each agent simulator is a subclass of the `Agent` class.

There are three types of agents in the simulator: Civilian agents, ModelWorker agents, and ModelCoordinator agents. The latter two types of agents are comprised of a simulated computer, a planner, a state evaluator, and a control loop. The Civilian agent is very simple and only contains a scheduler and a few callback functions.

5.5.1 ModelWorker Agents

ModelWorker agents implement the design and the algorithms for the worker agent described in Chapter 3. ModelWorker agents contain instances of the real-time simulation classes described in Section 5.3, as well as classes and modules that implement a control loop, a planner and plan library, sensors, and motors.

The Control Loop

The control loop is a thread that implements the algorithm described in Section 3.6.2 in Chapter 3.

Planning

Planning by ModelWorker agents is divided into three parts. The first part is a simple reactive planner. A full list of possible events is described in Section 5.7.4. When an event (Opportunity, Barrier, or PCF) is received from the control loop, the planner uses the event's properties (in particular, its type and number) to decide the appropriate response. Some events have only one response; for example, if a ModelWorker agent encounters Barrier *B1* ("Civilian has died"), the only response it can make is to give up its "Rescue Civilian" goal, since no other action will allow it to complete the goal.

However, some events present the agent with a choice of responses. In this case, the agent evaluates each possible response and selects the plan that resolves the event the fastest. For example, if the agent encounters PCF *P2* ("Fire is approaching the civilian"), it can either attempt to postpone its rescue actions and instead extinguish the fire, or relay the problem to the coordinator to handle. If the ModelWorker expects to extinguish the fire and complete the rescue by its deadline, it will choose the "Extinguish" plan; otherwise, the worker must relay the problem to the coordinator to deal with.

The second component of the planning process is a plan library consisting of six plans (each in its own Perl module), corresponding to the six possible goals the agent can attempt. These goals are outlined in Section 5.7.3. Each plan is further broken into three parts: initialization/generation, execution, and consistency checking.

When a plan is created, the initialization/generation portion of the plan is called, causing the plan's internal state to be initialized, and all information necessary to execute the plan to be collected, potentially including a call to the path planner. The plan is checked for basic errors as it is generated; if an error (such as Barrier) is found, the error is returned to the planner and plan generation is aborted. Otherwise the generation was successful and the plan may be executed if desired.

The execution portion of a plan is a large `if` block, decided based on the current step and sub-step of the plan. For example, step 1 of a plan may be to travel to a specific location. The sub-steps of step 1 would then be the individual movements necessary to reach that location. Step 2 might be to perform Rescue actions until the target civilian has been rescued. The execution portion of a plan automatically recognizes when a step or sub-step has been completed; the next call to the execution portion will result in the next step of the plan being executed.

The final portion of a plan in the plan library is consistency checking. This portion of the plan is generally called by the state evaluator (described in the next section), though it can be called any time the agent needs to verify that its state is still consistent with that required by the plan.

Consistency checking is also divided into parts based on the current step (and possibly substep) of the plan, since different steps will have different requirements for their execution. When the agent's state evaluator calls this portion of the plan, the agent's most recent local state information is compared with the state required for the plan. If part of the agent's local state is inconsistent with the plan's expected state, the appropriate Barrier or PCF is generated and returned to the caller. This is the essential mechanism by which Barriers and PCFs are recognized and dispatched by worker agents.

The last part of the planning process involves sending the newly generated and approved plan to the plan executor, a separate thread that is responsible for calling the plan's execution portion repeatedly until the plan is complete. The plan is sent to the plan executor via the delayed IPC mechanism described in Section 5.3.1.

Some parts of a plan's execution portion may require the agent to wait for an asynchronous event to occur. For example, once the agent's motors have been instructed to move the agent to the next location on a path, the plan must wait until the agent arrives at the new location before sending the next command. The plan executor allows a plan to "block" itself until the asynchronous event occurs, without preventing the planner from running.

Sensors

Periodically, the agent will receive updates from its sensors. The frequency of sensor updates is controlled by the `SENSOR_UPDATE_COST` option. A sensor update consists of the agent reading the most recent information within its visibility range from the world map data structure described in Section 5.4.1 and storing it in a buffer local to the agent.

Once this operation is complete the sensor hardware signals an asynchronous event, triggering the execution of the state evaluator thread. The state evaluator reads the updated map information from the buffer and merges it with the agent's local state. If a new Civilian is observed at this time, the state evaluator will generate an Opportunity event and relay it to the control loop (and thence to the planner). Likewise, if new terrain is observed, the state evaluator

will generate the appropriate Opportunity events and relay them to the control loop as well.

Because the sensors update relatively infrequently, it is possible that a condition that a plan is waiting for will occur, but will not be noticed by the agent for a short period of time.

Motors

A ModelWorker agent can move about the world by issuing commands to its motor simulators. Motor commands are simply directives to move to the next cell in a certain direction. Movement is simulated by first verifying that the agent can move into the desired map cell. If the cell contains fire, is outside the map, or requires a height change larger than that allowed by the `MAX_SLOPE` configuration parameter, the movement is aborted and a special flag is set indicating a collision has occurred. If the movement is allowed, the motor simulator computes the time required to travel to the new cell and registers an action for that time in the agent's scheduler. When the action occurs the movement is considered complete and the agent's position on the world map is updated to reflect the new state. In either case, the motors signal an asynchronous event when the agent stops moving.

5.5.2 ModelCoordinator

ModelCoordinator agents implement the design and the algorithms for the coordinator agent described in Chapter 3. Like ModelWorker agents, ModelCoordinator agents derive from the main Agent class and contain instances of the real-time simulation classes described in Section 5.3, as well as classes and modules that implement a control loop, a planner and a plan library. However, unlike the ModelWorker class, the ModelCoordinator has no motors or sensors. Instead it has classes to model each of its workers, and to help it select workers to perform tasks.

The Control Loop

The control loop is a thread that implements the algorithm described in Section 3.6.2 in Chapter 3.

Planning

Planning by ModelCoordinator agents is structured in the same manner as planning for ModelWorker agents. Like ModelWorker agents, the ModelCoordinator agent has a planner and a plan library. However, there are some key differences in how plans are generated, and how they are executed and monitored.

When the ModelCoordinator receives an event (such as a Barrier), it must construct a plan to resolve the problem posed by the event, including determining which workers are to be used to execute the new plan. Therefore, part of the ModelCoordinator's planning phase includes evaluating the candidate workers and selecting one or more to perform the task. In the implementation used in the ModelCoordinator, this step is accomplished by evaluating, for each possible solution to the event, the potential of each worker to perform the plan associated with the solution. If the ModelCoordinator expects that a worker could successfully complete a plan, the worker and the plan are added to a list of candidates. Once all possible solutions for the problem are examined and the list of candidate workers is built, the ModelCoordinator selects the worker with the earliest plan-completion date and proceeds to assign it the new task.

Because the ModelCoordinator does not actually execute the plans it generates, it does not have an equivalent of the ModelWorker's "plan executor." Instead, when a plan has been generated and assigned to a worker, it is bundled into a message and sent (via the `RT::MASMessage` mechanism) to the selected worker. The worker then unbundles the plan and, after a cleanup step to correct for small differences in global and local state models, begins execution of the plan.

Global State Representation

The ModelCoordinator agent receives updated state information from the various ModelWorker agents it controls. This information is processed by the agent's state evaluator, which splits the information into global and worker-specific information. Global information includes updated map information and new objects discovered. Worker-specific information includes the worker's current position and its current goal.

5.5.3 Civilian

The Civilian agent is a very simple Agent subclass that models the health of a civilian in the simulation using hit points, as mentioned in Section 5.1.1. The civilian starts with a given number of these points, as specified in the scenario configuration files (See Appendix A). A callback is used to periodically deduct hit points from the civilian's total. If the number of hit points a civilian has drops below zero, the civilian is "dead."

The Civilian agent is informed by the World's fire simulator when the cell it occupies is ignited. While the cell remains in the "on fire" state, the Civilian will lose hit points at a rapid rate.

ModelWorker agents that are performing a rescue action on the Civilian agent do so by calling the Civilian's `Rescue` function. Each rescue action increments the Civilian agent's rescue counter. Once the counter exceeds the threshold set by the civilian's difficulty (which is multiplied by the `RESCUE_TARGET` configuration parameter), the Civilian agent has been "rescued." Any ModelWorker agents performing rescue actions can detect this condition to determine when a rescue is complete.

5.6 Implementation Assumptions

To simplify the implementation of our model, we have made certain assumptions about the capabilities of the various agents:

- Messages (set via the `RT::MASMessage` mechanism) are sent over a reliable link. An unreliable link is more realistic, but adds unnecessary complexity to the simulation. If desired, one can use the link bandwidth (as specified by the `MAS_MESSAGE_COST` configuration option) to simulated the latency that results from an error correction system.

- All agents start with a map of the terrain only; the locations of objects (other agents and civilians) are initially unknown, and do not become known until the agent's sensors begin processing the information they receive from the world simulator.

- Worker agents can localize themselves and other objects automatically.

- Worker agents can see the area around them in a 20 cell (10 metre) radius, even through walls.
- Worker agents are invulnerable and cannot be injured or damaged.
- All agents can automatically determine how many hit points a civilian has left.
- In contrast to RoboCup Rescue, where each agent can only perform one type of action (Rescue, Extinguish, or Move), our worker agents all have the same capabilities. This assumption was made to simplify the planning step.
- Each agent has only one planner thread and thus can only generate one plan at a time. Our model is not restricted to a single planner thread; however, multiple planner threads increases the complexity of the implementation.
- Likewise, the coordinator agent uses a single planning thread, and has no plan executor thread to monitor the plan as it progresses. Instead, it relies on the worker assigned to the plan to monitor it for failures, and report them if they occur.
- When a worker completes the task assigned to it by the coordinator, the coordinator immediately restores the worker's autonomy, rather than hold the worker in reserve in case a new task occurs.
- The workers use a stack to track their current goals. The topmost goal on the stack is the goal the agent is actively working on. The coordinator models each worker's stack as part of its model of that worker. For example, worker Worker12's goal stack might look like this (refer to Section 5.7.3 for details on the meaning of *G1*, etc):

  ```
  G0
  G1, TARGET_CIVILIAN=Civ6, TARGET_LOCATION=<185, 56>
  G2, TARGET_LOCATION=<183, 68>
  ```

  The goal stack above indicates that Worker12 has three goals it is attempting to complete. The bottom-most goal, *G0*, is its mission goal, which dictates the constraints that the worker must operate under. The next goal is a suspended goal to rescue a civilian named Civ6, who can be found at the location given. Finally, the currently active goal, at

the top of the stack, is a goal to extinguish a fire that Worker12 has encountered on its way to Civ6's location.

The coordinator's model of that worker's stack would look very similar:

```
G0, WORKER=Worker12
G1, WORKER=Worker12, TARGET_CIVILIAN=Civ6, TARGET_LOCATION=<185, 56>
G2, WORKER=Worker12, TARGET_LOCATION=<183, 68>
```

- On occasion the coordinator will specify a path for a worker to follow based on stale information (that is, the worker has moved from the starting point of the path it is supposed to follow). In this case, the worker simply regenerates the first part of the path to correct for the error. This is an artifact of the simplified path planning method used, and so this extra computation is considered "free" (it does not count against the worker thread's CPU usage simulation).

5.7 Messages, Goals, and Events

5.7.1 Overview

This section provides a description of all the messages each agent can send in the system. It also lists all the goals each agent can attempt to accomplish, and all the events (divided into Opportunities, Barriers, and Potential Causes of Failure) an agent can encounter.

5.7.2 Messages

Agents in a multi-agent system must communicate to accomplish their tasks. In our system, the communication is very tightly specified, so that only certain agents can send certain messages. As well, workers can only send a message to the coordinator (and not to other workers), while the coordinator can send a message to any worker. The tables below list each message that workers and the coordinator can send, as well as its type and what sort of data the message includes. Note that all messages sent from a worker to the coordinator contain local state updates.

Table 5.1 lists all the messages that the coordinator can send to a worker.

NEW_MISSION	Assign a new mission goal to a worker
Contains:	Nothing.
Description:	This message causes the worker to instantiate a new Goal object to represent its mission. In other circumstances, this message could contain the new Goal object as generated by the coordinator, but as our implementation does not have multiple missions, this was not necessary.
REVOKE_AUTONOMY	Revoke the worker's autonomy
Contains:	A task (Goal an Plan objects) for the worker to begin executing.
Description:	This message causes the worker to lose its autonomy, set its current goal to the enclosed Goal object, and begin executing the plan included in the message. If a worker is already controlled, it can be assigned a new task with the NEW_TASK message, described below.
RESTORE_AUTONOMY	Restore the worker's autonomy
Contains:	Nothing.
Description:	This message causes the worker to drop any goals given to it by the coordinator (i.e., those goals marked as "not autonomous"). The worker will then validate the plan for the topmost autonomous goal on its goal stack and, assuming it is still valid, will resume executing it. If the plan is not valid, the appropriate Barrier or PCF will be raised and handled as usual.
NEW_TASK	Assign a new task to an already controlled worker
Contains:	A task (Goal and Plan objects) for the worker to begin executing.
Description:	This message is similar to the REVOKE_AUTONOMY message, except it is used when the worker is already controlled by the coordinator (REVOKE_AUTONOMY messages are sent to autonomous workers the coordinator wishes to begin controlling). The receiving worker will suspend its current task to work on the task given in the message.

RESUME_TASK	Instruct a controlled worker to resume a previously suspended task
Contains:	The identifier of the task to be resumed.
Description:	This message allows the coordinator to instruct a worker to stop working on a task it was assigned and resume executing a suspended task. The task to be resumed is specified in the message, and must be a task that the coordinator has already assigned to the worker with a `REVOKE_AUTONOMY` or `NEW_TASK` message.

Table 5.1: Messages the coordinator can send to a worker.

Table 5.2 lists all the messages that a worker can send to the coordinator.

HEARTBEAT	The worker is updating the coordinator on its current local state
Type:	Heartbeat
Contains:	State updates only.
Description:	This message is sent when no other messages have been sent by the worker for a specified period of time (see Table A.2).
REGISTER_AGENT	The worker has joined the coordinator's team
Type:	Notification
Contains:	The worker's name
Description:	This message is sent by each worker when it initializes itself, to inform the coordinator of its presence in the team. The coordinator should respond with a `NEW_MISSION` message.
GOAL_CHANGED	The worker has changed its current goal
Type:	Notification
Contains:	The worker's new goal.
Description:	This message is sent by a worker whenever its current goal changes, to inform the coordinator of the change. The coordinator uses this message to keep its model of the worker's goal state updated when the worker is autonomous.

TASK_COMPLETION	The worker has completed executing the plan for a goal it was working on
Type:	Notification
Contains:	The completed Goal object
Description:	This message is sent by a worker whenever it has finished executing a plan. If the worker is controlled by the coordinator, this message will usually provoke a response of either a NEW_TASK, RESUME_TASK, or RESTORE_AUTONOMY message from the coordinator. Otherwise, this message is simply used to keep the coordinator up-to-date on the worker's current goal.
UNHANDLED_OPPORTUNITY	The worker has encountered an Opportunity it can't exploit
Type:	Notification
Contains:	The Opportunity the worker could not exploit.
Description:	This message is sent by a worker whenever it determines that an Opportunity event is sufficiently interesting that it should be exploited, but that the worker cannot handle the event itself.
UNHANDLED_BARRIER	The worker has encountered a Barrier event it can't resolve
Type:	Panic
Contains:	The Barrier the worker could not resolve.
Description:	This message is sent by a worker whenever it cannot determine the solution to a Barrier event on its own.
UNHANDLED_PCF	The worker has encountered a PCF event it can't resolve
Type:	Panic
Contains:	The PCF the worker could not resolve.
Description:	This message is sent by a worker whenever it cannot determine the solution to a PCF event on its own.

Table 5.2: Messages a worker can send to the coordinator.

5.7.3 Goals

Based on an analysis of the capabilities of the workers and the coordinator in the implemented system, we produced a list of six goals that can arise during the course of the simulation. Within the simulation, these goals are referred to by number, as in *G2*, to indicate goal number two. Likewise, events (Barriers, Opportunities, and PCFs) are specified by a letter indicating their type followed by a number indicating the specific event. For example, *P3* indicates PCF number three, and *O1* indicates Opportunity number one. The specific Opportunities, Barriers, and PCFs that can arise during the simulation are described in the next section.

Goal 0	Mission goal
Possible Events:	*O1*, *O2*
Plan:	No plan.
Goal 1	Rescue a specific Civilian
Target:	The specific Civilian to be rescued.
Possible Events:	*O1*, *B1*, *B3*, *B5*, *P1*, *P2*, *P3*
Plan:	1. Travel to a cell adjacent to the target Civilian, 2. Perform Rescue actions on the target Civilian until it is rescued.
Goal 2	Extinguish a specific fire
Target:	The specific cell to be extinguished.
Possible Events:	*O1*, *B2*, *B3*, *B5*, *P3*, *P4*
Plan:	1. Travel to within `EXTINGUISH_PROXIMITY` cells of the target, 2. Perform Extinguish actions on the target cell until the fire is out.
Goal 3	Get another agent to rescue a Civilian
Target:	The specific Civilian to be rescued.
Possible Events:	*B4*
Note:	This goal is the coordinator's version of *G1*; whichever agent is assigned the new task will perform the plan as if the goal were *G1*, including checking for the events that can occur for that goal.
Coordinator Plan:	1. Select the worker to perform the task, 2. Send new task to the selected worker.

Worker Plan:	1. Travel to a cell adjacent to the target Civilian, 2. Perform Rescue actions on the target Civilian until it is rescued.
Goal 4	Get another agent to Extinguish a fire
Target:	The specific cell to be extinguished.
Possible Events:	*B4*
Note:	This goal is the coordinator's version of *G2*; whichever agent is assigned the new task will perform the plan as if the goal were *G2*, including checking for the events that can occur for that goal.
Coordinator Plan:	1. Select the worker to perform the task, 2. Send new task to the selected worker.
Worker Plan:	1. Travel to within `EXTINGUISH_PROXIMITY` cells of the target, 2. Perform Extinguish actions on the target cell until the fire is out.
Goal 5	Explore an unseen or out-of-date part of the map
Target:	The specific unexplored cell.
Possible Events:	*O1, B3, B5*
Plan:	1. Travel in the direction of the unexplored cells.

Table 5.3: Agent goal descriptions.

5.7.4 Events

The following table describes the possible events that can occur during the course of the simulation, and the goals or actions that will handle the event. For each event, a brief description of the event's meaning is provided, as well as an indication of under the conditions under which the event can occur. The possible responses, in the form of one or more goals or special actions that can deal with the event, are also given. Finally, each event is assigned a priority, which is used to break ties should multiple events occur at the same time. This priority system is used for simplicity and in lieu of a more complex mechanism to decide precedence between simultaneous events.

There are three special actions: *Ignore Event*, *Restart Goal*, and *Give Up Goal*. *Ignore Event* means the event is dropped (not handled) by the agent. *Restart Goal* means the plan for the

goal to which the event pertains should be regenerated to correct for an unexpected state. *Give Up Goal* means that the agent will no longer spend time performing the goal to which the event pertains; for example, event *B1* will cause the pertinent goal (likely a *G1* or *G3*) to be discarded, since further progress on the goal is no longer possible.

Opportunities

Opportunity 1	Found Civilian
Can Occur:	Any time.
Priority:	11
Responses:	*G1*, or *G3*, or *Ignore Event*
Opportunity 2	Found unexplored region of the map
Can Occur:	Any time.
Priority:	12
Responses:	*G5*, or *Ignore Event*

Table 5.4: Agent Opportunity event descriptions.

Barriers

Barrier 1	Civilian is dead.
Can Occur:	While performing, or about to perform, a Rescue action.
Priority:	1
Responses:	*Give Up Goal*
Barrier 2	Fire is extinguished.
Can Occur:	While performing, or about to perform, a Extinguish action.
Priority:	1
Responses:	*Give Up Goal*
Barrier 3	Path is blocked by fire during movement.
Can Occur:	While moving to a location.
Priority:	2

Responses:	*G2*, or *G4*, or *Give Up Goal*
Barrier 4	No agents are available to perform the task.
Can Occur:	While the coordinator is trying to assign a task to an agent.
Priority:	2
Responses:	*Ignore Event*, or *Give Up Goal*
Barrier 5	No path exists to target location.
Can Occur:	While generating a path to travel from point to point.
Priority:	3
Responses:	*G3* (if appropriate), or *G4* (if appropriate), or *Give Up Goal*
Barrier 6	Agent is not at expected location.
Can Occur:	When resuming a paused goal that requires the agent to be at a specific location.
Priority:	3
Responses:	*Restart Goal*

Table 5.5: Agent Barrier event descriptions.

Potential Causes of Failure

PCF 1	Civilian will likely die before rescue is complete.
Can Occur:	While performing, or about to perform, a Rescue action.
Priority:	1
Responses:	*G3*, or *Give Up Goal*
PCF 2	Fire is endangering a Civilian.
Can Occur:	While performing, or about to perform, a Rescue action.
Priority:	2
Responses:	*G2*, *G3*, *G4*, or *Ignore Event*
PCF 3	Will not arrive at location in time to complete rescue of target Civilian before the required deadline.
Can Occur:	While travelling to a location to perform Rescue actions.

