Experiments in High-Level Robot Control Using ConGolog — Reactivity, Failure Handling, and Knowledge-Based Search

by

Kenneth B L Tam

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements
for the degree of

Master of Science

Graduate Programme in Computer Science
York University
North York, Ontario

Abstract

Getting robots to do complex tasks is very difficult. The robot must not only be able to navigate through the world and execute various primitive tasks such as "go forwards by 5 cm" and "take a picture", but also support the specification and execution of complex behaviors, which requires reasoning about goals and actions. The occurrence of exogenous events and runtime failures, and having incomplete information of the environment have long been two major problems encountered by most robots operating in the real world. In this thesis, we develop an approach to building robust high-level control modules using the ConGolog language and prioritized interrupts to handle exogenous events and runtime failures. Such a module is implemented and tested on a mobile robot in a real environment for a mail delivery task. We also develop a *knowledge-based approach* to guide search for people in the presence of incomplete knowledge.

Acknowledgments

First of all, I would like to thank my thesis supervisors, Professor Yves Lespérance and Professor Michael Jenkin, for their guidance, advice, and patience in the development of this thesis. Without their foresight, this thesis would never have happened.

In addition, I would like to acknowledge the other members of my examination committee, Professor Milios Evangelos and Professor Neal Madras, for their patience in reviewing my thesis and useful suggestions in revising its contents.

I would like to express my gratitude to my friends here at York University and the University of Toronto, especially Xiaoyan Qian, Lewis Lo, Alex Tsui, Victor Wong, Ying Sun, My Hoa Dang, Kobert Lam, and Susan Chiu, for their support and encouragement. Thanks also go to Edwin Law for his help in preparing the video and pictures for this thesis and Iraj Mantegh for his contribution in the implementation of the low-level reactive system for the robot used in the experiment.

Finally, I owe my deepest gratitude to my family for their support, endurance, and patience in the past years.

Contents

List of Figures

Chapter 1

Introduction

Robots operating in the real world face many problems. Three of them manifest themselves in the following scenarios:

A mobile robot is going to a room X and, on the way to X, encounters some unanticipated semi-permanent obstacle such as a used cabinet waiting for removal, or a closed door, or a newly constructed partition. Instead of waiting for the removal of the obstacle, a reasonable solution is to try to find another path to X or give up on the goal and start performing another task. This requires a high-level robot control module which is able to trigger and terminate actions, monitor plan execution, identify failures in the plan, and recover from these failures.

In another scenario, a robot starts delivering an order Q whose service can be deferred. At some point during the delivery, an order P that is urgent and requires immediate attention comes in. The reasonable thing to do is to suspend the service to Q; serve the urgent order P; and then resume serving Q (because order P may miss its deadline otherwise). This requires a high-level robot control module which is able to react to significant events in its environment, by switching goals and actions, in a dynamic fashion at runtime.

In the last scenario, a robot is asked to find a person in an office with a large number of rooms without it having complete information about that person's location. In this case, the robot may need days to find the person if it simply searches every room one by one. The robot may even fail to catch the person if he/she keeps on moving from one room to another. However, the robot may have knowledge it can use to limit the search or increase its chances of finding the person. Or it may be able to acquire such knowledge as it searches, say by asking people it meets or finding evidence of where the person is. A good high-level robot control module should contain a dynamic knowledge base and be able to use it to perform search tasks more efficiently and flexibly.

The first scenario illustrates the problem of runtime failures, which occur when an action performed by the robot fails to attain its objectives. The second scenario is an example of the problems raised by the occurrence of exogenous events, which are events caused by other agents or nature and which may affect the robot. The last scenario shows the difficulties of conducting search for objects or people in the presence of incomplete information about the task domain.

In this chapter, we first examine the problems mentioned above, which are the motivation for this thesis. Then, an overview of the approach developed in this thesis will be provided. This will be followed by a statement of the formal research objectives, and finally, by an outline of this thesis.

1.1 The Problems

A *dynamic domain* is a complex environment which is subject to changes not under the agent's control and cannot generally be sensed or modeled perfectly. A robot operating in such a domain must face the three major problems mentioned earlier, i.e. runtime failures, the occurrence of exogenous events, and the robot having incomplete information about the task domain.

1.1.1 Runtime Failures

Runtime failures arise at two levels. In order to recover from the low-level runtime failures, we need a low-level control module. Similarly, we need a high-level control module to recover from the *high-level runtime failures*.

Low-level runtime failures occur when continuation of the robot's current action will harm human beings or the robot itself. Therefore, the robot must stop the action before anybody gets hurt. This is so time-critical that only a low-level control module can handle it.