Priority:	2
Responses:	*G3, Ignore Event,* or *Give Up Goal*
PCF 4	Will not arrive at location in time to complete extinguishing of fire before the required deadline.
Can Occur:	While travelling to a location to perform Extinguish actions.
Priority:	3
Responses:	*G4, Ignore Event,* or *Give Up Goal*

Table 5.6: Agent Potential Cause of Failure event descriptions.

Chapter 6

Experiments

6.1 Overview

6.1.1 Introduction

In this chapter we present two scenarios that demonstrate the value of our model for real-time multi-agent systems. Together, these scenarios cover all of the events and goals that our simulation can encounter.

Each scenario is first described and its initial conditions are given, followed by an annotated set of excerpts from the main log file (`master.log`) produced by the simulation run. The log excerpts give the details of each agent's actions, though, for clarity, the log excerpts have been edited to remove extraneous messages.

6.1.2 Scenario Log File Format

Each line of the log file is prefixed with the name of the agent or subsystem that generated it. Thus, lines starting with the string "RT-Core" are messages generated by the core real-time simulation, while lines starting with "Worker2" are messages generated by the agent Worker2.

Deadlines are printed as an integer in angle brackets (e.g. <34645>), while positions are printed as a pair of integers in angle brackets (e.g. <100, 30>). To avoid confusing two similar events caused by different goals or different agents, the simulator represents goals and events in

the form,

AgentName:*GoalA-EventA*:*GoalB-EventB*...

AgentName is the name of the agent that generated the goal or event (not the agent that is performing it!). *EventA* is an event the agent encountered while working on the goal *GoalA*. The agent's response to the event was to perform *GoalB*, which later encountered *EventB*, and so on. Thus, the context of each goal and event can be seen in the name of the object.

Note that some parts of the simulator depend on a random number generator, so different runs of the same scenario may produce slightly different sequences of events.

Details on creating a scenario or on running the simulator with a specific scenario can be found in Appendix A. For the details of each goal and event, please see Sections 5.7.3 and 5.7.4.

An Example

Below is an example log file excerpt. For this example, line numbers have been included for easy description of the excerpt; no line numbers are included in the actual log excerpts in the rest of the chapter:

```
 1   RT-Core: --------- TICK at time 3230
 2   Worker8: Hardware: Sensors updating... Worker8: done
 3   Worker8: STATE_EVALUATOR running
 4   Worker8: StateEvaluator: Merging updated sensor information into local state...Worker8: done
 5   World: Predicting extinguish of fire at <125, 27> will take 6520 ticks
 6   Civ2: Predicting death of Civ2 in 54200 ticks
 7   Worker8: PLANNER running
 8   Worker8: Generating plan 1: Rescue civilian Civ2 at <125, 25>.  Currently at <137, 20>
 9   Worker8: Plan 1 must be complete by <57430>
10   Worker8: Accepted the challenge of Worker8:G0-O1, using goal Worker8:G0-O1:G1
11   RT-Core: Agent Worker8 is sending a message to agent Coordinator.
12   Worker8: PLAN_EXECUTOR running
13   Worker8: Doing step 1.0
14   Worker8: Initiating movement to <138, 20>
```

In each excerpt, the first line will be logged by the real-time core simulation to indicate the current time. In this example, the time is 3230 ticks.

The second line is logged by an agent, Worker8, and indicates that that worker's hardware subsystem performed a sensor update at this time.

The third line indicates that Worker8 then began executing code in its `STATE_EVALUATOR` thread, and that the next set of log entries by Worker8 will be logged by that thread.

Line 4 is printed from inside this thread, and indicates that Worker8 spent some time merging the new information collected by the hardware sensors (in line 1) into the worker's local state information.

In line 5, we see that the worker made a request of the World simulator for its prediction for how much time would be necessary to extinguish the fire at map location <125, 27>. The World simulator has inserted a log message to indicate what its prediction is for this time.

Similarly, line 6 is printed by an civilian agent called Civ2, in response to a request from Worker8 about the time the civilian has until its death. The predicted time left before Civ2 dies is 54200 ticks.

In lines 7 through 11 we can see that Worker8 has stopped executing code in its `STATE_EVALUATOR` thread, and is now running its `PLANNER` thread. Worker8 generates a plan to accomplish goal *G1* (see Section 5.7.3 for an explanation of what this goal does). We can see that Worker8 is currently located at position <137, 20>, and must get to a map cell next to location <125, 25>, where Civ2 is located. We can also see in line 9 that Worker8 has determined that the rescue it intends to perform must be completed by time 57430.

Line 10 shows that Worker8 has determined it is capable of meeting that deadline and has decided to exploit the Opportunity ("`the challenge of Worker8:G0-O1`") by executing a plan to accomplish goal *G1* ("`using goal Worker8:G0-O1:G1`")

Line 11 indicates that Worker8 has sent an inter-agent message to the agent named "Coordinator" (via the `MASMessage` mechanism described in Section 5.3.1). Because the message contents are not human-readable, they are not printed. To determine what type of message was sent we would have to look ahead to when the Coordinator agent received and processed the message.

Finally, lines 12 through 14 show that Worker8 has stopped executing code in its `PLANNER` thread, and is now running its `PLAN_EXECUTOR` thread. In this new thread, Worker8 begins executing step 1.0 of a plan, which causes it to initiate movement towards the cell at location <138, 20>.

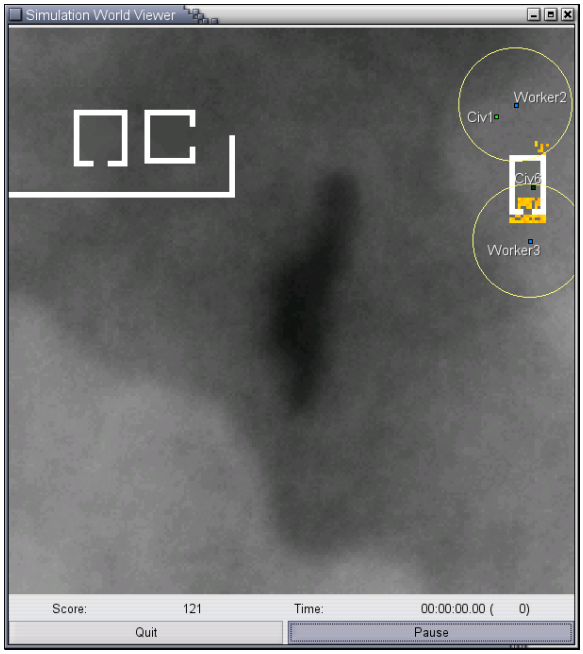

Figure 6.1: The starting state of scenario one, with Worker and Civilian agents labelled.

6.2 Scenario One

6.2.1 Background

In this scenario there are two Worker agents and two injured civilian agents, as shown in Figure 6.1 (see Section A.2.2 for an explanation of the simulator interface). Each Worker agent will initially detect only one of the two civilians, and proceed to rescue the particular civilian that it sees. However, due to various obstructions and deadlines, a number of interesting events occur as the simulation progresses. In particular, to successfully complete the rescue of civilian Civ6, Worker3 will require the aid of Worker2; otherwise, the rescue will not complete before Civ6 dies due to its injuries.

The next section examines the sequence of events that result from this scenario in detail.

6.2.2 Scenario Details

The very first action that each Worker agent performs (as part of its initialization) is to send a registration message to the Coordinator. This message enables the Coordinator to begin modelling the actions of the Worker, and lets it know that the Worker is available to perform tasks should the Coordinator find it necessary.

In response to the registration message, the Coordinator sends a `NEW_MISSION` message to the Worker:

```
RT-Core: --------- TICK at time 12
RT-Core: Agent Coordinator is sending a message to agent Worker2.
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker2): REGISTER_AGENT
Coordinator: Registered new worker Worker2
RT-Core: Agent Coordinator is sending a message to agent Worker3.
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker3): REGISTER_AGENT
Coordinator: Registered new worker Worker3
RT-Core: Agent Coordinator is sending a message to agent Worker2.
```

Upon receipt of the message, each Worker becomes fully activated and begins executing its mission, which in this case is to explore the terrain and rescue civilians:

```
RT-Core: --------- TICK at time 17
Worker2: CONTROL_LOOP running
Worker2: Got a NEW_MISSION message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker2: Starting new goal Worker2:G0
RT-Core: --------- TICK at time 22
Worker3: CONTROL_LOOP running
Worker3: Got a NEW_MISSION message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker3: Starting new goal Worker3:G0
```

Worker2 must spend some time to observe and process its surroundings. Once it has done so, the agent will recognize that there is a civilian (Civ1) within its field of view, and will generate an Opportunity event (*O1*) to rescue the civilian it sees. This excerpt of the log also shows the various threads in the agent running. The `CONTROL_LOOP` thread runs and receives an Opportunity event Inter-Process Communication (IPC) message, via the simulated IPC mechanism, from the `STATE_EVALUATOR` thread that ran previously. The `CONTROL_LOOP` then relays the event to the `PLANNER` thread.

The planner uses the information given in Section 5.7.4 to determine the possible actions to take in response. In this case, the planner first tries using the goal *G1* as the response. It generates a plan for the goal, tests it against the current and predicted state, and finds that the plan will succeed. The planner therefore selects the plan to achieve *G1* as the proper action to take:

```
RT-Core: --------- TICK at time 100
Worker2: Hardware: Sensors updating... Worker2: done
Worker2: STATE_EVALUATOR running
Worker2: StateEvaluator: Merging updated sensor information into local state...Worker2: done
Civ1: Predicting death of Civ1 in 465600 ticks
Worker2: Civ1's HP is 97
RT-Core: --------- TICK at time 105
Worker2: CONTROL_LOOP running
Worker2: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 133 new Opportunities
Worker2: PLANNER running
Worker2: Generating plan 1: Rescue civilian Civ1 at <172, 31>.  Currently at <179, 27>
Worker2: Plan 1 must be complete by <465700>
Civ1: Predicting rescue of Civ1 will take 36000 ticks with 1 rescuers
Worker2: Agent must get to <173, 31>
Worker2: Accepted the challenge of Worker2:G0-O1, using goal Worker2:G0-O1:G1
```

Likewise, Worker3 will observe a different civilian (Civ6) and decide to rescue it, in a manner similar to that described above:

```
RT-Core: --------- TICK at time 100
Worker3: Hardware: Sensors updating... Worker3: done
Worker3: STATE_EVALUATOR running
Worker3: StateEvaluator: Merging updated sensor information into local state...Worker3: done
Civ6: Predicting death of Civ6 in 115200 ticks
Worker2: Civ1's HP is 97
RT-Core: --------- TICK at time 105
Worker3: CONTROL_LOOP running
Worker3: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 107 new Opportunities
Worker3: PLANNER running
Worker3: Generating plan 1: Rescue civilian Civ6 at <185, 56>.  Currently at <184, 75>
Worker3: Plan 1 must be complete by <115300>
Civ6: Predicting rescue of Civ6 will take 96000 ticks with 1 rescuers
Worker3: Agent must get to <185, 57>
Worker3: Accepted the challenge of Worker3:G0-O1, using goal Worker3:G0-O1:G1
```

Once the plan has been finalized, Worker2 sends a Notification message to the Coordinator informing it of Worker2's new goal, and wakes up the `PLAN_EXECUTOR` thread, allowing it to start executing the current plan:

```
RT-Core: --------- TICK at time 237
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
Worker2: PLANNER running
Worker2: PLAN_EXECUTOR running
Worker2: Doing step 1.0
Worker2: Initiating movement to <178, 28>
```

Each time the sensors receive new information, the Workers check the newly observed state of the world against the state required for their plans:

```
RT-Core: --------- TICK at time 300
Worker2: Hardware: Sensors updating... Worker2: done
Worker2: STATE_EVALUATOR running
Worker2: StateEvaluator: Merging updated sensor information into local state...Worker2: done
Worker2: Testing agent state against that expected for step 1.0
Civ1: Predicting death of Civ1 in 465600 ticks
Worker2: Civ1's HP is 97
Worker2: Motors are 1 and agent is at <179, 27> (should be at <179, 27>)
```

Because Worker3's plan requires it to travel a longer path than Worker2, its planning step is correspondingly longer. Thus, Worker3 does not begin working on its plan until time 357, after Worker2 has already begun moving towards its target:

```
RT-Core: --------- TICK at time 357
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
Worker3: PLANNER running
Worker3: PLAN_EXECUTOR running
Worker3: Doing step 1.0
Worker3: Initiating movement to <185, 74>
```

When the Coordinator receives Worker2's `GOAL_CHANGED` Notification message, it incorporates the information contained in it (which includes updated map information) into its global state model. The log excerpt below also shows the Coordinator's model of Worker2's current goal stack. The "0" is the bottom-of-the-stack mission goal (*G0*) for Worker2. Above that goal is the newly acquired *G1* goal, generated in response to the observed Opportunity (*O1*). The Coordinator also knows that Worker2's *G1* is targeted at civilian Civ1, located at map cell $<172, 31>$:

```
RT-Core: --------- TICK at time 414
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 2 found objects, and 8 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker2): GOAL_CHANGED
Coordinator: Current goals for worker Worker2:
Coordinator:    0
Coordinator:    1!WORKER=Worker2!TARGET_CIVILIAN=Civ1!TARGET_LOCATION=<172, 31>
```

Shortly after the Coordinator receives the updated state information, Worker2 completes its first movement action, triggering an asynchronous event that awakens its `PLAN_EXECUTOR` thread. The `PLAN_EXECUTOR` thread then initiates the next movement action and begins waiting for it to complete (see Figure 6.2):

```
RT-Core: --------- TICK at time 449
Worker2: Agent Worker2 has arrived at its next stop, <178, 28>.
Worker2: PLAN_EXECUTOR running
Worker2: Doing step 1.1
Worker2: Initiating movement to <177, 29>
```

As each worker moves, new and unexplored cells of the map are discovered. Each cell generates an Opportunity event (*O2*), but because *O2* has a lower priority that the worker's current goal (*O1:G1*), these Opportunities are ignored:

```
RT-Core: --------- TICK at time 905
Worker2: CONTROL_LOOP running
Worker2: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 37 new Opportunities
Worker2: PLANNER running
Worker3: CONTROL_LOOP running
Worker3: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 34 new Opportunities
Worker3: PLANNER running
```

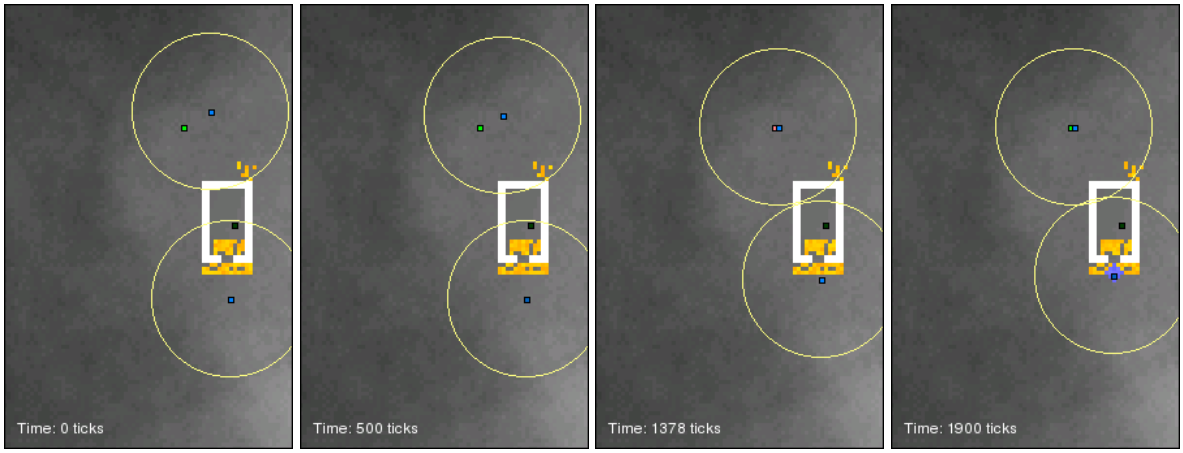

Figure 6.2: Scenario one at 0, 500, 1378, and 1900 ticks.

After a short time, Worker2 arrives at a cell adjacent to Civ1. When its PLAN_EXECUTOR next runs, it recognizes that the first phase of the plan is complete and begins executing the next phase, to rescue the civilian. Also shown below is one of Worker2's checks made to ensure deadlines will be met: Worker2 compares the expected time to rescue Civ1 with the expected time before Civ1 dies. In this case, Civ1 is expected to live a long time compared to the time required for the rescue to be complete, so Worker2 does not generate a PCF event:

```
RT-Core: --------- TICK at time 1378
Worker2: Agent Worker2 has arrived at its next stop, <173, 31>.
Worker2: PLAN_EXECUTOR running
Worker2: Doing step 2.0
RT-Core: --------- TICK at time 1400
Worker2: Hardware: Sensors updating... Worker2: done
Worker2: STATE_EVALUATOR running
Worker2: StateEvaluator: Merging updated sensor information into local state...Worker2: done
Worker2: Testing agent state against that expected for step 2.0
Civ1: Predicting death of Civ1 in 465600 ticks
Civ1: Predicting rescue of Civ1 will take 35400 ticks with 1 rescuers
Worker2: W_PLAN(Worker2:G0-O1:G1) is the active plan, checking for Barrier events
Worker2: Civ1's HP is 97
Worker2: Motors are 0 and agent is at <173, 31> (should be at <173, 31>)
```

However, Worker3's plan now appears to be in danger, as the fire is blocking the entrance to the building in which Civ6 is located. When Worker3 tries to move into the building, it finds its path blocked by the fire. The agent's hardware recognizes this condition and aborts the movement. When the PLAN_EXECUTOR is notified of the error state, it automatically suspends the affected goal (*G1*).

```
RT-Core: --------- TICK at time 1600
Worker3: Hardware: Sensors updating... Worker3: done
Worker3: STATE_EVALUATOR running
Worker3: StateEvaluator: Merging updated sensor information into local state...Worker3: done
Worker3: Testing agent state against that expected for step 1.6
Civ6: Predicting death of Civ6 in 115200 ticks
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Civ6's HP is 24
Worker3: Motors are 1 and agent is at <183, 69> (should be at <183, 69>)
RT-Core: --------- TICK at time 1624
Worker3: Agent Worker3 movement aborted, path is blocked at <183, 68>
Worker3: PLAN_EXECUTOR running
Worker3: Movement aborted, agent has encountered an obstacle
Worker3: Plan for goal Worker3:G0-O1:G1 is SUSPENDED
```

At this point, the agent does not know the cause of the problem, so no further action can be taken until the next sensor update:

```
RT-Core: --------- TICK at time 1700
Worker3: Hardware: Sensors updating... Worker3: done
Worker3: STATE_EVALUATOR running
Worker3: StateEvaluator: Merging updated sensor information into local state...Worker3: done
Worker3: Testing agent state against that expected for step 1.6
Civ6: Predicting death of Civ6 in 115200 ticks
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Generated new event Worker3:G0-O1:G1-B3
Worker3: Civ6's HP is 24
Worker3: Motors are 0 and agent is at <183, 69> (should be at <183, 69>)
Worker3: New event Worker3:G0-O1:G1-B3 found!
```

The `STATE_EVALUATOR` finds the cause of the problem (fire is blocking the path), and generates the appropriate Barrier event (*B3*). The new event is sent to the `CONTROL_LOOP` thread, which relays the IPC message to the appropriate thread to be handled (the `PLANNER` thread). The planner determines the best solution is to simply put out the fire, so a new goal is generated (*G2*) to perform this task:

```
RT-Core: --------- TICK at time 1705
Worker3: CONTROL_LOOP running
Worker3: Got a BARRIER_EVENT message from STATE_EVALUATOR, there are 1 new Barriers
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: Barrier, #3, by <15094>
Worker3: Barrier Worker3:G0-O1:G1-B3 has occurred
Worker3: Generating plan 2: Extinguish fire at at <183, 68>.  Currently at <183, 69>
Worker3: Plan 2 must be complete by <15094>
World: Predicting extinguish of fire at <183, 68> will take 4000 ticks
Worker3: Agent must get to <183, 69>
Worker3: Accepted the challenge of Worker3:G0-O1:G1-B3, using goal Worker3:G0-O1:G1-B3:G2
```

Once the plan is generated, Worker3 sends a `GOAL_CHANGED` Notification message to the Coordinator to inform it of Worker3's new state and then begins executing the goal. The number in brackets ("(0)") is the total accumulated extinguish points for the target cell just before the extinguish action occurs:

```
RT-Core: --------- TICK at time 1777
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
Worker3: PLANNER running
Worker3: PLAN_EXECUTOR running
Worker3: Doing step 2.0
Worker3: Extinguishing fire at <183, 68> (0)
```

At the next sensor update, we can see that Worker3 is continuing to check the observed state of the world against the state expected for each of its plans, including the suspended rescue plan:

```
RT-Core: --------- TICK at time 1800
Worker3: Hardware: Sensors updating... Worker3: done
Worker3: STATE_EVALUATOR running
Worker3: StateEvaluator: Merging updated sensor information into local state...Worker3: done
Worker3: Testing agent state against that expected for step 1.6
Civ6: Predicting death of Civ6 in 115200 ticks
Worker3: W_PLAN(Worker3:G0-O1:G1) is NOT the active plan, skipping Barrier event checks
Worker3: Testing agent state against that expected for step 2.0
World: Predicting extinguish of fire at <183, 68> will take 3400 ticks
```

At time 1934 the Coordinator receives the `GOAL_CHANGED` message sent by Worker3. The Coordinator adds the new goal to its model of what Worker3 is currently doing:

```
RT-Core: --------- TICK at time 1934
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 1 lost objects, 1 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker3): GOAL_CHANGED
Coordinator: Current goals for worker Worker3:
Coordinator:    0
Coordinator:    1!WORKER=Worker3!TARGET_CIVILIAN=Civ6!TARGET_LOCATION=<185, 56>
Coordinator:    2!WORKER=Worker3!TARGET_LOCATION=<183, 68>
```

In this scenario, Heartbeat messages are sent if 1500 ticks (five seconds) have elapsed since the last inter-agent message (including Notifications, Panics, and previous Heartbeats) was sent:

```
RT-Core: --------- TICK at time 4517
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
RT-Core: --------- TICK at time 4522
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

After some time, Worker3 performs a final Extinguish action on the fire. In this scenario, a fire must accumulate 10000 extinguish points to be considered put out:

```
RT-Core: --------- TICK at time 5377
Worker3: PLAN_EXECUTOR running
Worker3: Doing step 2.0
Worker3: Extinguishing fire at <183, 68> (9001)
```

The fire is now extinguished, so at the next sensor update the agent detects this change and generates an appropriate Barrier event, *B2*:

```
RT-Core: --------- TICK at time 5400
Worker3: Hardware: Sensors updating... Worker3: done
Worker3: STATE_EVALUATOR running
Worker3: StateEvaluator: Merging updated sensor information into local state...Worker3: done
Worker3: Testing agent state against that expected for step 1.6
Civ6: Predicting death of Civ6 in 115200 ticks
Worker3: W_PLAN(Worker3:G0-O1:G1) is NOT the active plan, skipping Barrier event checks
Worker3: Testing agent state against that expected for step 2.0
Worker3: Generated new event Worker3:G0-O1:G1-B3:G2-B2
Worker3: New event Worker3:G0-O1:G1-B3:G2-B2 found!
```

The Barrier causes Worker3 to remove the current goal (*G2*) from its goal stack and resume its previous goal instead. The worker also sends two Notifications messages to the Coordinator. The first is a `TASK_COMPLETE` message, and indicates that the previous task (*G2*) is now complete. The second is a `NEW_TASK` message that indicates the previous task that the worker has resumed working on:

```
RT-Core: --------- TICK at time 5405
Worker3: CONTROL_LOOP running
Worker3: Got a BARRIER_EVENT message from STATE_EVALUATOR, there are 1 new Barriers
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: Barrier, #2, by <15094>
Worker3: Barrier Worker3:G0-O1:G1-B3:G2-B2 has occurred
Worker3: Target fire is extinguished (Goal was Worker3:G0-O1:G1-B3:G2)
Worker3: Worker3:G0-O1:G1-B3:G2 versus Worker3:G0-O1:G1-B3:G2
Worker3: Giving up on goal Worker3:G0-O1:G1-B3:G2, dependant of Worker3:G0-O1:G1-B3:G2
Worker3: (clearing HANDLING_FAULTS flag)
Worker3: Resuming suspended goal Worker3:G0-O1:G1
Worker3: Plan for goal Worker3:G0-O1:G1 is UNSUSPENDED
Worker3: Testing agent state against that expected for step 1.6
Civ6: Predicting death of Civ6 in 115200 ticks
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Civ6's HP is 24
Worker3: Motors are 0 and agent is at <183, 69> (should be at <183, 69>)
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

When the Coordinator receives the `TASK_COMPLETE` message, it removes the finished task from its list of the worker's tasks:

```
RT-Core: --------- TICK at time 5657
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker3): TASK_COMPLETION
Coordinator: Current goals for worker Worker3:
Coordinator:    0
Coordinator:    1!WORKER=Worker3!TARGET_CIVILIAN=Civ6!TARGET_LOCATION=<185, 56>
Coordinator: (clearing HANDLING_FAULTS flag)
```

As Worker3 moves, new Opportunities arise to explore new parts of the map. However, the simple priority system we use ensures that the Opportunity to rescue a civilian takes precedence, so Worker3 ignores the new Opportunities.

```
RT-Core: --------- TICK at time 6205
Worker3: CONTROL_LOOP running
Worker3: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 37 new Opportunities
Worker3: PLANNER running
```

Worker3 continues to make its way towards Civ6, putting out fires as it moves. However, the time required to extinguish the fires cuts into the time available to complete the rescue of Civ6. At time 12100, Worker3 encounters another Barrier event (*B3*):

```
RT-Core: --------- TICK at time 12100
Worker3: Hardware: Sensors updating... Worker3: done
Worker3: STATE_EVALUATOR running
Worker3: StateEvaluator: Merging updated sensor information into local state...Worker3: done
Worker3: Testing agent state against that expected for step 1.11
Civ6: Predicting death of Civ6 in 100800 ticks
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Generated new event Worker3:G0-O1:G1-B3
Worker3: Civ6's HP is 21
Worker3: Motors are 0 and agent is at <184, 64> (should be at <184, 64>)
Worker3: New event Worker3:G0-O1:G1-B3 found!
```

However, when Worker3 attempts to generate a plan to resolve the Barrier, it finds that doing so would cause it to miss the deadline imposed by its earlier goal *G1*.

In the following log excerpt, Worker3 receives the Barrier event and attempts to generate the plan to extinguish the fire causing the Barrier (that is, the plan for goal *G2*). However, the new plan causes the estimated completion time for its earlier goal (*G1*) to be pushed back. The estimated completion time for *G1* is now later than the deadline by which *G1* must be completed. Thus, Worker3 cannot perform goal *G2* without violating the constraints imposed on it by *G1*. In our architecture, this situation requires that Worker3 suspend its actions, send a Panic message to the Coordinator, and await its response.

```
RT-Core: --------- TICK at time 12105
Worker3: CONTROL_LOOP running
Worker3: Got a BARRIER_EVENT message from STATE_EVALUATOR, there are 1 new Barriers
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: Barrier, #3, by <14848>
Worker3: Barrier Worker3:G0-O1:G1-B3 has occurred
Worker3: Generating plan 2: Extinguish fire at at <185, 62>.  Currently at <184, 64>
Worker3: Plan 2 must be complete by <14848>
World: Predicting extinguish of fire at <185, 62> will take 3999.2 ticks
Worker3: Agent must get to <184, 64>
Worker3: Generated new event Worker3:G0-O1:G1-B3:G2-P4
Worker3: Can't do anything about Worker3:G0-O1:G1-B3 using goal Worker3:G0-O1:G1-B3:G2,
         due to Worker3:G0-O1:G1-B3:G2-P4
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

A short time later the Coordinator receives the `UNHANDLED_BARRIER` Panic message from Worker3 and attempts to find a solution. The log excerpt below also shows the information

contained in the message, in a list of the form `FIELDNAME = value`. The first solution it tries is to send Worker2 to help rescue Civ6; however, this plan is rejected because Worker2 will not arrive at Civ6 in time to complete the rescue either[1].

Because no agent can successfully be sent to help rescue Civ6, the Coordinator falls back on a "default" plan to have Worker3 extinguish the fire regardless of the deadline faults:

```
RT-Core: --------- TICK at time 12489
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 1 lost objects, 1 found objects, and 23 cells whose On Fire state has changed
Coordinator: Got a PANIC message from MAS_MESSAGE_RECV (Agent Worker3): UNHANDLED_BARRIER
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: Barrier, #3, by <14848>
Coordinator: Worker3 has encountered Barrier Worker3:G0-O1:G1-B3
Coordinator:    GENERATING_AGENT = Worker3
Coordinator:    DEADLINE = <14848>
Coordinator:    CAUSE = Worker3:G0-O1:G1
Coordinator:    FULLNAME = Worker3:G0-O1:G1-B3
Coordinator:    SHORTNAME = B3
Coordinator:    TYPE = Barrier
Coordinator:    PRIORITY = 2
Coordinator:    NUMBER = 3
Coordinator:      ARGS{TARGET_LOCATION} = <185, 62>
Coordinator: Worker3's path is blocked by fire (Goal was Worker3:G0-O1:G1)
Coordinator: Trying goal 3...
Coordinator:    Trying worker Worker2...
Coordinator: Generating plan 3: Rescue civilian Civ6 at <185, 56>.  Currently at <173, 31>
Coordinator: Plan 3 must be complete by <14848>
Civ6: Worker2 registered as a rescuer (total=2)
Civ6: Predicting rescue of Civ6 will take 48000 ticks with 2 rescuers
Coordinator: Agent must get to <184, 56>
Civ6: Worker2 unregistered as a rescuer (total=1)
Coordinator:       Worker Worker2 can't accept the challenge of Worker3:G0-O1:G1-B3, because
                   of Worker3:G0-O1:G1-B3:G3-P3
Coordinator: No workers can perform this task
Coordinator: Generating plan 4: Extinguish fire at at <185, 62>.  Currently at <184, 64>
Coordinator: Plan 4 must be complete by <14848>
World: Predicting extinguish of fire at <185, 62> will take 3999.2 ticks
Coordinator: Agent must get to <184, 64>
Coordinator: Worker Worker3 will accept the challenge of Worker3:G0-O1:G1-B3, using
             goal Worker3:G0-O1:G1-B3:G4
```

Once the plan has been generated, a `REVOKE_AUTONOMY` message is sent to Worker3 containing the new goal and plan objects for it to perform, and the new goal is added to the Coordinator's model of Worker3's goal stack:

```
RT-Core: --------- TICK at time 13249
Coordinator: Preparing to revoke autonomy of Worker3, new task will be Worker3:G0-O1:G1-B3:G4
Coordinator: Revoking autonomy of agent Worker3
Coordinator: Sending first task to Worker3 and having it do goal Worker3:G0-O1:G1-B3:G4
```

[1]Worker2 will also have to put out the fire that is blocking Worker3's path. Note that a more sophisticated planner could likely produce a solution in which Worker3 clears a path while Worker2 travels to Civ6's location, but in our implementation this possibility is not considered.

```
RT-Core: Agent Coordinator is sending a message to agent Worker3.
Coordinator: Current goals for worker Worker3:
Coordinator:    0
Coordinator:    1!WORKER=Worker3!TARGET_CIVILIAN=Civ6!TARGET_LOCATION=<185, 56>
Coordinator:    4!WORKER=Worker3!TARGET_LOCATION=<185, 62>
Coordinator: PLANNER running
```

Worker3 receives the message and performs a quick check of the new plan against its current state. If there are any inconsistencies (such as Worker3 being in a different location than the plan expects), they are corrected. The new goal is marked as the active goal, and its plan is dispatched to the agent's PLAN_EXECUTOR thread, which begins executing the plan:

```
RT-Core: --------- TICK at time 13453
Worker3: CONTROL_LOOP running
Worker3: Got a REVOKE_AUTONOMY message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker3: Stopping plans and motors for agent and waiting for new plan
Worker3: Got goal Worker3:G0-O1:G1-B3:G4 from coordinator, generating plan...
Worker3:    GENERATING_AGENT = Coordinator
Worker3:    DEADLINE = <14848>
Worker3:    CAUSE = Worker3:G0-O1:G1-B3
Worker3:    FULLNAME = Worker3:G0-O1:G1-B3:G4
Worker3:    SHORTNAME = G4
Worker3:    PRIORITY = 2
Worker3:    NUMBER = 4
Worker3:    PLAN = W_PLAN(Worker3:G0-O1:G1-B3:G4)
Worker3:    OWNER = Worker3
Worker3:      ARGS{WORKER} = Worker3
Worker3:      ARGS{TARGET_LOCATION} = <185, 62>
Worker3: Generating plan 4: Extinguish fire at at <185, 62>.  Currently at <184, 64>
Worker3: Plan 4 must be complete by <14848>
World: Predicting extinguish of fire at <185, 62> will take 3999.2 ticks
Worker3: Agent must get to <184, 64>
Worker3: Starting new goal Worker3:G0-O1:G1-B3:G4
Worker3: PLAN_EXECUTOR running
Worker3: Doing step 2.0
Worker3: Extinguishing fire at <185, 62> (2)
```

Worker3 has now lost its autonomy and must spend its time executing the plan given to it by the Coordinator. Of course, as one would expect, Worker3 immediately determines that the current plan will cause its original plan (for *G1*) to miss its deadline. However, because Worker3 is not autonomous, it cannot handle the problem itself. Instead, Worker3 must send an UNHANDLED_PCF Panic message to the Coordinator:

```
RT-Core: --------- TICK at time 13505
Worker3: CONTROL_LOOP running
Worker3: Got a PCF_EVENT message from STATE_EVALUATOR, there are 1 new PCFs
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: PCF, #4, by <14848>
Worker3: PCF Worker3:G0-O1:G1-B3:G4-P4 has occurred
Worker3: Not autonomous, can't handle Worker3:G0-O1:G1-B3:G4-P4, relaying to coordinator
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

The Coordinator receives the Panic message and determines that, at least for the moment, none of the plans it has available can remedy the situation[2].

```
RT-Core: --------- TICK at time 13626
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a PANIC message from MAS_MESSAGE_RECV (Agent Worker3): UNHANDLED_PCF
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: PCF, #4, by <14848>
Coordinator: Worker3 has encountered PCF Worker3:G0-O1:G1-B3:G4-P4
Coordinator:    GENERATING_AGENT = Worker3
Coordinator:    DEADLINE = <14848>
Coordinator:    CAUSE = Worker3:G0-O1:G1-B3:G4
Coordinator:    FULLNAME = Worker3:G0-O1:G1-B3:G4-P4
Coordinator:    SHORTNAME = P4
Coordinator:    TYPE = PCF
Coordinator:    PRIORITY = 3
Coordinator:    NUMBER = 4
Coordinator:      ARGS{TARGET_LOCATION} = <185, 62>
Coordinator: Worker3 says it will not arrive in time to perform an extinguish
             (Goal was Worker3:G0-O1:G1-B3:G4)
```

When Worker3 extinguishes the fire, it detects the condition as a Barrier (*B2*) (because the fire was extinguished before the agent thought it would be) and, because it has no autonomy, relays the Barrier to the Coordinator as a Notification message:

```
RT-Core: --------- TICK at time 17105
Worker3: CONTROL_LOOP running
Worker3: Got a BARRIER_EVENT message from STATE_EVALUATOR, there are 1 new Barriers
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: Barrier, #2, by <14848>
Worker3: Barrier Worker3:G0-O1:G1-B3:G4-B2 has occurred
Worker3: Not autonomous, can't handle Worker3:G0-O1:G1-B3:G4-B2, relaying to coordinator
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

The Coordinator determines that the worker has finished extinguishing the fire and decides to tell the worker to stop working on goal *G4*. It removes the entry for the goal from its model of Worker3's goal stack and, because *G4* was the only goal that the Coordinator had given to Worker3, sends Worker3 a `REVOKE_AUTONOMY` message.

```
RT-Core: --------- TICK at time 17346
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 13 cells whose On Fire state has changed
Coordinator: Got a PANIC message from MAS_MESSAGE_RECV (Agent Worker3): UNHANDLED_BARRIER
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: Barrier, #2, by <14848>
```

[2]Once more, this behaviour is a result of the limited planner we are using. A more complex planner would likely produce a better solution.

```
Coordinator: Worker3 has encountered Barrier Worker3:G0-O1:G1-B3:G4-B2
Coordinator: Worker3's target fire is extinguished (Goal was Worker3:G0-O1:G1-B3:G4)
Coordinator: Current goals for worker Worker3:
Coordinator:    0
Coordinator:    1!WORKER=Worker3!TARGET_CIVILIAN=Civ6!TARGET_LOCATION=<185, 56>
Coordinator: Worker3 has completed goal Worker3:G0-O1:G1-B3:G4
Coordinator: No goals left on goal Coordinator goal stack, preparing to restore autonomy to Worker3
Coordinator: Restoring autonomy of agent Worker3
RT-Core: Agent Coordinator is sending a message to agent Worker3.
Coordinator: (clearing HANDLING_FAULTS flag)
```

Worker3 receives the message and removes the completed goal from its stack. It then resumes its last autonomous goal (which was *G1*). Finally, Worker3 notifies the Coordinator that it has changed goals with a `GOAL_CHANGED` Notification message.

```
RT-Core: --------- TICK at time 17353
Worker3: CONTROL_LOOP running
Worker3: Got a RESTORE_AUTONOMY message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker3: Restoring autonomy and restarting suspended plans for agent
Worker3: Giving up on goal Worker3:G0-O1:G1-B3:G4, not an autonomous goal
Worker3: (clearing HANDLING_FAULTS flag)
Worker3: Resuming suspended goal Worker3:G0-O1:G1
Worker3: Plan for goal Worker3:G0-O1:G1 is UNSUSPENDED
Worker3: Testing agent state against that expected for step 1.11
Civ6: Predicting death of Civ6 in 100800 ticks
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Civ6's HP is 21
Worker3: Motors are 0 and agent is at <184, 64> (should be at <184, 64>)
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

Worker3 encounters a similar fired-related Barrier at time 18405. However, just as it is completing the plan to extinguish the fire, its analysis of the state for its plan for goal *G1* determines that the Civilian is going to die before the rescue completes, causing it to generate a new PCF event, *P1*. Because Worker3 is still being controlled by the Coordinator, it relays the problem to that agent as a Panic message:

```
RT-Core: --------- TICK at time 24105
Worker3: CONTROL_LOOP running
Worker3: Got a PCF_EVENT message from STATE_EVALUATOR, there are 1 new PCFs
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: PCF, #1, by <120100>
Worker3: PCF Worker3:G0-O1:G1-P1 has occurred
Worker3: Not autonomous, can't handle Worker3:G0-O1:G1-P1, relaying to coordinator
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

The Coordinator attempts to resolve the PCF by assigning a second worker (Worker2) to help with the rescue. The excerpt below shows the details contained in the PCF event received by the Coordinator, including the deadline at time 120400 and the target civilian's name and location. By sending Worker2 to help Worker3, the Coordinator determines that the rescue can be

completed by the PCF's deadline, without also violating any constraints imposed by Worker2's current goals. Note that the Coordinator can be highly confident that its model of Worker2's current state (including its current goals) is accurate because Worker2 has been sending periodic Heartbeat messages to it, and has not yet sent a `TASK_COMPLETION` or `GOAL_CHANGED` message.