In many situations, the low-level control module is unable to recover from failures because the cause of the failures may be something that exists in the world permanently or stay there for quite a long time. A partition constructed recently or a used cabinet waiting for removal are two good examples. In these situations, the only way to recover from failures is to plan a new path or switch to another goal. These can only be done at a higher level in the control hierarchy since planning and reasoning must be involved. It is this type of runtime failures that we are interested in in this thesis.

1.1.2 Exogenous Events

Exogenous events are events produced by other agents. For example, the urgent shipment order placed by the user is an exogenous event to the mail delivery robot in the second scenario. It is very important for a robot to handle these events in a timely manner. Since dealing with exogenous events may involve the re-scheduling of tasks and the switching of goals and actions, it can only be done at a higher level of the control hierarchy.

1.1.3 Incomplete Information about the Task Domain

As illustrated by the object/person search task in the third scenario, incomplete information about the task domain makes search difficult. With incomplete information about the location of the object, search may be inefficient because the robot has to look into every possible location. The situation gets even worse if it is seeking a mobile object because the mobility of the object keeps the search space from ever shrinking. In this case, the object may never be found.

To make search for an object more efficient and increase the chance of finding it, the robot can seek information about the object and update its knowledge base at runtime. The updated knowledge base can then be used to assist planning and reasoning, and explore the search space intelligently.

1.2 Our Approach

Motivated by the above problems, this thesis will develop a general mechanism to handle runtime failures and exogenous events reactively. In addition, a preliminary version of a knowledge-based approach to search for objects will also be developed.

1.2.1 Runtime Failure and Exogenous Event Handling

As suggested by the scenarios described earlier in this chapter, the occurrence of runtime failures and exogenous events calls for a reactive high-level control module that can select appropriate actions to deal with the failure or event.

In contrast to a *low-level robot control module* which mainly consists of control routines for sensors and effectors, a *high-level robot control module* is characterized by its ability to perform tasks at a higher level of abstraction. *High-level control* may involve switching between goals, triggering planning or replanning, starting and terminating actions, monitoring plan execution and taking corrective action(s) if something goes wrong. While most existing work in designing high-level control modules emphasizes the ability to plan and replan, we emphasize the ability to utilize the agent's knowledge base in reasoning and the use of a knowledge-based language to specify complex behaviors. In addition, for a control module to be *reactive*, it must be able to monitor the environment and respond to changes in it.

Since concurrency with prioritized interrupts supports reactivity and flexible resource allocation (see chapter 4 for details), we will use it to develop a runtime failure and exogenous event handling mechanism. A high-level control module based on this model will be built using the ConGolog agent programming language [30, 31, 13], which supports concurrent execution of threads and prioritized interrupts.

The module will be interfaced to other components in a hierarchical control architecture. The system will be tested by using it to drive a mobile robot to perform a mail delivery task in an office setting. Our robot will handle runtime failures (such as getting blocked on the way to someone's mailbox) and exogenous events (such as having an urgent shipment order arrive) reactively.

1.2.2 Knowledge-Based Search for Objects

Search with incomplete information about the task domain requires the robot to acquire knowledge at runtime and use its knowledge to reason about the next place to search. Knowledge helps to make search more time-efficient and improve the chance of finding an object (see chapter 5 for details). Therefore, we will argue that search for objects in the presence of incomplete information should be done using knowledge-based approach. A *knowledge-based approach* uses a knowledge base for reasoning and a knowledge-based language for specifying complex behaviors using conditions and goals at a high-level of abstraction, i.e. in terms that are natural in the task domain. Notice that ConGolog is also a good knowledge-based language in this sense.

The high-level control module built for runtime failure and exogenous event handling will be extended to incorporate a knowledge-based approach to search for objects. The resulting module extends the robot's capabilities by acquiring information and updating its knowledge base at runtime, and using the updated knowledge to select the next place to search. The module will be tested on a mail delivery task where mail must be delivered in person. The test will be performed in a simulated office environment in which people are allowed to move from one room to another.

1.3 Research Objectives

In brief, the main research objectives of this thesis are:

- to develop an approach to the design of high-level control modules which handle runtime failures and significant exogenous events reactively; our approach involves the use of concurrency and prioritized interrupts in the agent programming language ConGolog;
- to implement such a high-level robot control module and test it on a mobile robot in the real world;
- to develop a preliminary version of a mechanism to search for objects/people in the presence of incomplete information about the task domain using a knowledge-based approach; and
- to extend the high-level robot control module written for runtime failure and exogenous event handling by incorporating a knowledge-based approach to search for objects/people and test it in a simulated environment.