```
RT-Core: --------- TICK at time 24495
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a PANIC message from MAS_MESSAGE_RECV (Agent Worker3): UNHANDLED_PCF
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: PCF, #1, by <120400>
Coordinator: Worker3 has encountered PCF Worker3:G0-O1:G1-P1
Coordinator:    GENERATING_AGENT = Worker3
Coordinator:    DEADLINE = <120400>
Coordinator:    CAUSE = Worker3:G0-O1:G1
Coordinator:    FULLNAME = Worker3:G0-O1:G1-P1
Coordinator:    SHORTNAME = P1
Coordinator:    TYPE = PCF
Coordinator:    PRIORITY = 1
Coordinator:    NUMBER = 1
Coordinator:      ARGS{TARGET_CIVILIAN} = Civ6
Coordinator:      ARGS{TARGET_LOCATION} = <185, 56>
Coordinator: Worker3 expects civilian will die before rescue is complete (Goal was Worker3:G0-O1:G1)
Coordinator: Trying goal 3...
Coordinator:    Trying worker Worker2...
Coordinator: Generating plan 3: Rescue civilian Civ6 at <185, 56>.  Currently at <173, 31>
Coordinator: Plan 3 must be complete by <120400>
Civ6: Worker2 registered as a rescuer (total=2)
Civ6: Predicting rescue of Civ6 will take 48000 ticks with 2 rescuers
Coordinator: Agent must get to <185, 57>
Civ6: Worker2 unregistered as a rescuer (total=1)
Coordinator:       Worker Worker2 is a candidate for Worker3:G0-O1:G1-P1
Coordinator: Best candidate is Worker2
Civ6: Worker2 registered as a rescuer (total=2)
Coordinator: Worker Worker2 will accept the challenge of Worker3:G0-O1:G1-P1, using
             goal Worker3:G0-O1:G1-P1:G3
```

Meanwhile, having dealt with the PCF, Worker3 continues to execute the extinguish plan given to it earlier by the Coordinator. The next step of this plan (step 3.0) is to send a `TASK_COMPLETION` Notification message to the Coordinator:

```
RT-Core: --------- TICK at time 24548
Worker3: PLAN_EXECUTOR running
Worker3: Doing step 3.0
Worker3: Plan is complete!
Worker3: Worker3:G0-O1:G1-B3:G4 is complete
Worker3: Dropped Worker3:G0-O1:G1-B3:G4, new top-of-stack goal is Worker3:G0-O1:G1
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

Even while the Coordinator's planner is attempting to determine the best course of action[3],

[3]Note that, though the plan appears to already have been determined at time 24495, the simulator forces the planning code to wait an appropriate amount of time before its results can be sent to any other threads, to account for the time that would be required to perform the planning on a real system.

its CONTROL_LOOP thread receives the TASK_COMPLETION message sent to it by Worker3 and responds by removing the completed goal from its model of the Worker3's goal stack and then restoring that worker's autonomy with a RESTORE_AUTONOMY message:

```
RT-Core: --------- TICK at time 24626
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker3): TASK_COMPLETION
Coordinator: Current goals for worker Worker3:
Coordinator:    0
Coordinator:    1!WORKER=Worker3!TARGET_CIVILIAN=Civ6!TARGET_LOCATION=<185, 56>
Coordinator: Worker3 has completed goal Worker3:G0-O1:G1-B3:G4
Coordinator: No goals left on goal Coordinator goal stack, preparing to restore autonomy to Worker3
Coordinator: Restoring autonomy of agent Worker3
RT-Core: Agent Coordinator is sending a message to agent Worker3.
Coordinator: (clearing HANDLING_FAULTS flag)
```

Worker3 receives the message and once again resumes working on its original goal, *G1*:

```
RT-Core: --------- TICK at time 24633
Worker3: CONTROL_LOOP running
Worker3: Got a RESTORE_AUTONOMY message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker3: Restoring autonomy and restarting suspended plans for agent
Worker3: Resuming suspended goal Worker3:G0-O1:G1
Worker3: Plan for goal Worker3:G0-O1:G1 is UNSUSPENDED
Worker3: Testing agent state against that expected for step 1.14
Civ6: Predicting death of Civ6 in 96000 ticks
Worker3: Generated new event Worker3:G0-O1:G1-P1
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Civ6's HP is 20
Worker3: Motors are 0 and agent is at <185, 61> (should be at <185, 61>)
Worker3: Ignoring event Worker3:G0-O1:G1-P1, it's already being handled
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
Worker3: PLAN_EXECUTOR running
Worker3: Doing step 1.14
Worker3: Initiating movement to <186, 60>
```

Once planning is complete, the Coordinator adds the new goal to its model of Worker2's goal stack and sends a REVOKE_AUTONOMY message to Worker2, which at this time is still rescuing the original civilian it discovered, Civ1:

```
RT-Core: --------- TICK at time 25177
Coordinator: Preparing to revoke autonomy of Worker2, new task will be Worker3:G0-O1:G1-P1:G3
Coordinator: Revoking autonomy of agent Worker2
Coordinator: Sending first task to Worker2 and having it do goal Worker3:G0-O1:G1-P1:G3
RT-Core: Agent Coordinator is sending a message to agent Worker2.
Coordinator: Current goals for worker Worker2:
Coordinator:    0
Coordinator:    1!WORKER=Worker2!TARGET_CIVILIAN=Civ1!TARGET_LOCATION=<172, 31>
Coordinator:    3!WORKER=Worker2!TARGET_CIVILIAN=Civ6!TARGET_LOCATION=<185, 56>
```

Meanwhile, Worker3 has arrived at its destination and can now begin performing rescue actions on Civ6:

```
RT-Core: --------- TICK at time 25348
Worker3: Agent Worker3 has arrived at its next stop, <185, 57>.
Worker3: PLAN_EXECUTOR running
Worker3: Doing step 2.0
```

When Worker2 receives the Coordinator's message, it stops rescuing Civ1 and begins to move towards Civ6 using the plan given to it by the Coordinator:

```
RT-Core: --------- TICK at time 25390
Worker2: CONTROL_LOOP running
Worker2: Got a REVOKE_AUTONOMY message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker2: Stopping plans and motors for agent and waiting for new plan
Worker2: Plan for goal Worker2:G0-O1:G1 is SUSPENDED
Worker2: Got goal Worker3:G0-O1:G1-P1:G3 from coordinator, generating plan...
Worker2: Generating plan 3: Rescue civilian Civ6 at <185, 56>.  Currently at <173, 31>
Worker2: Plan 3 must be complete by <121300>
Civ6: Predicting rescue of Civ6 will take 47700 ticks with 2 rescuers
Worker2: Agent must get to <185, 57>
Worker2: Starting new goal Worker3:G0-O1:G1-P1:G3
```

As the simulation progresses, the fire slowly spreads. One of the dangers the workers must watch for is the proximity of the fire to their rescue targets. As Worker3 begins performing rescue actions, it predicts that a nearby fire is likely to reach Civ6's location before the rescue is complete. This prediction causes Worker3 to generate a new PCF, *P2*. Worker3 determines that it can handle the problem on its own if it spends some time extinguishing the fire at issue. After the planning completes, the worker sends a `GOAL_CHANGED` Notification message to the Coordinator and begins working on the new goal:

```
RT-Core: --------- TICK at time 25405
Worker3: CONTROL_LOOP running
Worker3: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 35 new Opportunities
Worker3: Got a PCF_EVENT message from STATE_EVALUATOR, there are 1 new PCFs
Worker3: PLANNER running
RT-Core: --------- TICK at time 25429
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: PCF, #2, by <68667>
Worker3: PCF Worker3:G0-O1:G1-P2 has occurred
Worker3: Generating plan 2: Extinguish fire at at <182, 58>.  Currently at <185, 57>
Worker3: Plan 2 must be complete by <68667>
World: Predicting extinguish of fire at <182, 58> will take 4000 ticks
Worker3: Agent must get to <185, 57>
Worker3: Accepted the challenge of Worker3:G0-O1:G1-P2, using goal Worker3:G0-O1:G1-P2:G2
RT-Core: --------- TICK at time 25501
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
Worker3: PLANNER running
```

When the fire is extinguished, Worker3 generates a Barrier event (*B2*), which causes it to discard its current goal. However, as the worker resumes rescuing Civ6, it detects another nearby fire that poses a threat to its rescue goal:

```
RT-Core: --------- TICK at time 29605
Worker3: CONTROL_LOOP running
Worker3: Got a BARRIER_EVENT message from STATE_EVALUATOR, there are 1 new Barriers
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: Barrier, #2, by <68667>
Worker3: Barrier Worker3:G0-O1:G1-P2:G2-B2 has occurred
Worker3: Target fire is extinguished (Goal was Worker3:G0-O1:G1-P2:G2)
Worker3: Worker3:G0-O1:G1-P2:G2 versus Worker3:G0-O1:G1-P2:G2
Worker3: Giving up on goal Worker3:G0-O1:G1-P2:G2, dependant of Worker3:G0-O1:G1-P2:G2
Worker3: (clearing HANDLING_FAULTS flag)
Worker3: Resuming suspended goal Worker3:G0-O1:G1
Worker3: Testing agent state against that expected for step 2.0
Civ6: Predicting death of Civ6 in 96000 ticks
Civ6: Predicting rescue of Civ6 will take 47700 ticks with 2 rescuers
Worker3: Closest fire to Civ6 is at <188, 58>, 3.60555127546399 cells away
World: Predicting combustion of <185, 56> in 43266.6153055679 ticks
Worker3: Civilian should be rescued by <77305>
Worker3: Generated new event Worker3:G0-O1:G1-P2
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Civ6's HP is 20
Worker3: Motors are 0 and agent is at <185, 57> (should be at <185, 57>)
Worker3: New event Worker3:G0-O1:G1-P2 found!
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

Worker3 cannot extinguish the fire because it can't get close enough to the fire's location to perform an extinguish action on it[4], so it must relay the problem to the Coordinator to handle. While it awaits the Coordinator's response, it will continue to perform rescue actions:

```
RT-Core: --------- TICK at time 29629
Worker2: PLANNER running
Worker3: PLANNER running
Worker3: CONTROL_LOOP running
Worker3: Got a PCF_EVENT message from PLANNER, there are 1 new PCFs
Worker3: PLANNER running
Worker3: Order of events is:
Worker3: Type: PCF, #2, by <72872>
Worker3: PCF Worker3:G0-O1:G1-P2 has occurred
Worker3: Generating plan 2: Extinguish fire at at <188, 58>.  Currently at <185, 57>
Worker3: Plan 2 must be complete by <72872>
Worker3: Generated new event Worker3:G0-O1:G1-P2:G2-B5
Worker3: Can't do anything about Worker3:G0-O1:G1-P2 using goal Worker3:G0-O1:G1-P2:G2,
         due to Worker3:G0-O1:G1-P2:G2-B5
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

The Coordinator gets the message and attempts to find a solution to the PCF:

```
RT-Core: --------- TICK at time 29992
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a PANIC message from MAS_MESSAGE_RECV (Agent Worker3): UNHANDLED_PCF
```

[4]In fact, the worker is already close enough to the fire; however, the fire is actually in a cell that is marked as a "wall" on the map, and so there is no path that will take the worker to the target location. This is a limitation of our path planning system; a better path planner would recognize that the agent only needs to get "close enough" to the target location to perform the action.

```
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: PCF, #2, by <72872>
Coordinator: Worker3 has encountered PCF Worker3:G0-O1:G1-P2
Coordinator:    GENERATING_AGENT = Worker3
Coordinator:    DEADLINE = <72872>
Coordinator:    CAUSE = Worker3:G0-O1:G1
Coordinator:    FULLNAME = Worker3:G0-O1:G1-P2
Coordinator:    SHORTNAME = P2
Coordinator:    TYPE = PCF
Coordinator:    PRIORITY = 2
Coordinator:    NUMBER = 2
Coordinator:      ARGS{TARGET_CIVILIAN} = Civ6
Coordinator:      ARGS{CIVILIAN_LOCATION} = <185, 56>
Coordinator:      ARGS{TARGET_LOCATION} = <188, 58>
Coordinator: Worker3 says fire is approaching its civilian (Goal was Worker3:G0-O1:G1)
```

First, it attempts to send another worker to help perform the rescue. However, it finds that Worker2, the only other worker in the scenario, is already performing a more important task: Worker2 is on its way to rescue Civ6, and that goal has a higher priority than the new goal would[5]. See Section 5.7.4 for the priorities assigned to each event.

```
Coordinator: Trying goal 3...
Coordinator:    Trying worker Worker2...
Coordinator:       Worker Worker2 can't accept the challenge of Worker3:G0-O1:G1-P2, because
Coordinator:          Event's priority is 2, but worker's goal is 1
```

The Coordinator then attempts to find workers than could extinguish the fire. First it checks whether Worker3 can perform the task, and finds (as Worker3 itself did) that Worker3 cannot reach the location of the fire to perform the extinguish action. When the Coordinator checks whether Worker2 can perform the action, it again finds that Worker2 is already performing a more important action. The Coordinator has no plans available to deal with the situation, and so is forced to ignore the problem. Since *Ignore Event* is a possible response to PCF *P2*, this response is allowed. The Coordinator does not send a message to Worker3 in this case; Worker3 will simply continue to perform rescue actions and operate autonomously, as it did before it found the PCF.

```
Coordinator: Trying goal 4...
Coordinator:    Trying worker Worker3...
Coordinator: Generating plan 4: Extinguish fire at at <188, 58>.  Currently at <185, 57>
Coordinator: Plan 4 must be complete by <72872>
Coordinator: Generated new event Worker3:G0-O1:G1-P2:G4-B5
Coordinator:       Worker Worker3 can't accept the challenge of Worker3:G0-O1:G1-P2,
```

[5]Goals inherit their priority from the event that triggered them, so even though Worker2's current goal is the same as the Coordinator's proposed goal (both are a *G2*), the former was caused by a PCF with a higher priority than the latter.

```
                    because of Worker3:G0-O1:G1-P2:G4-B5
Coordinator:    Trying worker Worker2...
Coordinator:       Worker Worker2 can't accept the challenge of Worker3:G0-O1:G1-P2, because
Coordinator:          Event's priority is 2, but worker's goal is 1
Coordinator: No workers can perform this task
```

On its way to aide Worker3, Worker2 frequently finds that fire has blocked its movement forward. Because it is not autonomous, the Barrier events that it generates (*B3*) are relayed to the Coordinator. The Coordinator examines its global state each time it receives one of these events and determines that Worker2 should extinguish the fire in its path itself. These Barriers occur at times 31674 and 40004. Below is an excerpt from the Worker's processing of the latter:

```
RT-Core: --------- TICK at time 40105
Worker2: CONTROL_LOOP running
Worker2: Got a BARRIER_EVENT message from STATE_EVALUATOR, there are 1 new Barriers
Worker2: PLANNER running
Worker2: Order of events is:
Worker2: Type: Barrier, #3, by <71466>
Worker2: Barrier Worker3:G0-O1:G1-P1:G3-B3 has occurred
Worker2: Not autonomous, can't handle Worker3:G0-O1:G1-P1:G3-B3, relaying to coordinator
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
```

Here is the Coordinator's response:

```
RT-Core: --------- TICK at time 40313
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 1 lost objects, 1 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a PANIC message from MAS_MESSAGE_RECV (Agent Worker2): UNHANDLED_BARRIER
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: Barrier, #3, by <71466>
Coordinator: Worker2 has encountered Barrier Worker3:G0-O1:G1-P1:G3-B3
Coordinator:    GENERATING_AGENT = Worker2
Coordinator:    DEADLINE = <71466>
Coordinator:    CAUSE = Worker3:G0-O1:G1-P1:G3
Coordinator:    FULLNAME = Worker3:G0-O1:G1-P1:G3-B3
Coordinator:    SHORTNAME = B3
Coordinator:    TYPE = Barrier
Coordinator:    PRIORITY = 2
Coordinator:    NUMBER = 3
Coordinator:      ARGS{TARGET_LOCATION} = <184, 64>
Coordinator: Worker2's path is blocked by fire (Goal was Worker3:G0-O1:G1-P1:G3)
Coordinator: Trying goal 3...
Coordinator:    Trying worker Worker3...
Coordinator: Generating plan 3: Rescue civilian Civ6 at <185, 56>.  Currently at <185, 57>
Coordinator: Plan 3 must be complete by <71466>
Civ6: Worker3 registered as a rescuer (total=2)
Civ6: Predicting rescue of Civ6 will take 42600 ticks with 2 rescuers
Coordinator: Agent must get to <185, 57>
Civ6: Worker3 unregistered as a rescuer (total=2)
Coordinator:       Worker Worker3 can't accept the challenge of Worker3:G0-O1:G1-P1:G3-B3,
                   because of Worker3:G0-O1:G1-P1:G3-B3:G3-P1
Coordinator: No workers can perform this task
Coordinator: Generating plan 4: Extinguish fire at at <184, 64>.  Currently at <184, 66>
Coordinator: Plan 4 must be complete by <71466>
World: Predicting extinguish of fire at <184, 64> will take 3998.4 ticks
```

```
Coordinator: Agent must get to <184, 66>
Coordinator: Worker Worker2 will accept the challenge of Worker3:G0-O1:G1-P1:G3-B3, using
             goal Worker3:G0-O1:G1-P1:G3-B3:G4
RT-Core: --------- TICK at time 40433
Coordinator: Preparing to send new task Worker3:G0-O1:G1-P1:G3-B3:G4 to (already controlled) Worker2
Coordinator: Sending next task to Worker2 and having it do goal Worker3:G0-O1:G1-P1:G3-B3:G4
RT-Core: Agent Coordinator is sending a message to agent Worker2.
Coordinator: Current goals for worker Worker2:
Coordinator:    0
Coordinator:    1!WORKER=Worker2!TARGET_CIVILIAN=Civ1!TARGET_LOCATION=<172, 31>
Coordinator:    3!WORKER=Worker2!TARGET_CIVILIAN=Civ6!TARGET_LOCATION=<185, 56>
Coordinator:    4!WORKER=Worker2!TARGET_LOCATION=<184, 64>
Coordinator: PLANNER running
```

Eventually, Worker2 puts out the fires and sends a `TASK_COMPLETION` Notification message to the Coordinator. The Coordinator drops the completed goal and sends the worker a `RESUME_TASK` message, telling it to resume working on the previous goal the Coordinator had sent it (that is, *G3*):

```
RT-Core: --------- TICK at time 45522
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker2): TASK_COMPLETION
Coordinator: Current goals for worker Worker2:
Coordinator:    0
Coordinator:    1!WORKER=Worker2!TARGET_CIVILIAN=Civ1!TARGET_LOCATION=<172, 31>
Coordinator:    3!WORKER=Worker2!TARGET_CIVILIAN=Civ6!TARGET_LOCATION=<185, 56>
Coordinator: Worker2 has completed goal Worker3:G0-O1:G1-P1:G3-B3:G4
Coordinator: Dropped Worker3:G0-O1:G1-P1:G3-B3:G4, new top-of-stack goal is Worker3:G0-O1:G1-P1:G3
Coordinator: Resuming old goal for agent Worker2
```

Once Worker2 makes it past the various fires, it arrives at Civ6's location and begins helping Worker3 perform rescue actions:

```
RT-Core: --------- TICK at time 47201
Worker2: Agent Worker2 has arrived at its next stop, <185, 57>.
Worker2: PLAN_EXECUTOR running
Worker2: Doing step 2.0
```

The nearby fire that was previously inaccessible eventually spreads to an area that the workers can reach. When it does, Worker2 detects the condition and, because it is not autonomous, informs the Coordinator.

```
RT-Core: --------- TICK at time 47329
Worker2: PLANNER running
Worker2: Order of events is:
Worker2: Type: PCF, #2, by <81242>
Worker2: PCF Worker3:G0-O1:G1-P1:G3-P2 has occurred
Worker2: Not autonomous, can't handle Worker3:G0-O1:G1-P1:G3-P2, relaying to coordinator
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
```

The Coordinator attempts to find a solution to the problem once more, and this time is able to assign Worker3 to perform the extinguish action on the fire:

```
RT-Core: --------- TICK at time 47883
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 1 lost objects, 1 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a PANIC message from MAS_MESSAGE_RECV (Agent Worker2): UNHANDLED_PCF
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: PCF, #2, by <81242>
Coordinator: Worker2 has encountered PCF Worker3:G0-O1:G1-P1:G3-P2
Coordinator:    GENERATING_AGENT = Worker2
Coordinator:    DEADLINE = <81242>
Coordinator:    CAUSE = Worker3:G0-O1:G1-P1:G3
Coordinator:    FULLNAME = Worker3:G0-O1:G1-P1:G3-P2
Coordinator:    SHORTNAME = P2
Coordinator:    TYPE = PCF
Coordinator:    PRIORITY = 2
Coordinator:    NUMBER = 2
Coordinator:      ARGS{TARGET_CIVILIAN} = Civ6
Coordinator:      ARGS{CIVILIAN_LOCATION} = <185, 56>
Coordinator:      ARGS{TARGET_LOCATION} = <187, 58>
Coordinator: Worker2 says fire is approaching its civilian (Goal was Worker3:G0-O1:G1-P1:G3)
Coordinator: Trying goal 3...
Coordinator:    Trying worker Worker3...
Coordinator: Generating plan 3: Rescue civilian Civ6 at <185, 56>.  Currently at <185, 57>
Coordinator: Plan 3 must be complete by <81242>
Civ6: Worker3 registered as a rescuer (total=2)
Civ6: Predicting rescue of Civ6 will take 38100 ticks with 2 rescuers
Coordinator: Agent must get to <185, 57>
Civ6: Worker3 unregistered as a rescuer (total=2)
Coordinator:       Worker Worker3 can't accept the challenge of Worker3:G0-O1:G1-P1:G3-P2,
                   because of Worker3:G0-O1:G1-P1:G3-P2:G3-P1
Coordinator: Trying goal 4...
Coordinator:    Trying worker Worker3...
Coordinator: Generating plan 4: Extinguish fire at at <187, 58>.  Currently at <185, 57>
Coordinator: Plan 4 must be complete by <81242>
World: Predicting extinguish of fire at <187, 58> will take 3998.8 ticks
Coordinator: Agent must get to <185, 57>
Coordinator:       Worker Worker3 is a candidate for Worker3:G0-O1:G1-P1:G3-P2
Coordinator:    Trying worker Worker2...
Coordinator:       Worker Worker2 can't accept the challenge of Worker3:G0-O1:G1-P1:G3-P2, because
Coordinator:          Equal priorities, but event's deadline is <81242>, but worker's
                      deadline is <71466>
Coordinator: Best candidate is Worker3
Coordinator: Worker Worker3 will accept the challenge of Worker3:G0-O1:G1-P1:G3-P2, using
             goal Worker3:G0-O1:G1-P1:G3-P2:G4
```

When it receives the `REVOKE_AUTONOMY` message, Worker3 will temporarily lose its autonomy and stop performing rescue actions on Civ6 to put out the fire. Once the fire is out it will send either a Barrier event (*B3*), or a `TASK_COMPLETION` Notification message, depending on whether the fire is extinguished at the time predicted or not[6]. Whichever message is sent, the

[6]If the fire is extinguished earlier than expected, then the Barrier event will occur, otherwise the agent will treat the disappearance of the fire as the expected result of its plan and send the Notification message.

Coordinator will in turn send Worker3 a RESTORE_AUTONOMY message, allowing it to resume rescuing Civ6 with Worker2:

```
RT-Core: --------- TICK at time 52619
Worker3: CONTROL_LOOP running
Worker3: Got a RESTORE_AUTONOMY message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker3: Restoring autonomy and restarting suspended plans for agent
Worker3: Giving up on goal Worker3:G0-O1:G1-P1:G3-P2:G4, not an autonomous goal
Worker3: (clearing HANDLING_FAULTS flag)
Worker3: Resuming suspended goal Worker3:G0-O1:G1
Worker3: Plan for goal Worker3:G0-O1:G1 is UNSUSPENDED
Worker3: Testing agent state against that expected for step 2.0
Civ6: Predicting death of Civ6 in 76800 ticks
Civ6: Predicting rescue of Civ6 will take 35400 ticks with 2 rescuers
Worker3: Closest fire to Civ6 is at <188, 56>, 3 cells away
World: Predicting combustion of <185, 56> in 36000 ticks
Worker3: Civilian should be rescued by <88019>
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Civ6's HP is 16
Worker3: Motors are 0 and agent is at <185, 57> (should be at <185, 57>)
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

Worker2's STATE_EVALUATOR's analysis of the situation now shows that the rescue should be completed before any more dangers approach, as illustrated in the log excerpt below. Civ6 should be rescued by time 88100, before Civ6 dies and before the nearest fire reaches its position at time 89600:

```
RT-Core: --------- TICK at time 53600
Worker2: Hardware: Sensors updating... Worker2: done
Worker2: STATE_EVALUATOR running
Worker2: StateEvaluator: Merging updated sensor information into local state...Worker2: done
Worker2: Testing agent state against that expected for step 2.0
Civ1: Predicting death of Civ1 in 412800 ticks
Civ1: Predicting rescue of Civ1 will take 11400 ticks with 1 rescuers
Worker2: W_PLAN(Worker2:G0-O1:G1) is NOT the active plan, skipping Barrier event checks
Worker2: Testing agent state against that expected for step 2.0
Civ6: Predicting death of Civ6 in 76800 ticks
Civ6: Predicting rescue of Civ6 will take 34500 ticks with 2 rescuers
Worker2: Closest fire to Civ6 is at <188, 56>, 3 cells away
World: Predicting combustion of <185, 56> in 36000 ticks
Worker2: Fire will reach civilian by <89600>
Worker2: Civilian should be rescued by <88100>
```

Note that, as the simulation progresses, both worker agents are continually adjusting their predictions for when their various tasks will be complete and when the various deadlines will occur:

```
RT-Core: --------- TICK at time 72600
Worker2: Hardware: Sensors updating... Worker2: done
Worker2: STATE_EVALUATOR running
Worker2: StateEvaluator: Merging updated sensor information into local state...Worker2: done
Worker2: Testing agent state against that expected for step 2.0
Civ1: Predicting death of Civ1 in 384000 ticks
Civ1: Predicting rescue of Civ1 will take 11400 ticks with 1 rescuers
```

```
Worker2: W_PLAN(Worker2:G0-O1:G1) is NOT the active plan, skipping Barrier event checks
Worker2: Testing agent state against that expected for step 2.0
Civ6: Predicting death of Civ6 in 52800 ticks
Civ6: Predicting rescue of Civ6 will take 15600 ticks with 2 rescuers
Worker2: Closest fire to Civ6 is at <184, 58>, 2.23606797749979 cells away
World: Predicting combustion of <185, 56> in 26832.8157299975 ticks
Worker2: Fire will reach civilian by <99433>
Worker2: Civilian should be rescued by <88200>
```

Both workers are also sending Heartbeat messages to the Coordinator, keeping it up-to-date on their progress:

```
RT-Core: --------- TICK at time 85517
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
RT-Core: --------- TICK at time 85522
RT-Core: Agent Worker3 is sending a message to agent Coordinator.
```

At time 88001 Worker2 performs the final rescue action on Civ6. At this time, Civ6's rescue is complete. We can see this by examining the next sensor update. At time 88100, Worker2 tests the observed state against that expected for step 3.0 of its plan. Step 3.0 is the final "clean up" phase of the plan for goals *G1* and *G3*, indicating that Worker2 has already detected that the rescue operation is finished. Note, however, that Worker3 is still working on step 2.0 of the plan, since it has not yet detected that the rescue operation is finished:

```
RT-Core: --------- TICK at time 88001
Worker2: PLAN_EXECUTOR running
Worker2: Doing step 2.0
RT-Core: --------- TICK at time 88100
Worker2: Hardware: Sensors updating... Worker2: done
Worker2: STATE_EVALUATOR running
Worker2: StateEvaluator: Merging updated sensor information into local state...Worker2: done
Worker2: Testing agent state against that expected for step 2.0
Civ1: Predicting death of Civ1 in 369600 ticks
Civ1: Predicting rescue of Civ1 will take 11400 ticks with 1 rescuers
Worker2: W_PLAN(Worker2:G0-O1:G1) is NOT the active plan, skipping Barrier event checks
Worker2: Testing agent state against that expected for step 3.0
Civ6: Predicting death of Civ6 in 48000 ticks
Worker3: Hardware: Sensors updating... Worker3: done
Worker3: STATE_EVALUATOR running
Worker3: StateEvaluator: Merging updated sensor information into local state...Worker3: done
Worker3: Testing agent state against that expected for step 2.0
Civ6: Predicting death of Civ6 in 48000 ticks
Civ6: Predicting rescue of Civ6 will take 0 ticks with 2 rescuers
Worker3: Closest fire to Civ6 is at <184, 57>, 1.4142135623731 cells away
World: Predicting combustion of <185, 56> in 16970.5627484771 ticks
Worker3: Civilian should be rescued by <88100>
Worker3: W_PLAN(Worker3:G0-O1:G1) is the active plan, checking for Barrier events
Worker3: Civ6's HP is 10
Worker3: Motors are 0 and agent is at <185, 57> (should be at <185, 57>)
```

Worker2 now sends a `TASK_COMPLETION` Notification message to the Coordinator and waits for the Coordinator's next command:

```
RT-Core: --------- TICK at time 88601
Worker2: PLAN_EXECUTOR running
Worker2: Doing step 3.0
Worker2: Plan is complete!
Worker2: Worker3:G0-O1:G1-P1:G3 is complete
Worker2: Dropped Worker3:G0-O1:G1-P1:G3, new top-of-stack goal is Worker2:G0-O1:G1
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
```

The Coordinator receives the message and determines that Worker2 has completed all the goals assigned to it by the Coordinator, so the Coordinator sends Worker2 a `RESTORE_AUTONOMY` message.

```
RT-Core: --------- TICK at time 88679
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker2): TASK_COMPLETION
Coordinator: Current goals for worker Worker2:
Coordinator:    0
Coordinator:    1!WORKER=Worker2!TARGET_CIVILIAN=Civ1!TARGET_LOCATION=<172, 31>
Civ6: Worker2 unregistered as a rescuer (total=1)
Coordinator: Worker2 has completed goal Worker3:G0-O1:G1-P1:G3
Coordinator: No goals left on goal Coordinator goal stack, preparing to restore autonomy to Worker2
Coordinator: Restoring autonomy of agent Worker2
RT-Core: Agent Coordinator is sending a message to agent Worker2.
```

When Worker2 receives the message, it performs a state evaluation against the plan for its previous goal, *G1*, and detects a new Barrier event, *B6*. Barrier *B6* indicates that the agent is not in the correct position to perform the action it wishes to (in this case, Worker2 is too far from Civ1's location to perform rescue actions on it):

```
RT-Core: --------- TICK at time 88686
Worker2: CONTROL_LOOP running
Worker2: Got a RESTORE_AUTONOMY message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker2: Restoring autonomy and restarting suspended plans for agent
Worker2: Resuming suspended goal Worker2:G0-O1:G1
Worker2: Plan for goal Worker2:G0-O1:G1 is UNSUSPENDED
Worker2: Testing agent state against that expected for step 2.0
Civ1: Predicting death of Civ1 in 369600 ticks
Civ1: Predicting rescue of Civ1 will take 11400 ticks with 1 rescuers
Worker2: W_PLAN(Worker2:G0-O1:G1) is the active plan, checking for Barrier events
Worker2: Civ1's HP is 77
Worker2: Motors are 0 and agent is at <185, 57> (should be at <173, 31>)
Worker2: Plan for goal Worker2:G0-O1:G1 is SUSPENDED
Worker2: Generated new event Worker2:G0-O1:G1-B6
Worker2: New event Worker2:G0-O1:G1-B6 found!
Worker2: CONTROL_LOOP running
Worker2: Got a BARRIER_EVENT message from CONTROL_LOOP, there are 1 new Barriers
```

Worker2 responds to the Barrier event by regenerating its plan for goal *G1*. The new plan includes travelling back to the location of Civ1 and then performing rescue actions on it until it is rescued:

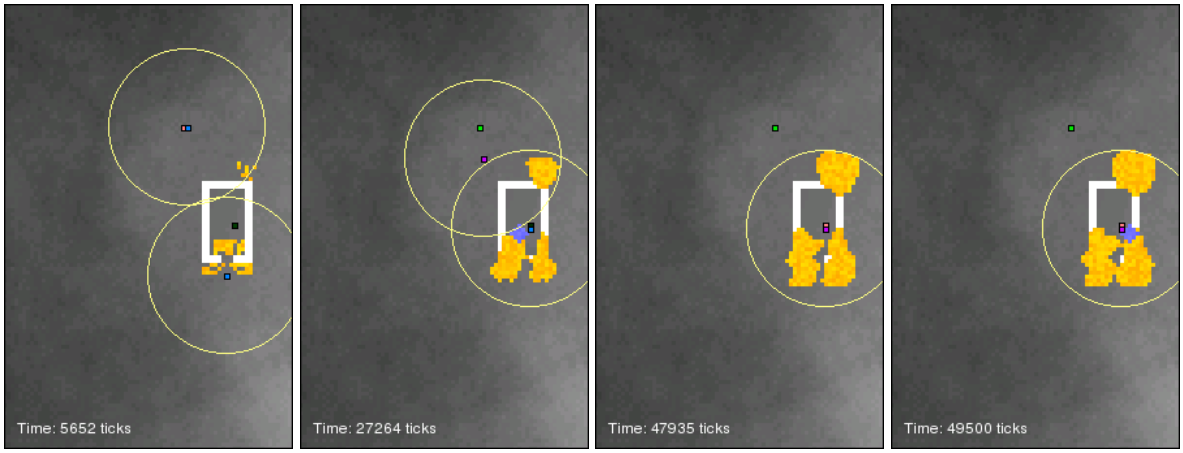

Figure 6.3: Scenario one at 5652, 27264, 47935, and 49500 ticks.

```
RT-Core: --------- TICK at time 88686
Worker2: PLANNER running
Worker2: Order of events is:
Worker2: Type: Barrier, #6, by <446886>
Worker2: Barrier Worker2:G0-O1:G1-B6 has occurred
Civ1: Worker2 registered as a rescuer (total=1)
Worker2: Generating plan 1: Rescue civilian Civ1 at <172, 31>.  Currently at <185, 57>
Worker2: Plan 1 must be complete by <458286>
Civ1: Predicting rescue of Civ1 will take 11400 ticks with 1 rescuers
Worker2: Agent must get to <173, 32>
Worker2: Resuming suspended goal Worker2:G0-O1:G1
Worker2: Plan for goal Worker2:G0-O1:G1 is UNSUSPENDED
Worker2: Accepted the challenge of Worker2:G0-O1:G1-B6, using goal Worker2:G0-O1:G1
Worker2: PLAN_EXECUTOR running
Worker2: Doing step 1.0
Worker2: Initiating movement to <186, 57>
```

Worker2 begins moving back towards Civ1. However, while it was rescuing Civ6 the fire has spread and covered much of the area around the exit to the building Civ6 was located in. Worker2 must therefore spend time extinguishing the fires that block its movement. Each time it encounters fire in its path, it goes through the same process shown above: it is able to determine on its own a solution to the problem, and proceeds each time to extinguish the fire without requesting help from the coordinator. Worker2 encounters fire blocking its path at times 89261, 94474, 100084, 105799, 111513, 117164, 123056, 128677, 134223, 140074, and 145811.

Eventually, Worker2 reaches the location of Civ1 and resumes making rescue actions. The rescue of Civ1 is uneventful, and, at time 165300, Civ1's rescue is complete and the simulation ends. See Figure 6.3 for more screenshots of the scenario's progression.

6.2.3 Summary

Scenario one demonstrates that our implementation can be used to test different multi-agent architectures in a simulated real-time environment that can change unexpectedly. In addition, it shows that our model is capable of handling these unexpected events in a timely manner, and that our coordinator is able to facilitate teamwork amongst otherwise independent agents by adjusting their autonomy when necessary.

In particular, scenario one demonstrates the following ideas.

- Workers can autonomously exploit interesting Opportunities that arise, for example, when Worker3 first observes Civ6 and decides on its own to rescue it.
- Workers can autonomously discard uninteresting Opportunities. For example, as each worker moves it detects many Opportunities to explore the terrain around it; however, because the Worker is already performing a more important task it does not take advantage of these Opportunities.
- Workers can autonomously detect and handle unexpected Barriers that occur. For example, early in the scenario Worker3 is able to decide for itself to extinguish the fire that blocks its path, with no need to contact the Coordinator.
- Workers can autonomously detect and handle unexpected PCFs that occur. For example, at time 25405, Worker3 decides on its own to extinguish a nearby fire that is threatening its rescue target.
- When workers cannot handle a Barrier or PCF, they can get help from the coordinator. When Worker3 determines that its rescue of Civ6 will not be completed before Civ6 dies, it correctly contacts the Coordinator for help.
- The coordinator can maintain a global picture of the entire world using information given to it by individual worker agents, and can use this picture to solve problems that individual Workers are unable to solve. For example, at time 12100, when Worker3 needs to put out a fire, it finds that it cannot do so without also violating its prior commitment to rescue Civ6. Using its global state information, the Coordinator is able to reason about task deadlines to determine that no other agents in the simulation are capable of helping Worker3 rescue

Civ6 without violating any agent constraints, and instead instructs Worker3 to put out the fire despite the missed deadlines.

- The coordinator can facilitate teamwork between workers by adjusting their autonomy. For example, when Worker3 required help to complete the rescue of Civ6, the Coordinator is able to revoke Worker2's autonomy and instruct it to help Worker3 in its task.
- Controlled workers are able to depend on the coordinator to solve problems they may encounter while controlled. When Worker2 begins travelling to Civ6's location, it repeatedly requires the aid of the Coordinator to get past a number of fires that block its path.

In the next section, scenario two will demonstrate some additional ideas that are not shown by scenario one.

6.3 Scenario Two

6.3.1 Background

This scenario is very similar to the first scenario. There are two Worker agents and three injured civilian agents, as shown in Figure 6.4. Unlike in scenario one, in this experiment only Worker2 will detect a civilian (Civ3); Worker1 will instead exploit an Opportunity to explore the terrain nearby. However, because of the wall that separates it from the civilian, Worker2 will be unable to exploit its Opportunity to rescue Civ3. Thus, the coordinator must allocate another worker (Worker1) to perform this task. As the worker executes the plan, it encounters some additional civilians that need to be rescued. This requires the coordinator to reassign an already controlled worker to perform an additional task.

The next section examines the sequence of events that result from this scenario in detail.

6.3.2 Scenario Details

This scenario begins with a similar initialization process to that described in Scenario 1. Once initialization is complete and it has been registered with the Coordinator, Worker1 examines its surroundings. Immediately it receives a number of Opportunities. Each is Opportunity *O2*, and,

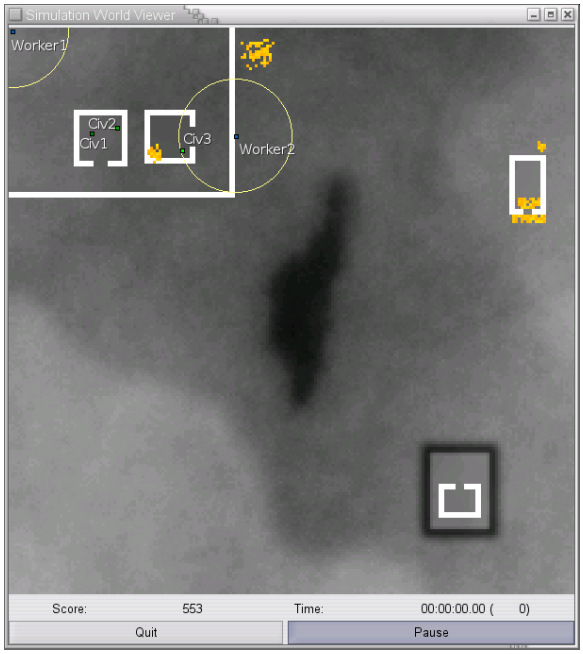

Figure 6.4: The starting state of scenario two, with Worker and Civilian agents labelled.

since all of the events have the same priority, Worker1 selects the first event it detected and generates a plan to explore the terrain around position <0, 20>. Note that the deadline for this plan is given as `<inf>`, indicating that there is no deadline associated with this event (it does not matter how long Worker1 spends trying to accomplish this particular goal).

```
RT-Core: --------- TICK at time 105
Worker1: CONTROL_LOOP running
Worker1: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 37 new Opportunities
Worker1: PLANNER running
Worker1: Generating plan 5: Explore area around <0, 20>.  Currently at <1, 1>
Worker1: Plan 5 must be complete by <inf>
Worker1: Agent must get to <1, 16>
Worker1: Accepted the challenge of Worker1:G0-O2, using goal Worker1:G0-O2:G5
```

Meanwhile, Worker2 also begins processing its sensor data, and also discovers a number of interesting Opportunities. After sorting them by priority, Worker2 tries to exploit the highest priority Opportunity, *O1*, an Opportunity to rescue civilian Civ3. Worker2 decides to attempt a rescue, and tries to generate a plan to achieve *G1*. However, the worker soon discovers that it cannot perform the rescue of Civ3 (since there is a large wall blocking all movement to Civ3's

location). Instead, Worker2 relays the Opportunity event to the Coordinator to handle and examines the next highest-priority Opportunity it encountered, an *O2*. However, attempting to exploit this Opportunity runs into a similar problem; the location of this new event is also behind the large wall and cannot be reached by Worker2. However, Opportunities to explore terrain are not interesting enough to be relayed to the Coordinator, so Worker2 simply discards this Opportunity instead:

```
RT-Core: --------- TICK at time 105
Worker2: CONTROL_LOOP running
Worker2: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 133 new Opportunities
Worker2: PLANNER running
Civ3: Worker2 registered as a rescuer (total=1)
Worker2: Generating plan 1: Rescue civilian Civ3 at <61, 43>.  Currently at <80, 38>
Worker2: Plan 1 must be complete by <1814500>
Worker2: Generated new event Worker2:G0-O1:G1-B5
Worker2: Can't do anything about Worker2:G0-O1 using goal Worker2:G0-O1:G1, due to Worker2:G0-O1:G1-B5
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
Civ3: Worker2 unregistered as a rescuer (total=0)
Worker2: Generating plan 5: Explore area around <60, 38>.  Currently at <80, 38>
Worker2: Plan 5 must be complete by <inf>
Worker2: Generated new event Worker2:G0-O2:G5-B5
Worker2: Can't do anything about Worker2:G0-O2 using goal Worker2:G0-O2:G5, due to Worker2:G0-O2:G5-B5
```

Many of the remaining Opportunities Worker2 discovered turn out to be unexploitable due to their location (these are all *O2* Opportunities, and so are all discarded rather than relayed to the Coordinator); however, Worker2 eventually finds an Opportunity that it can exploit, and so formulates a plan to do so:

```
Worker2: Generating plan 5: Explore area around <80, 18>.  Currently at <80, 38>
Worker2: Plan 5 must be complete by <inf>
Worker2: Agent must get to <81, 22>
Worker2: Accepted the challenge of Worker2:G0-O2, using goal Worker2:G0-O2:G5
```

The Coordinator receives Worker2's `UNHANDLED_OPPORTUNITY` message and attempts to assign a worker to handle it. After some planning steps, the Coordinator is able to determine that, because Worker1's current task (exploring, goal *G5*) is of lower priority than this new task (rescuing, goal *G3*), and because Worker1 can complete the task by the deadline, Worker1 can be assigned the new task.