1.4 Outline of the Thesis

In chapter 2, we will discuss related work on robot architecture and high-level robot control, failure handling, and search tasks. Chapter 3 will cover the logical foundations and components of the ConGolog framework. In chapter 4, the approach we use to tackle exogenous events and runtime failures will be discussed. We will see how search for people in domains with incomplete information can be done using a knowledge-based approach in chapter 5. In chapter 6, we introduce the architecture of our robot. The experiments performed on the implementation of the design described in chapter 4 on our robot are discussed in chapter 7. This thesis is then concluded by a discussion of our contributions and possible future research directions.

Chapter 2

Background and Related Work

Motivated by the thesis objectives mentioned in chapter 1, this chapter is aimed at providing readers with some general background on research done in the following areas: robot architectures and high-level robot control, runtime failure and exogenous events handling, and techniques used in searching for objects.

2.1 Robot Architectures and High-Level Robot Control

As mentioned in chapter 1, high-level robot control may involve switching between goals, triggering planning or replanning, starting and terminating actions, monitoring plan execution and taking corrective actions if something goes wrong.

Approaches proposed for robot control can be grouped into 3 classes: they are the classical (also called "deliberative"), reactive (also called "behavioral"), and hybrid approaches. Each class has its own methodology and principles for building robots. This section tries to summarize the different forms of high-level robot control and their relation with the different robot architectures. For the sake of clarity, detailed descriptions of implemented robots/agents will not be provided in this chapter. Interested readers may consult appendix A for more details.

2.1.1 Deliberative Approaches

Deliberative approaches are founded on the *physical-symbol system hypothesis* of Newell and Simon [42]. According to the hypothesis, we can describe or represent everything in the world, such as physical objects, properties, relations, etc. as symbols. These symbols can then be manipulated by processes according to some symbolically coded sets of rules. Therefore, such systems are claimed to be capable of generating intelligent action.

An agent, if built using this kind of approach, usually has an explicit symbolic world model, a set of logical inference rules to operate on the world model and a centralized planning and reasoning module. Decisions are made mainly through logical reasoning which relies heavily on pattern matching and symbolic manipulation. In many cases, the planning or reasoning module works like a theorem prover. Control in this kind of agent rests mainly on the centralized planning and reasoning module. Every decision, no matter time-critical, has to be produced by this deliberation module and the resulting action is sent to the agent's executor for execution.

As discussed in Wooldridge *et al.* [61], if one wants to build an agent in this way, then there are at least two important sets of problems to be solved. These are the transduction problems and the representation/reasoning problems. The *transduction problems* are problems of translating the real world into an accurate symbolic representation in time for that description to be useful; the *representation/reasoning problems* concern how the complex world can be symbolically represented and how one can reason with these representations in time for the results to be useful.

There have been various attempts to build agents in this way. For example, Bratman *et al.* [7] propose the *Intelligent Resource-bounded Machine Architecture (IRMA)* which is based on an agent theory where beliefs, desires, and intentions are modeled. Vere *et al.* [60] argue that technologies currently available are sufficient to construct an intelligent autonomous agent. To provide evidence for this, they construct the *HOMER* agent which integrates various state-of-the-arts technologies. These include natural language understanding, temporal reasoning and planning, episodic memory, and general world knowledge. HOMER is tested in a simulated two-dimensional "Seaworld".

These approaches are good at handling tasks with multiple goals, in which planning and reasoning are required, but they have two major drawbacks. First, they suffer from the fact that not every situation in the real world can be represented completely and accurately [1]. Therefore, the actions chosen may be inappropriate. Second, these approaches lack reactivity to environmental changes. This is partly because these approaches rely heavily on mechanisms such as logical inference, pattern matching, search, instantiation and unification, most of which are computationally intractable [9, 2]. It also comes from the fact that every action has to be deliberated by a centralized planner and reasoner before being executed. As a result, agents with purely deliberative architectures are usually characterized by having a long sense-think-act cycle. Much current research has dealt with techniques for controlling deliberation to allow timely reaction. For example, IRMA tries to control deliberation by considering and selecting only plans that are consistent with the agent's current beliefs and intentions.

2.1.2 Reactive Approaches

Due to the drawbacks of the deliberative approaches, researchers such as Brooks and Maes have been advancing alternative approaches called "*reactive*" or "*behavioral*". Two key ideas are emphasized in these approaches.

1. "situatedness" and "embodiment": an agent can be called "intelligent" if it can live and operate in the real world [10, 61].
2. "intelligence" and "emergence": in contrast to the deliberative approaches which take intelligence to reside in the physical-symbol system [42], reactive approaches stress that "intelligent behavior arises as a result of an agent's interactions with its environment" [61].