```
RT-Core: --------- TICK at time 253
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 2 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker2): UNHANDLED_OPPORTUNITY
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: Opportunity, #1, by <1814500>
Coordinator: Trying goal 3...
```

```
Coordinator:    Trying worker Worker2...
Coordinator: Generating plan 3: Rescue civilian Civ3 at <61, 43>.  Currently at <80, 38>
Coordinator: Plan 3 must be complete by <1814500>
Civ3: Worker2 registered as a rescuer (total=1)
Coordinator: Generated new event Worker2:G0-O1:G3-B5
Civ3: Worker2 unregistered as a rescuer (total=0)
Coordinator:       Worker Worker2 can't accept the challenge of Worker2:G0-O1,
                   because of Worker2:G0-O1:G3-B5
Coordinator:    Trying worker Worker1...
Coordinator: Generating plan 3: Rescue civilian Civ3 at <61, 43>.  Currently at <1, 1>
Coordinator: Plan 3 must be complete by <1814500>
Civ3: Worker1 registered as a rescuer (total=1)
Civ3: Predicting rescue of Civ3 will take 12000 ticks with 1 rescuers
Coordinator: Agent must get to <62, 42>
Civ3: Worker1 unregistered as a rescuer (total=0)
Coordinator:       Worker Worker1 is a candidate for Worker2:G0-O1
Coordinator: Best candidate is Worker1
Civ3: Worker1 registered as a rescuer (total=1)
Coordinator: Worker Worker1 will accept the challenge of Worker2:G0-O1, using goal Worker2:G0-O1:G3
```

The Coordinator then updates its model of Worker1's state and transmits a REVOKE_AUTONOMY message to it:

```
RT-Core: --------- TICK at time 1165
Coordinator: Preparing to revoke autonomy of Worker1, new task will be Worker2:G0-O1:G3
Coordinator: Revoking autonomy of agent Worker1
Coordinator: Sending first task to Worker1 and having it do goal Worker2:G0-O1:G3
RT-Core: Agent Coordinator is sending a message to agent Worker1.
Coordinator: Current goals for worker Worker1:
Coordinator:    0
Coordinator:    5!WORKER=Worker1!TARGET_LOCATION=<0, 20>
Coordinator:    3!WORKER=Worker1!TARGET_CIVILIAN=Civ3!TARGET_LOCATION=<61, 43>
```

Worker1 is still busy executing its exploration plan at this time:

```
RT-Core: --------- TICK at time 1349
Worker1: Agent Worker1 has arrived at its next stop, <1, 8>.
Worker1: PLAN_EXECUTOR running
Worker1: Doing step 1.7
Worker1: Initiating movement to <1, 9>
```

However, just after it initiates movement for step 1.7 of its plan, Worker1 receives the REVOKE_AUTONOMY message from the Coordinator. Because Worker1 has been moving around since the Coordinator's plan was formulated, Worker1 must update that plan to reflect its current state before it can begin to execute it. As described in Section 5.6, this updating is necessary due to the method of path planning used, and is considered a "free" action. Once the plan is updated, Worker1 begins moving towards Civ3:

```
RT-Core: --------- TICK at time 1372
Worker1: CONTROL_LOOP running
Worker1: Got a REVOKE_AUTONOMY message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker1: Stopping plans and motors for agent and waiting for new plan
```

```
Worker1: Plan for goal Worker1:G0-O2:G5 is SUSPENDED
Worker1: Got goal Worker2:G0-O1:G3 from coordinator, generating plan...
Worker1:    GENERATING_AGENT = Coordinator
Worker1:    DEADLINE = <1814500>
Worker1:    CAUSE = Worker2:G0-O1
Worker1:    FULLNAME = Worker2:G0-O1:G3
Worker1:    SHORTNAME = G3
Worker1:    PRIORITY = 11
Worker1:    NUMBER = 3
Worker1:    PLAN = W_PLAN(Worker2:G0-O1:G3)
Worker1:    OWNER = Worker1
Worker1:      ARGS{WORKER} = Worker1
Worker1:      ARGS{TARGET_CIVILIAN} = Civ3
Worker1:      ARGS{TARGET_LOCATION} = <61, 43>
Worker1: Generating plan 3: Rescue civilian Civ3 at <61, 43>.  Currently at <1, 7>
Worker1: Plan 3 must be complete by <1814500>
Civ3: Predicting rescue of Civ3 will take 12000 ticks with 1 rescuers
Worker1: Agent must get to <62, 42>
Worker1: Starting new goal Worker2:G0-O1:G3
Worker1: PLAN_EXECUTOR running
Worker1: Doing step 1.0
Worker1: Initiating movement to <1, 8>
```

Both workers are now occupied with travelling to their next destination, so the only interactions occurring for a time are Heartbeat messages:

```
RT-Core: --------- TICK at time 3024
RT-Core: Agent Worker1 is sending a message to agent Coordinator.
RT-Core: --------- TICK at time 3030
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
```

Eventually, Worker2 arrives at the area it wanted to explore:

```
RT-Core: --------- TICK at time 3103
Worker2: Agent Worker2 has arrived at its next stop, <81, 22>.
Worker2: PLAN_EXECUTOR running
Worker2: Doing step 2.0
Worker2: Plan is complete!
Worker2: Worker2:G0-O2:G5 is complete
Worker2: Dropped Worker2:G0-O2:G5, new top-of-stack goal is Worker2:G0
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
Worker2: Resuming suspended goal Worker2:G0
RT-Core: Agent Worker2 is sending a message to agent Coordinator.
Worker2: (clearing HANDLING_FAULTS flag)
```

Seeing only more *O2* events (no *O1* events), Worker2 generates a new goal to explore a different map area and continues on its way. From Figure 6.4 we can see that Worker2 will not encounter any more *O1* events, and indeed Worker2 spends the rest of the simulation exploring the map. As such we will focus on the activities of Worker1 for the remainder of the scenario.

After a time, Worker1's movement towards Civ3 brings it close enough to detect two more injured civilians, Civ1 and Civ2. Because Worker1 is not autonomous, it must relay these Opportunities to the Coordinator to handle. The other 28 Opportunities Worker1 detected are of type *O2* and are not pursued:

```
RT-Core: --------- TICK at time 4305
Worker1: CONTROL_LOOP running
Worker1: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 59 new Opportunities
Worker1: PLANNER running
Worker1: Not autonomous, can't handle Worker2:G0-O1:G3-O1, relaying to coordinator
RT-Core: Agent Worker1 is sending a message to agent Coordinator.
Worker1: Not autonomous, can't handle Worker2:G0-O1:G3-O1, relaying to coordinator
RT-Core: Agent Worker1 is sending a message to agent Coordinator.
Worker2: CONTROL_LOOP running
Worker2: Got an OPPORTUNITY_EVENT message from STATE_EVALUATOR, there are 48 new Opportunities
```

The Coordinator receives the first of the UNHANDLED_OPPORTUNITY Notification messages. It determines that Worker1 should suspend its current goal to rescue Civ3 and instead rescue Civ1. This determination is based on the relative deadlines of the tasks; since Civ1 is more injured than Civ3, its deadline is expires sooner, so the Coordinator assigns it a higher priority than the rescue of Civ3:

```
RT-Core: --------- TICK at time 4562
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 1 lost objects, 3 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker1): UNHANDLED_OPPORTUNITY
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: Opportunity, #1, by <947500>
Coordinator: Trying goal 3...
Coordinator:    Trying worker Worker1...
Coordinator: Generating plan 3: Rescue civilian Civ1 at <29, 37>.  Currently at <30, 18>
Coordinator: Plan 3 must be complete by <947500>
Civ1: Worker1 registered as a rescuer (total=1)
Civ1: Predicting rescue of Civ1 will take 12000 ticks with 1 rescuers
Coordinator: Agent must get to <30, 38>
Civ1: Worker1 unregistered as a rescuer (total=0)
Coordinator:       Worker Worker1 is a candidate for Worker2:G0-O1:G3-O1
Coordinator:    Trying worker Worker2...
Coordinator: Generating plan 3: Rescue civilian Civ1 at <29, 37>.  Currently at <81, 22>
Coordinator: Plan 3 must be complete by <947500>
Civ1: Worker2 registered as a rescuer (total=1)
Coordinator: Generated new event Worker2:G0-O1:G3-O1:G3-B5
Civ1: Worker2 unregistered as a rescuer (total=0)
Coordinator:       Worker Worker2 can't accept the challenge of Worker2:G0-O1:G3-O1,
                   because of Worker2:G0-O1:G3-O1:G3-B5
Coordinator: Best candidate is Worker1
Civ1: Worker1 registered as a rescuer (total=1)
Coordinator: Worker Worker1 will accept the challenge of Worker2:G0-O1:G3-O1, using
             goal Worker2:G0-O1:G3-O1:G3
```

The Coordinator sends the assigned rescue task to Worker1. When the second UNHANDLED_OPPORTUNITY Notification message arrives, the Coordinator attempts to assign the task and finds that no agents are capable of performing it. In our current implementation, the Opportunity is discarded when this situation occurs. A more appropriate response would be to put the second Opportunity on "hold" until an agent can be assigned to exploit it:

```
RT-Core: --------- TICK at time 5204
Coordinator: Preparing to send new task Worker2:G0-O1:G3-O1:G3 to (already controlled) Worker1
Coordinator: Sending next task to Worker1 and having it do goal Worker2:G0-O1:G3-O1:G3
RT-Core: Agent Coordinator is sending a message to agent Worker1.
Coordinator: Current goals for worker Worker1:
Coordinator:    0
Coordinator:    5!WORKER=Worker1!TARGET_LOCATION=<0, 20>
Coordinator:    3!WORKER=Worker1!TARGET_CIVILIAN=Civ3!TARGET_LOCATION=<61, 43>
Coordinator:    3!WORKER=Worker1!TARGET_CIVILIAN=Civ1!TARGET_LOCATION=<29, 37>
Coordinator: PLANNER running
Coordinator: Order of events is:
Coordinator: Type: Opportunity, #1, by <1228300>
Coordinator: Trying goal 3...
Coordinator:    Trying worker Worker1...
Coordinator:       Worker Worker1 can't accept the challenge of Worker2:G0-O1:G3-O1, because
Coordinator:          Equal priorities, but event's deadline is <1228300>, but worker's
                      deadline is <947500>
Coordinator:    Trying worker Worker2...
Coordinator: Generating plan 3: Rescue civilian Civ2 at <38, 35>.  Currently at <81, 22>
Coordinator: Plan 3 must be complete by <1228300>
Civ2: Worker2 registered as a rescuer (total=1)
Coordinator: Generated new event Worker2:G0-O1:G3-O1:G3-B5
Civ2: Worker2 unregistered as a rescuer (total=0)
Coordinator:       Worker Worker2 can't accept the challenge of Worker2:G0-O1:G3-O1,
                   because of Worker2:G0-O1:G3-O1:G3-B5
Coordinator: No workers can perform this task
```

Worker1, already controlled, receives the Coordinator's NEW_TASK message and, after making minor corrections to it, begins executing the new plan:

```
RT-Core: --------- TICK at time 5408
Worker1: CONTROL_LOOP running
Worker1: Got a NEW_TASK message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker1: Plan for goal Worker2:G0-O1:G3 is SUSPENDED
Worker1: Got goal Worker2:G0-O1:G3-O1:G3 from coordinator, generating plan...
Worker1: Generating plan 3: Rescue civilian Civ1 at <29, 37>.  Currently at <40, 21>
Worker1: Plan 3 must be complete by <947500>
Civ1: Predicting rescue of Civ1 will take 12000 ticks with 1 rescuers
Worker1: Agent must get to <30, 38>
Worker1: Starting new goal Worker2:G0-O1:G3-O1:G3
Worker1: PLAN_EXECUTOR running
Worker1: Doing step 1.0
Worker1: Initiating movement to <39, 20>
```

Eventually, Worker1 completes the rescue of Civ1. It sends a TASK_COMPLETION Notification message to the Coordinator and waits for instructions:

```
RT-Core: --------- TICK at time 21252
Worker1: PLAN_EXECUTOR running
Worker1: Doing step 3.0
Worker1: Plan is complete!
Worker1: Worker2:G0-O1:G3-O1:G3 is complete
Worker1: Dropped Worker2:G0-O1:G3-O1:G3, new top-of-stack goal is Worker2:G0-O1:G3
RT-Core: Agent Worker1 is sending a message to agent Coordinator.
Worker1: (clearing HANDLING_FAULTS flag)
```

The Coordinator instructs Worker1 to resume the rescue of Civ3, its old *G3* goal, by sending it a RESUME_TASK message:

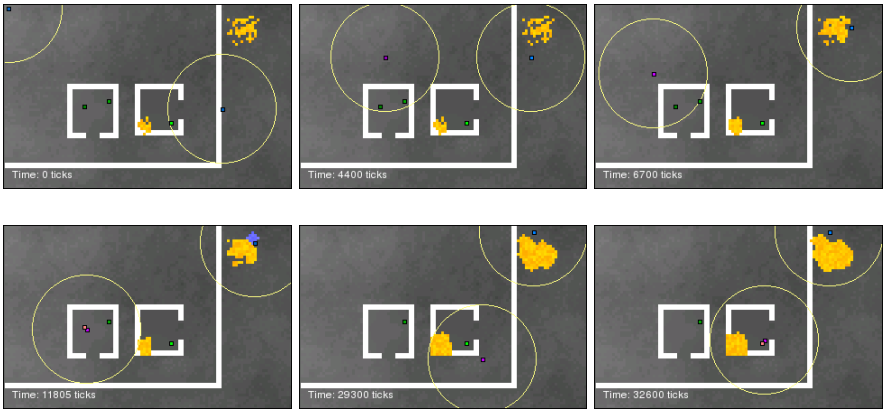

Figure 6.5: Scenario two at 0, 4400, 6700, 11805, 29300, and 32600 ticks.

```
RT-Core: --------- TICK at time 21330
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker1): TASK_COMPLETION
Coordinator: Current goals for worker Worker1:
Coordinator:    0
Coordinator:    5!WORKER=Worker1!TARGET_LOCATION=<0, 20>
Coordinator:    3!WORKER=Worker1!TARGET_CIVILIAN=Civ3!TARGET_LOCATION=<61, 43>
Civ1: Worker1 unregistered as a rescuer (total=0)
Coordinator: Worker1 has completed goal Worker2:G0-O1:G3-O1:G3
Coordinator: Dropped Worker2:G0-O1:G3-O1:G3, new top-of-stack goal is Worker2:G0-O1:G3
Coordinator: Resuming old goal for agent Worker1
RT-Core: Agent Coordinator is sending a message to agent Worker1.
Coordinator: (clearing HANDLING_FAULTS flag)
```

When Worker1 gets the `RESUME_TASK` message, it examines its plan for the old goal, *G3*, to rescue Civ3. However, Worker1 discovers that the plan for this goal is no longer correct: it assumes Worker1 is at a different location than it currently is. Subsequently, Worker1 generates a Barrier event, *B6*, a special Barrier that causes the Worker to simply regenerate the plan's path on its own. This Barrier stems from the method of path planning we use. Once the path has been updated, Worker1 resumes executing its plan:

```
RT-Core: --------- TICK at time 21597
Worker1: CONTROL_LOOP running
Worker1: Got a RESUME_TASK message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker1: Preparing to resume goal Worker2:G0-O1:G3
Worker1: Resuming suspended goal Worker2:G0-O1:G3
Worker1: Plan for goal Worker2:G0-O1:G3 is UNSUSPENDED
Worker1: Testing agent state against that expected for step 1.7
Worker1: Testing agent state against that expected for step 1.44
Civ3: Predicting death of Civ3 in 1800000 ticks
Worker1: Civ3's HP is 250
```

```
Worker1: Motors are 0 and agent is at <30, 38> (should be at <40, 20>)
Worker1: Plan for goal Worker2:G0-O1:G3 is SUSPENDED
Worker1: Expected to be at <62, 42>, but am at <30, 38> instead (extinguish)
Worker1: Generated new event Worker2:G0-O1:G3-B6
Worker1: New event Worker2:G0-O1:G3-B6 found!
Worker1: CONTROL_LOOP running
Worker1: Got a BARRIER_EVENT message from CONTROL_LOOP, there are 1 new Barriers
Worker1: PLANNER running
Worker1: Order of events is:
Worker1: Type: Barrier, #6, by <1795851>
Worker1: Barrier Worker2:G0-O1:G3-B6 has occurred
Worker1: Generating plan 3: Rescue civilian Civ3 at <61, 43>.  Currently at <30, 38>
Worker1: Plan 3 must be complete by <1821597>
Civ3: Predicting rescue of Civ3 will take 12000 ticks with 1 rescuers
Worker1: Agent must get to <62, 42>
Worker1: Resuming suspended goal Worker2:G0-O1:G3
Worker1: Plan for goal Worker2:G0-O1:G3 is UNSUSPENDED
Worker1: Accepted the challenge of Worker2:G0-O1:G3-B6, using goal Worker2:G0-O1:G3
Worker1: PLAN_EXECUTOR running
Worker1: Doing step 1.0
Worker1: Initiating movement to <30, 39>
```

Worker1 eventually reaches Civ3 and rescues it successfully. Since the Opportunity to rescue Civ2 was discarded by the Coordinator, after completing the rescue of Civ3, the Coordinator releases Worker1 and restores its autonomy:

```
RT-Core: --------- TICK at time 43876
Coordinator: StateEvaluator: Merging updated state information into global state... Coordinator: done
Coordinator: There were 0 lost objects, 0 found objects, and 0 cells whose On Fire state has changed
Coordinator: Got a NOTIFICATION message from MAS_MESSAGE_RECV (Agent Worker1): TASK_COMPLETION
Coordinator: Current goals for worker Worker1:
Coordinator:    0
Coordinator:    5!WORKER=Worker1!TARGET_LOCATION=<0, 20>
Civ3: Worker1 unregistered as a rescuer (total=0)
Coordinator: Worker1 has completed goal Worker2:G0-O1:G3
Coordinator: No goals left on goal Coordinator goal stack, preparing to restore autonomy to Worker1
Coordinator: Restoring autonomy of agent Worker1
RT-Core: Agent Coordinator is sending a message to agent Worker1.
Coordinator: (clearing HANDLING_FAULTS flag)
```

Worker1 then resumes working on its original, autonomously generated, goal of exploring part of the map. Note that again Worker1 generates a *B6* event, since it is no longer at the location that the resumed plan expects:

```
RT-Core: --------- TICK at time 43883
Worker1: CONTROL_LOOP running
Worker1: Got a RESTORE_AUTONOMY message from MAS_MESSAGE_RECV (Agent Coordinator)
Worker1: Restoring autonomy and restarting suspended plans for agent
Worker1: Resuming suspended goal Worker1:G0-O2:G5
Worker1: Plan for goal Worker1:G0-O2:G5 is UNSUSPENDED
Worker1: Testing agent state against that expected for step 1.7
Worker1: Plan for goal Worker1:G0-O2:G5 is SUSPENDED
Worker1: Expected to be at <1, 8>, but am at <62, 42> instead (movement)
Worker1: Generated new event Worker1:G0-O2:G5-B6
Worker1: New event Worker1:G0-O2:G5-B6 found!
Worker1: CONTROL_LOOP running
```

```
Worker1: Got a BARRIER_EVENT message from CONTROL_LOOP, there are 1 new Barriers
Worker1: PLANNER running
Worker1: Order of events is:
Worker1: Type: Barrier, #6, by <inf>
Worker1: Barrier Worker1:G0-O2:G5-B6 has occurred
Worker1: Generating plan 5: Explore area around <0, 20>.  Currently at <62, 42>
Worker1: Plan 5 must be complete by <inf>
Worker1: Agent must get to <4, 19>
Worker1: Resuming suspended goal Worker1:G0-O2:G5
Worker1: Plan for goal Worker1:G0-O2:G5 is UNSUSPENDED
Worker1: Accepted the challenge of Worker1:G0-O2:G5-B6, using goal Worker1:G0-O2:G5
Worker1: PLAN_EXECUTOR running
Worker1: Doing step 1.0
Worker1: Initiating movement to <63, 41>
```

Worker1 returns to exploring the map, and the simulation ends. See Figure 6.5 for screenshots of the scenario's progression.

6.3.3 Summary

Scenario two shows some additional interesting cases that did not occur in scenario one. For example, the Coordinator is used to assign a goal to exploit an Opportunity to a different worker (Worker2) than the worker that discovered the Opportunity. As well, when Worker2 detects a additional civilian that can be rescued, the Coordinator reassigns it to the new task of rescuing the civilian, demonstrating how the coordinator can give more tasks to already-controlled workers.

In particular, scenario two demonstrates the following ideas.

- The coordinator can assign an otherwise unexploitable Opportunity to a different worker that is better able to take advantage of it.
- The coordinator can reassign an already-controlled worker to a new task should the task be deemed more important than the worker's current one.
- When a controlled worker completes an assigned task but has other assigned tasks to complete before it regains its autonomy, the coordinator can instruct it to resume those tasks.

In the next chapter, we will discuss the results of these experiments in more detail.

Chapter 7

Discussion

7.1 Advantages Of Our Approach

7.1.1 Inherent Value Of The Model

Our research aims to develop methods that allow autonomous groups of agents to reason about problem solving when faced with real-time and resource constraints. We propose a specific architecture for the coordination of agents within a society and specify conditions for communication amongst agents, with a focus on describing how to manage unexpected obstacles in the environment.

Unexpected Events

The proposed classification of unexpected events provided in our architecture into Barriers, Opportunities and Potential Causes of Failure is especially useful for providing customized solutions to dealing with these events. In addition, this classification is a useful basis for designing experiments to test the environments that a cooperative real-time multi-agent system may face.

The experiments described in Chapter 6 illustrate almost all of the possible events that can occur in the simulations as designed. Specifically, they show Opportunities being handled by workers and being relayed to the coordinator to be assigned to other workers, Barriers occurring and being handled, PCFs being raised and then directly handled by the worker, and PCFs being handled by the coordinator by assigning them to another worker.

Our architecture is flexible enough to handle a wide variety of unexpected events, as demonstrated by the experimentation in the previous chapter, which shows many different cases of these events occurring in different circumstances. The implementation successfully shows these events being handled within the architecture.

Architecture

Another critical aspect of our proposed architecture is its decision to keep some agents controlled and others fully autonomous, specifying when the autonomy of agents may be adjusted.

The changing nature of the autonomy of the agents and the benefits of allowing for this adjustment of autonomy are demonstrated in the experimentation. For example, in the first scenario discussed in Chapter 6, our coordinator determines that, to successfully complete an important rescue task on time, it is necessary to revoke the autonomy of a worker agent (Worker2) and instruct it to come to the aid of the original worker agent that began the rescue. This case exemplifies the value of having some agents autonomous and others controlled: in the scenario, the autonomous agents are able to react to changes in their environment in a timely manner and without wasting time communicating with other agents, while those agents that are controlled allow the system to avoid time-consuming negotiation and communication. Our architecture allows the coordinator to ensure that any collaboration that becomes necessary can be managed in a timely and efficient manner by removing most of the communication bottlenecks.

The coordinator also provides a global perspective for the system, allowing the system to avoid problems where agents, due to lack of information, perform actions that are not beneficial to the society. For example, during the simulation of scenario one, Worker3 finds that a fire is approaching the civilian it is rescuing. However, performing the action of extinguishing the fire will take up enough time that Worker3 will no longer be able to finish its rescue before the civilian dies. Because the coordinator maintains a global perspective and does not need to contact any other workers to query their states, it is able to very quickly determine that no other agents can come to the aid of Worker3 such that its deadlines are not violated, and so it instructs Worker3 to extinguish the fire regardless of the problem (since no other worker could do better than Worker3). Thus, the coordinator's ability to immediately access information not available to individual workers allows it to make better decisions faster than individual workers could.

Communication

The final critical element of the architecture is the support for communication between the agents, with the proposed messaging strategies.

The simplest message type, Heartbeats, were included as part of the design to ensure that the information that the coordinator bases its decisions on is kept reasonably up-to-date. For example, in both experiment scenarios there were long periods when a worker was performing a task and did not need to communicate with the coordinator. During these times, the only messages the workers sent to the coordinator were Heartbeat messages, which contained updated information about the worker's current activities. Without such information, the coordinator's model of the worker's activities could become increasingly divergent from the worker's true actions (e.g. the third panel in Figure 6.2).

In addition to keeping the coordinator abreast of each worker's current actions, Heartbeat messages ensure that the coordinator can adequately model and track each worker's set of deadlines. Because the coordinator itself cannot observe the environment in which its workers are operating, its ability to model the entire world is dependent on the workers sending the coordinator the required information. For example, suppose that the coordinator receives a message from a worker indicating that the worker is going to begin rescuing a civilian, and that its initial prediction for the deadline of this task is at time 30000. As the worker makes progress on its goal, it will be able to refine this prediction to provide a more accurate estimate of when the deadline is. However, without Heartbeat messages, the coordinator will not be able to benefit from these more accurate predictions, and may as a result make incorrect decisions when important tradeoffs must be made.

Notification messages are the standard type of message used in our architecture; most of the messages a worker sends to the coordinator are of this type (for example, `GOAL_CHANGED` and `UNHANDLED_OPPORTUNITY` are both Notification messages). As well as containing any message-specific information (such as a goal or plan object, or an event), Notification messages also contain any local state information that has changed since the worker last sent a message. Thus, Notification messages also serve to keep the coordinator updated about the state of the worker that sent the message, so that it can better analyse the situation. All of the different Notification messages are shown in the experimentation described in Chapter 6.

Panic messages were included as a separate category in order to allow them to be given

higher priority or treat them in some special manner, because they are information about a fault (a Barrier or a PCF) that has occurred, which may be important to deal with. They are currently not handled differently in the implementation, because the implementation is sufficiently simple that it was not necessary to separate this type of event from the lower priority Heartbeat and Notification message types. However, in a more complex situation, where the coordinator is far less idle, it would be more important for certain messages, and in particular Panic-level messages, to be recognized and addressed with priority.

The messages that the coordinator sends to workers to revoke or restore autonomy are not truly part of the above classification, since they do not contain state updates, and only ever contain commands from the coordinator for the worker to perform.

Adjustable Autonomy

The model presented in this thesis is also valuable for researchers designing adjustable autonomy systems. As discussed in Chapter 2, one challenge of multi-agent systems is to determine when the agents in the system should adjust their autonomy. Our model proposes that agents should be monitoring for unexpected Opportunities, Barriers, and PCFs, and that the latter two types of events are in fact what may cause an agent to lose its autonomy. More specifically, we propose that agents contact a coordinator, who will then determine which agents should have their autonomy revoked, in order to address the unexpected events. We are therefore proposing a specific method for agent-based adjustable autonomy.

7.1.2 Value Of The Implementation

The implementation described in Chapter 5 encodes a methodology for addressing soft real-time constraints within a multi-agent system. As discussed in Section 5.3, the real-time core incorporated into the implementation serves to simulate the actual real-time operating system and hardware, and is therefore useful for testing real-time multi-agent systems.

Our implementation provides a valuable testbed for our model, which we have used in Chapter 6 to demonstrate its abilities. However, our implementation also provides an architecture-neutral method that can be used to compare different multi-agent system designs. Because the implementation is split into world simulators and agent simulators, it is a straightforward pro-

cess to replace the currently implemented agents, based on our model, with new agents that use an entirely different form of coordination or have an entirely different structure. New agents built to use our simulator can still take advantage of the supporting functions it provides, such as real-time schedulers and simulated radio links.

It is also an easy matter to create new scenarios with which to test agents (ours or otherwise). Because scenarios are specified using only plain text files and standard PNG-format images, no special software is required to create new ones or adjust existing ones. For example, a scenario could be modified that has more or fewer agents, or more or fewer civilians, than the previous scenario. Individual civilians could be made harder or easier to rescue by adjusting their starting hit points or the civilian's rescue difficulty. Even the basic rules of the simulation can be changed: a scenario could be created in which inter-agent communication was free, or was very expensive, or where fires are easier or harder to extinguish. Simulator scenarios are very flexible and are easy to create and adjust.

It is also easy to create new agents for use with our simulator. To create a new agent, simply create a subclass of the `Agent` class described in Appendix A. The `Agent` class provides hardware simulators for mobile agents, such as motors and sensors. Also provided by this class are the `Goal`, `Deadline`, and `Event` objects that encapsulate goals, deadlines, and the three types of unexpected events described in Section 3.3. The existing `Goal` and `Event` objects detail those goals and events described in Section 5.7, and may need to be augmented to include any new goals, Barriers, Opportunities, or PCFs the new agents can encounter. Of course, any new agents can also take advantage of the many real-time and world simulation classes that are part of the basic simulator.

The final step towards creating a new agent involves creating a new scenario for the agent, and ensuring that the scenario's configuration file is set to refer to the new agent class. Once configured, the `World` class will automatically instantiate the new agent class based on the entries in the scenario's object specification file.

Appendix A explains in detail the files and classes found in the simulator source code. It also provides complete documentation on the scenario specification files, so that new scenarios may be created easily.

7.1.3 Comparison With Alternative Architectures

Our proposed architecture creates a coordinator agent that has total authority over all the other agents in the system. The coordinator assigns missions to its agents and allows them to work autonomously until a new task (such as a Barrier or PCF) arises. Workers in the system communicate only with the coordinator. We contrast this design with the more common approach of having agents negotiate with each other whenever tasks need to be assigned or problems need to be solved [15]. We believe this more democratic approach does not operate well with the requirements of real-time response and restricted communications bandwidth. In such a system, the negotiations to assign a task may not even be complete before the task's deadline elapses.

However, having a single coordinator controlling a number of agents remotely can also require more communications bandwidth than is available. In addition, the delay in communicating can itself become a problem over long distances or noisy channels. To resolve this problem we allow our agents to perform as much work as possible without relying on the central coordinator. When an agent is autonomous, the only messages it sends are notifications of changes in its state. Such messages can become delayed or lost without compromising the system. Only when the agent cannot proceed alone will it contact the coordinator for assistance.

Our architecture is a compromise between these two extremes. Systems that use our architecture should gain the flexibility and robustness that are the major benefits of a peer-based system, and the faster response time and global perspective that a centralized system provides. By avoiding the overhead of negotiation, while allowing individual agents as much autonomy as possible, our design can assign an appropriate task to an agent more quickly.

The simulator described in Chapter 5 was designed to support implementations of these two alternative architectures for real-time multi-agent systems. Our original plans called for experiments in which each of the three architectures (our model, a peer-based model, and a centrally-controlled model) would be run through a set of scenarios. Our simulator computes a score, displayed in the user interface, that is the sum of the current hit points of all civilians in the scenario. We planned to determine when our model performs better than, or worse than, the other two methods, by comparing the resulting scores of each system in a given scenario. We expected that our model would perform at least as well as the centrally planned model, since our model in the worst case (all agents controlled) reduces to that model. We also expected that our model would outperform the peer-based models when communication costs were high, and

when there are many conflicting tasks that must be completed. As discussed in Section 7.3, we wish to pursue this additional experimentation in the future.

7.1.4 Inherent Value For Various Applications

We have briefly discussed the relevance of our work for the application area of RoboCup Rescue. Although we have currently only explored the RoboCup Rescue simulation league, the aim is for our proposed model to be useful in controlling actual robots performing search and rescue tasks. We have also discussed, in a very general way, the application area of directing robots in remote locations.

Some aspects of our current solution are already somewhat promising for the control of robot agents. In particular, in a large multi-agent system with no central coordinator, putting large amounts of computing power on every agent is costly, and there is no need for every agent to maintain a model of global state. Worker agents need only be aware of their local environment. We suspect that this will be especially important for robotic agents because such agents are expensive to produce: the cheaper the individual agents are, the more of them we can use in the system.

As mentioned, our research also aims to provide a framework for adjusting the autonomy of agents within multi-agent systems. Our hybrid approach, allowing fully autonomous agents to co-exist with those controlled by a coordinator, is presented as a valuable strategy when systems are faced with real-time constraints and unexpected events. In contrast with research such as [13] on how humans can specify adjustments to agent autonomy, we clarify circumstances under which agents may elect to revoke their autonomy, requesting assistance in the completion of their tasks, from a central coordinator. There may therefore be a variety of applications requiring adjustable autonomy that would benefit from our research.

7.2 Related Work

7.2.1 Task Allocation And Reallocation

Ortiz & Rauenbush

Our work is similar to that of Ortiz & Rauenbush [15], in that both deal with the allocation and re-allocation of tasks in an uncertain environment. Both their work and ours also have the goal of minimizing communication by communicating only when necessary, and only to those necessary. However, while Ortiz & Rauenbush's new tasks are exclusively generated outside of their framework, our tasks can come from inside the system (from the coordinator or a worker), as well as from an outside agent. Ortiz & Rauenbush select a temporary mediator to supervise task allocation. This mediator then holds an auction on the task. In contrast, our system has a permanent agent in the role of supervisor: the coordinator, who dictates task assignments to its workers. Finally, while Ortiz & Rauenbush support planning with a temporal dimension, they do not handle any other aspects of a real-time system. Our system can handle real-time failures such as slipping schedules and endangered deadlines.

7.2.2 Robotics

Balch And Arkin

Balch and Arkin [2] examine three different modes of communication in a system of autonomous agents. Agents observe each other for state markers (such as lights or sounds) that indicate the agent is doing something interesting. Their system produces cooperation between agents when, for example, one agent observes that another is performing a task that the agent wants to perform as well. Thus, their system does not use a coordinating agent; it relies instead on worker agents observing a problem and proactively working to solve it. In contrast, our system uses a central coordinator to help manage urgent problems that individual agents cannot solve, making use of its global perspective to organize the team of agents in an environment with many different, possibly conflicting, tasks.

Morimoto *et al.*

In addition, for RoboCup Rescue applications, our approach imposes more coordination among the individual agents, in contrast to that of Morimoto, Kono, & Takeuchi [11]. Their initial entry into the RoboCup Rescue contest, YabAI, relied on an algorithm similar to the emergent behaviour strategy employed by Balch and Arkin above. While YabAI was the winner in the first contest, its performance was still far below the level that would be required in a real disaster rescue scenario. An explicit coordination mechanism, such as in our model, can provide this needed enhancement.

Wegner and Anderson

Wegner and Anderson [22] are interested in using robotic agents to help humans find victims after a disaster. Robots are particularly suited to this domain due to their ability to enter places that too small or too dangerous for humans to go. Currently, such search-and-rescue robots are tele-operated by a human at a safe location. To be able to increase the number of robotic searchers in such an environment, rescue coordinators must be able to allocate enough trained humans to control the additional robots, since each user can only directly control a single robot at a time.

Wegner and Anderson propose a hybrid system in which those robots that are not being controlled remotely can operate autonomously instead. The autonomous robots can thus explore the area and determine individually what actions to take. However, in the event that a robot requires assistance or discovers an injured person that needs rescuing, the robot can signal the human operator for attention. Thus, their goal is to allow the human to focus on the important problems, the ones that the robots cannot solve on their own, and allow the robots to handle the simpler issues that they encounter on their own.

To achieve this balance, they create two new agents for each robot, in addition to the robot control agents already present. The first agent is a “mediation agent,” which is designed to blend the low-level commands of the tele-operator and the robot using a weighted function. For example, if the operator directs a robot to move in a particular direction, but the robot knows that doing so would cause it to fall down a hill, the mediation agent will adjust the autonomy of the robot to prevent the operator’s command from damaging the robot. Of course, should

the operator actually desire the robot to risk falling down the hill, the operator has the option to switch the agent into a fully tele-operated mode, where the operators commands are always obeyed. Agents can also be placed into a fully autonomous mode, in which the reverse is true (i.e., the agents own decisions are always followed).

In addition to the mediation agent, Wegner and Anderson describe an "intervention recognition agent," used to detect situations in which operator intervention is necessary. This agent is the component that determines what the weights used to balance autonomous versus tele-operated behaviour should be. The recognition agent analyses the robot's perceptions (that is, the information it receives from its various sensors), and compares this information to a knowledge base to determine the probability that the robot can accomplish its tasks successfully. The knowledge base is composed of a number of scenarios, such as the robot being stuck and unable to move.

Although this research is focused on tele-operation and the role of human overseers, this design is similar to our own, as both architectures allow many agents to operate autonomously, handing any minor unexpected events as they occur, and falling back on a more intelligent central operator in case of more difficult situations.

7.2.3 Adjustable Autonomy

Schillo

Our work is also related to other research in constructing adjustable autonomy systems. Schillo [18] describes how the structure of a multi-agent system can affect the autonomy of its agents. He discusses various kinds of organizations that may emerge in a multi-agent environment, focusing on mechanisms for agents to self-organize. Our approach also uses structural changes to adjust the autonomy of the workers, though in a more restricted manner. We are concerned with the re-organization of agents directed by a central coordinator.

Schreckenghost *et al.*

Schreckenghost *et al.* [19] proposes an architecture that supports multiple human interaction with multiple control agents. For example, in a space application, this architecture assigns each crew member a Crew Proxy agent. The Crew Proxy agents are then informed by Control Assistants about various anomalies and potentially adverse conditions. This contrasts with our

approach whereby worker agents are able to detect problems in the environment, but coordinators help to determine how to resolve potential problems. As in our research, Schreckenghost *et al.* emphasize the importance of communication to detect when tasks have been completed, and of centralization to prevent conflicting commands.

Scerri *et al.*

Our work is also related to other efforts on providing for adjustable autonomy of agents. For example, Scerri et al. [16], propose a system that uses "role allocation" to allocate a new task in the system. An agent (or its proxy) can accept or reject a proposed new task. If the task is rejected, the proxy proposes that another agent complete the task. Similar to our model, an agent may take on new tasks as opportunities arise, but in contrast, we employ a single coordinating agent to allocate new tasks to other agents, when the agent who detects the unexpected event cannot handle it, itself.

7.3 Future Work

We envision a number of possible topics for future research based on our model and implementation.

Scalability

One topic we would like to explore in more depth is the scalability of our architecture. In particular, we would like to investigate scenarios that are too complex for a single central coordinator, as is assumed in our current model. Such scenarios may require multiple coordinators, each with their own set of workers. In this situation, the problem of communication between different coordinators becomes an issue. A key challenge is to determine how best to divide the workload of the system amongst different coordinators. Solving this problem would allow us to implement a full RoboCup Rescue simulation league agent.

One possible solution could be to create a hierarchy of coordinators, in which each coordinator has as its workers other, lower-level, coordinators. In this manner we could extend our architecture to allow for many different populations of agents that must work together. How-

ever, in the absence of such a hierarchy, it would be important to determine how the coordinators should communicate amongst each other to ensure tasks are performed correctly. For example, coordinators might only send messages to other coordinators in response to Panic events, when the first coordinator needs to access a resource possessed only by the second. We expect our architecture is flexible enough to support both of these methods.

Prioritizing Events

Another area that we have not fully specified in this thesis is how various events are prioritized. It is possible that many unexpected events (Barriers, Opportunities, or PCFs) could occur within a short period of time. These events may impose different, perhaps conflicting, constraints on the worker or coordinator that must handle them. Thus, to implement our model one needs to provide a strategy for assigning priorities to each possible event. Our current implementation assigns statically assigned priorities to each event based on the priority of the goal that the event occurred for. However, in a more complex environment where there are many more possible events, a more rigorous method will probably need to be employed.

Planning and Assigning Tasks

In addition, the method by which the coordinator assigns tasks to workers should be studied more extensively. In our current implementation, the coordinator simply selects the worker that can complete the task successfully in the shortest time. It is likely that this simple strategy will fail in more complex scenarios. It is also likely that this component could be merged into the planning mechanism for the coordinator in order to allow the coordinator to reason about what agents it should select for a task.

Similarly, the planner we have used in our implementation is very simple and quite limited. For more complex scenarios, and for real-world experiments and usage, a more complex planner, such as those described in Chapter 2, should be included. We would like to integrate such a planner into our system so that it will more accurately reflect the timings needed to perform the various tasks that are now only approximated by our simulation.

However, because planning is a processor-intensive activity, it may become impractical to run the entire simulation on a single computer, even for small numbers of agents. Thus, we may also have to re-architect the simulation code to allow it to be distributed across multiple processors.

Extensions to RoboCup Rescue Modelling

Our simulation of the rescue scenario is extremely limited, since our aim is not to design a rescue agent. To extend our work to allow it to operate a complete system of rescue robots, or even the agents in the RoboCup Rescue simulation, we will need to more clearly enumerate the possible unexpected events, and include many events which we did not need to handle in our current implementation. For example, in a RoboCup Rescue situation, there will likely be many more possible PCF events, to handle the many possible deadline failures that can occur. Likewise, the set of goals and their plans will need to be enhanced and expanded.

Improved Path Planning

Some less ambitious future work we would like to do with the implementation is to replace the current static path planner (found in `World::PathPlanner`) with a more flexible navigation system. The current method of path planning required that the worker "adjust" any paths sent to it by the coordinator, because the coordinator's information on the worker's position can be incorrect, which would otherwise invalidate the path. A better solution is to put a more flexible navigation method, such as those discussed in [9], onto each worker.

Programming Language

A final adjustment to the simulation we would like to make would be to rewrite the simulator in a faster language than the current Perl 5. Perl 5 was chosen because it initially provided the ease-of-implementation requirements we needed to write the simulator in a reasonable time, with less debugging overhead. Rewriting the simulator in Java or C# should provide a much-needed speed increase in the simulation running time.

Additional Experiments

We would also like to do some more experimental work with our model and simulation. Our original goal was to directly compare our model with the peer-based and centrally-planned models described in section 7.1.3. Because implementing these alternatives was originally one of our goals, the current version of the simulator was designed to be independent of the structure of the

multi-agent system that would be running on top of it. Thus, it should be fairly straightforward to implement each of the architectures on top of the existing simulation code.

To properly compare our architecture to these alternatives, we would also have to devise many more scenarios, to provide a broader and more complete representation of the kinds of situations rescue agents might encounter in a real rescue situation. Particularly, scenarios where more than a few agents are operating should be extensively tested. For the comparison to be truly useful, we would also have to fill in the missing parts of our model, such as the planner, as has been mentioned earlier in this section. As well, the inclusion of more agents and more advanced planners could require rewriting of the simulator as also mentioned previously.

Fault Tolerance

A system built using our architecture must deal with some problems not discussed in this thesis, such as hardware failures and communications reliability. Because our focus was on inter-agent interaction and high-level fault-handling only, we do not specify the low-level aspects. However, in the future we would like to investigate how we can adjust the architecture to increase its fault tolerance. For example, we might introduce multiple coordinators for redundancy purposes, so that if one coordinator fails another could take over. Regardless of what modifications we make to increase our model's fault-tolerance, we would need to perform experiments to validate them.

Real Robots

Finally, we would like to eventually test our architecture on real robots, such as is described in [22]. Testing on real robots will allow us to test our system in a non-simulated environment; workers' interactions with the world will no longer be simulated, a limitation of our current implementation.

Chapter 8

Conclusion

8.1 Conclusion

This research explores an architecture that supports varying levels of autonomy within the same multi-agent system, outlining the conditions under which agents may communicate to adjust their level of autonomy. We discuss in more detail cases of Opportunities, Barriers, and Potential Causes of Failure, and propose a mechanism for reallocating tasks within the community to address these conditions, resulting in changes to the autonomy of the agents. Finally, we elaborate on how real-time constraints may impact the processing, communication, and autonomy of the agents. As such, we provide a framework for designing multi-agent systems that support adjustable autonomy in order to address new tasks that arise due to unexpected problems in a soft real-time environment. In addition, we provide an implementation that both demonstrates the flexibility of our model and serves as a useful testbed for experimenting with architectures for real-time multi-agent systems.

Appendix A

Simulator Documentation

A.1 Requirements

The simulator described in this chapter is implemented in Perl 5.8. It requires that the following modules be installed:

- Glib
- Gtk2
- Image::Imlib2
- Math::Random
- List::Compare
- Data::Dumper
- Heap::Fibonacci

All modules are available from CPAN (`www.cpan.org`).

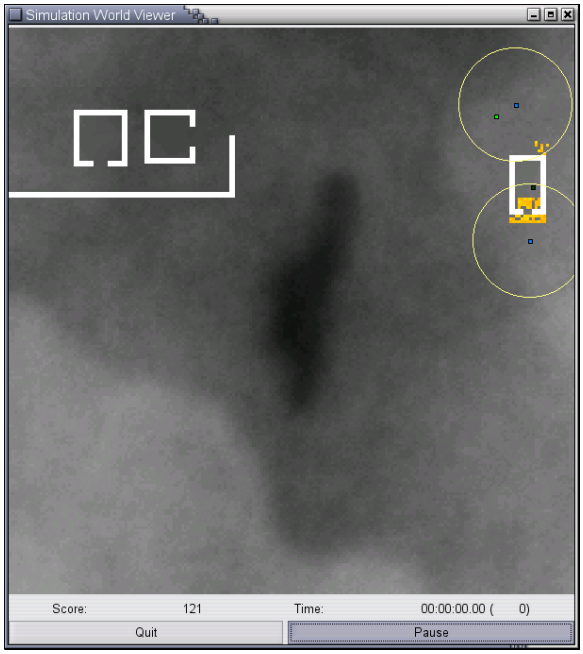

Figure A.1: Simulator User Interface

A.2 Usage

A.2.1 Running The Simulator

To execute the simulator, type

```
./sim config.txt
```

where `config.txt` is the name of the scenario configuration file to use. For example, to run the simulator on the "scenario1" scenario, type

```
./sim maps/scenario1/config.txt
```

A.2.2 Simulator Interface

Figure A.1 shows the user interface presented while the simulation is running. Most of the window shows an enlarged view of the terrain map, with objects represented by coloured boxes.

Blue squares are drawn for autonomous agents, purple squares for remotely-controlled agents, and green squares for civilians. The brightness of the colour indicates how many hit points the agent has remaining: the darker a coloured square, the fewer hit points that agent has. Fires are drawn as areas of orange and yellow squares. If an agent is performing an Extinguish action, the area being extinguished is drawn in light blue.

For agents that can "see," the visibility range is indicated by a yellow circle. All map cells in this range (even those behind walls) are visible to the agent, while all cells outside are not.

The bottom of the window shows the current "score" and the current simulation time. The score is the sum of the hit points of all Civilian agents in the simulation. The simulation time is displayed in hours:minutes:seconds format, followed by the number of ticks in brackets.

The "Quit" button stops the simulation immediately and exits the simulator. The "Pause" button is a toggle that allows the simulation clock to be stopped and started as desired.

A.3 Creating A Scenario

A.3.1 Overview

To create a scenario for the simulation, four files are required: a configuration file, a terrain map, a reachability map, and an object specification file. Table A.1 summarizes some of the terms used when discussing scenarios.

Term	Description
Ticks	The basic unit of time. The duration of time a tick represents is set by the configuration option `TICKS_PER_SECOND`.
Cells	A map cell represents a 0.5 m $\times$ 0.5 m area. Each pixel in the terrain bitmap (see Section A.3.3 corresponds to a cell in the world map.
Combust Points	These integer values track how close a cell is to catching on fire. Section 5.4.2 describes how cells accumulate combust points.
Extinguish Points	These integer values track how much effort has been applied towards extinguishing a fire. Section 5.4.2 describes how cells accumulate extinguish points.
Hit Points	These integer values track how much injury a civilian has sustained. See Section 5.5.3 for details on how a civilian's hit points change over time.
Rescue Points	These integer values track how much effort has been applied towards rescuing a civilian. Section 5.5.3 describes how civilians accumulate rescue points.

Table A.1: Common scenario terms.

A.3.2 The Configuration File

The configuration file sets the parameters of the simulation, and specifies the names of the other three files that, together this the configuration file, specify the scenario. Table A.2 describes the available configuration parameters and gives an example value for each.

The configuration file has the following format:

```
CONFIGURATION_PARAMETER_1:  value1
CONFIGURATION_PARAMETER_2:  value2
...
```

Blank lines and comments (text preceded by a # character) are ignored.

Option Name	Example Value	Description
Physical constants		
TICKS_PER_SECOND	300	Number of ticks per second of simulated time
MAX_SLOPE	2	The maximum slope (metres) an agent can traverse
Map constants		
WORLD_WIDTH	200	Width of world in cells
WORLD_HEIGHT	200	Height of world in cells
BUF_SCALE	3	Scale the world by this many pixels when drawing it (does not affect the simulation)
HEIGHT_SCALE	10	Number of height map gradations per vertical metre
Fire properties		
COMBUST_TARGET	10000	Number of combust points a cell must accumulate to catch fire
BURNINATE_POINTS	2500	Number of combust points per burning neighbour a cell accumulates per unit of time
BURNINATE_INTERVAL	3000	Time between burninations (10 seconds)
BURNOUT_POINTS	1	Number of extinguish points the fire automatically gains each second
Costs (time) to do things (in ticks)		
MAS_MESSAGE_COST	1	Cost per byte to send a message to another agent (1 = 2400 bps)
CONTROL_LOOP_COST	0	Base time to execute agent control loop code itself
PLAN_COST	24	Time to plan out one step of a plan
PATH_PLAN_COST	10	Time to plan out one step of a movement path
SENSOR_UPDATE_COST	100	Time between sensor updates
STATE_MERGE_COST	5	Time to merge new state information with current state information
STATE_EVALUATOR_COST	30	Time to evaluate the agent's state for events
Move action costs		
MOVE_BASE_COST	150	Time to move to an adjacent cell on the world (1m/s)
MOVE_FIRE_PENALTY	1000	Movement penalty (for path finding only) to move through fire
Rescue action costs		
RESCUE_COST	600	Number of ticks one (atomic) rescue action takes

Option Name	Example Value	Description
RESCUE_POINTS	500	Number of rescue points that accumulate per rescue action
RESCUE_TARGET	1000	Number of rescue points needed to rescue a civilian per difficulty level
FIRE_DANGER_PROXIMITY	10	How close a fire must be in cells to be considered a potential danger
Extinguish action costs		
EXTINGUISH_COST	600	Number of ticks one (atomic) extinguish action takes
EXTINGUISH_POINTS	1500	Number of extinguish points added per extinguish action
EXTINGUISH_TARGET	10000	Number of extinguish points a cell must accumulate to be extinguished
EXTINGUISH_RADIUS	2	Radius of extinguish action's effect
EXTINGUISH_PROXIMITY	10	Maximum reach of extinguish action
Civilian Agent properties		
CIV_DAMAGE_FREQ	12000	How often civilians take damage
CIV_ONFIRE_DAMAGE_FREQ	300	How often civilians take damage when they are on fire
Agents to use		
CIVILIAN_AGENT	Agent::Civilian	Agents controlled by the simulation
MOBILE_AGENT	Agent::ModelWorker	Agents that can move about and affect things
REMOTE_AGENT	Agent::ModelCoordinator	Agents that have no physical presence
Maps to use		
TERRAIN_FILE	terrain.png	Height map to use (PNG format)
REACHABILITY_FILE	reach.png	Reachability map to use (PNG format)
OBJECTS_FILE	objects.txt	List of objects in the scenario
Agent-specific configuration options		
MAP_AGE_LIMIT	18000	How old a part of the map must be before it is considered too out-of-date
VIS_DIST	20	Distance agents can see (cells)
WORKER_HEARTBEAT_FREQ	1500	Maximum time between MASMessages (5 seconds)

Table A.2: Available scenario configuration parameters.

A.3.3 The Terrain And Reachability Maps

The terrain map is a grey scale image in Portable Network Graphic (PNG) format. Each pixel of the PNG image corresponds to a cell of the world map. The value of each pixel indicates the height of a map cell, where black represents the lowest elevation and white the highest. Heights on the map are adjusted according to the HEIGHT_SCALE configuration parameter. Walls (or trenches) can be specified by ensuring creating an area that is higher (or lower) than the surrounding terrain by at more than MAX_SLOPE metres.

Accompanying the height map is a "reachability map," used to quickly determine what parts of the map are accessible. The reachability map can be automatically generated from the height map using the program **genReachable.pl**. **genReachable.pl** examines the heights of every cell

of the map and determines which groups of cells can and cannot be reached for every cell in the map. The result is a colour-coded "reachability" bitmap that is used to quickly determine whether an agent at a specific location can reach another location.

A.3.4 The Object Specification File

The object specification file describes the objects that are to be placed onto the terrain map. The file contains a sequence of object definitions, where each object definition has the following format:

```
object_type:  object_name {
   property1 = value1;
   property2 = value2;
   ...
}
```

No two objects may have the same `object_name`. Blank lines and comments (text preceded by a # character) are ignored. Valid `object_types` and their properties are described in Table A.3.

Object Type or Property	Example Value	Description
Civilian		Create an object of the CivilianAgent type
`Position`	*<54, 23>*	Position of the civilian agent on the map
`RescueDifficulty`	*23*	The difficulty of rescuing this civilian (higher required more time)
`MaxHP`	*50*	The maximum number of hit points the civilian can have
`StartHP`	*45*	The number of hit points the civilian starts the simulation with
`ConsoleOutput`	*yes*	Indicates whether the agent should send its log messages to the console
Fire		Create a circular area of fire
`Position`	*<10, 15>*	Centre of the area
`Radius`	*4*	Radius of the area
`Intensity`	*2.0*	The intensity of the fire in the area (larger is more intense)
FireBox		Create a rectangular area of fire
`Corner1`	*<100, 55>*	Upper-left corner of the rectangle
`Corner1`	*<110, 65>*	Lower-right corner of the rectangle
`Intensity`	*1.0*	The intensity of the fire in the area (larger is more intense)

Object Type or Property	Example Value	Description
MobileAgent		Create an object of the MobileAgent type
`Position`	*<5, 100>*	Position of the mobile agent on the map
`ConsoleOutput`	*yes*	Indicates whether the agent should send its log messages to the console
RemoteAgent		Create an object of the RemoteAgent type
`ConsoleOutput`	*yes*	Indicates whether the agent should send its log messages to the console

Table A.3: Available object types and their properties.

A.4 Program Documentation

A.4.1 Source Files

Table A.4 describes the files that make up the simulator.

Filename	Description
`Agent.pm`	This packages provides the base class for all the agent types (including civilians).
`Agent/Civilian.pm`	This packages models the simulated civilians.
`Agent/Deadline.pm`	This package implements a Deadline object.
`Agent/Event.pm`	These packages implement the various events (Opportunities, Barriers, and PCFs).
`Agent/Goal.pm`	This package implements a Goal object.
`Agent/Hardware.pm`	This file is included into the main agent, and provides the mechanisms needed to simulate specific agent hardware.
`Agent/Hardware/Motors.pm`	This module models the basic movement of a mobile agent. It is **not** the code that performs path-planning or obstacle avoidance; it is the code that simulates the low-level motors that drive the agent forward.
`Agent/ModelCoordinator.pm`	This package implements the coordinator agent described in Chapter 3.
`Agent/ModelCoordinator/Goal.pm`	This package implements a Goal object in the context of the coordinator.
`Agent/ModelCoordinator/PlanLib.pm`	This package implements the plans that execute all the possible goals that the coordinator may encounter.
`Agent/ModelCoordinator/PlanLib/Plan3.pm`	This file is included into the coordinator's plan library (PlanLib.pm). It implements the plan for goal #3, which is the coordinator's version of worker goal #1.
`Agent/ModelCoordinator/PlanLib/Plan4.pm`	This file is included into the coordinator's plan library (PlanLib.pm). It implements the plan for goal #4, which is the coordinator's version of worker goal #2.
`Agent/ModelCoordinator/Planner.pm`	This file is included into the coordinator agent, and implements the agent's planner.

Filename	Description
`Agent/ModelCoordinator/StateEvaluator.pm`	This file is included into the coordinator agent, and implements the global state evaluator.
`Agent/ModelCoordinator/TaskAllocator.pm`	This package provides a mechanism for selecting a worker to perform a task.
`Agent/ModelCoordinator/WorkerModel.pm`	This file provides methods for the coordinator to model the workers.
`Agent/ModelWorker.pm`	This package implements the Worker agent described in Chapter 3.
`Agent/ModelWorker/Goal.pm`	This package implements a Goal object in the context of a worker.
`Agent/ModelWorker/PlanLib.pm`	This package implements the plans that execute all the possible goals that workers may encounter.
`Agent/ModelWorker/PlanLib/Plan1.pm`	This file is included into the worker's plan library (PlanLib.pm). It implements the plan for worker goal #1.
`Agent/ModelWorker/PlanLib/Plan2.pm`	This file is included into the worker's plan library (PlanLib.pm). It implements the plan for worker goal #2.
`Agent/ModelWorker/PlanLib/Plan3.pm`	This file is included into the worker's plan library (PlanLib.pm). It implements the plan for worker goal #3.
`Agent/ModelWorker/PlanLib/Plan4.pm`	This file is included into the worker's plan library (PlanLib.pm). It implements the plan for worker goal #4.
`Agent/ModelWorker/PlanLib/Plan5.pm`	This file is included into the worker's plan library (PlanLib.pm). It implements the plan for worker goal #5.
`Agent/ModelWorker/Planner.pm`	This file is included into the worker agent, and implements the agent's planner and plan executor.
`Agent/ModelWorker/StateEvaluator.pm`	This file is included into the worker agent, and implements the agent's local state evaluator.
`genReachable.pl`	This program generates a reachability map from a terrain map based on the rules for movement in the configuration file.
`Logger.pm`	This packages provides a mechanism to easily write log messages to a file or to the console.
`Position.pm`	This packages encapsulates a two-vector, generally used as a position or displacement in the world model.
`RT.pm`	This module models basic properties of the simulation, such as the time.
`RT/IPC.pm`	This module models messaging between different parts of the same agent. Messages between agents are done by RT/MASMessage.pm. IPC messages take no time to send or receive.
`RT/MASMessage.pm`	This module models messaging between agents. Messages between different parts of the same agent are done by RT/IPC.pm.
`RT/Notifier.pm`	This module models notifications from hardware or software (aka interrupts).
`RT/Scheduler.pm`	This module models the real-time scheduler and CPU for an agent. The scheduler is used to simulate time by running certain actions at specific times.
`RT/Thread.pm`	This module models a thread of control within an agent. It uses the agent's scheduler to simulate execution time.
`sim`	This is the main program.

Filename	Description
`World.pm`	This packages models the world itself. It also handles drawing the world model to the display periodically.
`World/Fire.pm`	This package models fire in the world simulation. Fire is not an object in the world like agents are. Each cell can be on fire or not, and also has a ”heated” amount, which is how close the cell is to catching on fire.
`World/PathPlanner.pm`	This package implements an A* search to find a path from one point to another on a map, using either the actual World map, or an agent’s internal map.

Table A.4: Simulator source file descriptions.

Bibliography

[1] Ella M. Atkins, Tarek F. Abdelzaher, Kang G. Shin, and Edmund H. Durfee. Planning and Resource Allocation for Hard Real-Time, Fault-Tolerant Plan Execution. In *Proceedings of the third annual conference on Autonomous Agents*, pages 244–251. ACM Press, 1999.

[2] Tucker Balch and Ronald C. Arkin. Communication in Reactive Multiagent Robotic Systems. *Autonomous Robots*, 1(1):27–52, 1994.

[3] Gord Cormack. Lecture notes for CS452, Spring 2001.

[4] Edmund H. Durfee. Distributed Problem Solving and Planning. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 121–164. MIT Press, 2000.

[5] T. Estlin, R. Castano, B. Anderson, D. Gaines, F. Fisher, and M. Judd. Learning and Planning for Mars Rover Science. In *IJCAI 2003 workshop notes on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World modeling, planning, learning, and communicating*, Acapulco, Mexico, 2003.

[6] Michael Fleming and Robin Cohen. A Decision Procedure for Autonomous Agents to Reason About Interaction With Humans. In *The AAAI 2004 Spring Symposium on Interaction between Humans and Autonomous Systems over Extended Operation*, pages 81–86, 2004.

[7] H. Hexmoor, R. Falcone, and C. Castelfranchi, editors. *Papers from the IJCAI-2001 workshop on Delegation, Autonomy, and Control: Interacting With Autonomous Agents*, 2001.

[8] H. Hexmoor, R. Falcone, and C. Castelfranchi. *Agent Autonomy*. Kluwer Publishers, 2003.

[9] Sven Koenig. A Comparison of Fast Search Methods for Real-Time Situated Agents. In *The Third International Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 862–869, 2004.

[10] Rajiv Maheswaran, Milind Tambe, Pradeep Varakanthan, and Karen Myers. Adjustable Autonomy Challenges in Personal Assistant Agents: A Position Paper, 2004.

[11] Takeshi Morimoto, Kenji Kono, and Ikuo Takeuchi. YabAI: The First Rescue Simulation League Champion. In *RoboCup 2001: Robot Soccer World Cup V*, pages 49–59. Springer-Verlag, 2002.

[12] David J. Musliner, Edmund H. Durfee, and Kang G. Shin. World Modeling for the Dynamic Construction of Real-Time Control Plans. *Artificial Intelligence*, 74(1):83–127, 1995.

[13] Karen Myers and David Morley. Directing Agent Communities: An Initial Framework. In *Proceedings of the IJCAI-01 Workshop: Autonomy, Delegation, and Control: Interacting with Autonomous Agents*, 2001.

[14] Ranjit Nair, Takayuki Ito, Melind Tambe, and Stacy Marsella. Task Allocation in RoboCup Rescue Simulation Domain. In *The International Symposium on RoboCup*, 2002.

[15] Charles L. Jr Ortiz and Timothy Rauenbush. Dynamic Negotiation. In *Proceedings of the AAAI2002 Workshop on Planning with and for Multiagent Systems*, pages 64–71, Menlo Park, CA, 2002. AAAI Press.

[16] Paul Scerri, Lewis Johnson, David Pynadath, Paul Rosenbloom, Mei Si, Nathan Schurr, and Milind Tambe. A prototype infrastructure for distributed robot-agent-person teams. In *AAMAS-03*, 2003.

[17] Paul Scerri, David Pynadath, and Melind Tambe. Adjustable Autonomy for the Real World. In *Agent Autonomy*. Kluwer Publishers, 2004.

[18] M. Schillo. Self-Organization and Adjustable Autonomy: Two sides of the same medal? In *Proceedings of the AAAI2002 Workshop on Autonomy, Delegation, and Control: From Inter-agent to Groups*, pages 64–71, Menlo Park, CA, 2002. AAAI Press.

[19] D. Schreckenghost, C. Martin, P Bonasso, D. Kortenkamp, T. Miliam, and C. Thronesbery. Supporting Group Interaction Among Humans and Autonomous Agents. In *Proceedings of the AAAI2002 Workshop on Autonomy, Delegation, and Control: From Inter-agent to Groups*, pages 72–77, Menlo Park, CA, 2002. AAAI Press.

[20] S. Tadokoro, H. Kitano, T. Tomoichi, I. Noda, H. Matsubara, A. Shinjoh, T. Koto, I. Takeuchi, H. Takahashi, F. Matsuno, M. Hatayama, J. Nobe, , and S. Shimada. The RoboCup-Rescue: An International Cooperative Research Project of Robotics and AI for the Disaster Mitigation Problem. In *Proceedings of SPIE 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Controls (AeroSense), Conference on Unmanned Ground Vehicle Technology II*, 2000.

[21] Régis Vincent, Bryan Horling, Victor Lesser, and Thomas Wagner. Implementing soft real-time agent control. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 355–362, Montreal, Canada, 2001. ACM Press.

[22] Ryan Wegner and John Anderson. An Agent-Based Approach to Balancing Teleoperation and Autonomy for Robotic Search and Rescue. In *AI04 workshop on Agents Meet Robots*, 2004.

[23] Gerhard Weiss. Preface. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 1–23. MIT Press, 2000.