Some of the architectures proposed to achieve reactivity are:

Subsumption Architecture

Brooks' *Subsumption Architecture* [8, 9] is based on the principle of subsumption. His architecture is essentially a control hierarchy of layers associated with particular tasks or behaviors. Each layer has its own competence and competes with others to exercise control over the robot. The lower layers perform tasks that are more primitive (reflexes), and have a higher precedence over layers further up the hierarchy.

Network Architecture

Maes' *Agent Network Architecture (ANA)* [37] is similar to Brooks' Subsumption Architecture in the sense that both architectures are composed of self-sufficient task-decomposed modules. However, Maes' model is different from Brooks' in that she treats each module as a node in a network (somewhat like a neural network) rather than as a layer in a hierarchy. Perhaps the main difference between an ANA and a neural work is that a node in a neural network does not have a clear meaning while a node in ANA corresponds to an activity [61].

Routines

Some researchers propose architectures based on *routines* which are patterns of interactions between the agent and its environment. This idea is inspired by the observation that most of our daily activities require little or no new abstract thinking. Most tasks are done in ways that we have learned from past experience. Therefore, the reactivity of a robot can be improved if all its responses to environmental changes can be "routinized". A detailed discussion of the use of routines can be found in [1] and [26]. The earliest agent built on the concept of routines is called *Pengi* [1] which is designed to play the video game Pengo (similar to PacMan).

Situated-Automata

Another architecture proposal is the *situated-automata* paradigm [29, 28] which analyses the relation between the real world and its symbolic representation. The paradigm emphasizes that an agent's actions must be determined by the current situation and reactive action plans are compiled from symbolic descriptions.

These architectural proposals all have the following characteristics: tight coupling of perception to action, no explicitly represented world model because they claim that "the world is the best model of itself", and hardwired architecture (i.e. no reasoning module). In addition, all reasoning and decision making have been pre-compiled into action selection mechanisms and are distributed over various layers or modules. Lastly, there are no guarantees of optimality in the decisions made.

Since every action is a direct mapping from perception, there is no runtime planning in these approaches. If they do engage in planning, the plan will be compiled down into hardwired digital circuitry during the agent's construction [21].

Characterized by a short processing cycle of perceived information and tight coupling of perception and actions, agents in these approaches are able to respond to environmental change quickly. Furthermore, decentralized reasoning and decision making imply that decisions from all relevant modules have to compete with each other to exercise control over the robot. Therefore, the agent can be controlled by a layer at one point and another later on if the agent sees that a layer is fit to exercise control in that situation. Since any layer can be the control module of the robot, control in these approaches is highly distributed.

Although agents in these approaches are capable of exhibiting complex behaviors and are able to execute them at a high speed, the resulting plan or action may be sub-optimal because of the lack of high-level reasoning and planning, and the emphasis on low-level reactivity. The robot may be forever reacting to changes in the environment without ever making progress on its global goals. Furthermore, it is not clear that such system are capable of dealing with tasks which involve multiple goals or are complex.

2.1.3 Hybrid Approaches

Many researchers have observed that neither a pure deliberative nor a pure reactive architecture is suitable for building agents that live in complex dynamic domains [9, 23]. They have also noticed that these two approaches can supplement each other in some ways so that the resulting agent is capable of dealing with complex tasks with multiple goals on one hand and responding to environmental changes reactively on the other. The *hybrid approaches* combine the approaches introduced in section 2.1.1 and 2.1.2.

There are two ways to combine these two approaches. One of them is to make deliberation part of the control loop, that is, to interleave the formulation, selection, and execution of plans. The plan used are hierarchical and/or universal (see below). Formulation and selection of the sub-plan are done by a planning and reasoning module with respect to the current perceived data and some precoded rules. Plans are executed by the agent's effectors. The effects of actions are monitored by sensors and sent back to the planning and reasoning module for further deliberation. High-level control in this architecture rests mainly in the reasoning and planning module.

There have been various attempts to build agents in this way. For example, the *Teleo-Reactive Approach* [45] blends the concepts of continuous feedback in control theory and some useful ideas of computer science such as binding parameters at runtime, subroutines, and recursion, and applies them to robot control. The *Procedural Reasoning System (PRS)* [23] emphasizes the importance of beliefs, desires, and intentions in reasoning and planning and, at the same time, reactivity. The *Reactive Action Package (RAP)* approach [17, 16] emphasizes modularity and the hierarchical expansion of plans to allow the agent to attain the goal no matter which contingencies it may run into.

Another way to combine deliberation and reactivity is to make components that provide them run in parallel, i.e. with the low-level reactive control system remaining active while the high-level deliberative system is running. Therefore, the robot is able to execute all kinds of actions which include some that are time-critical and require little or no deliberation, and others that require planning and reasoning. Unlike the previous method in which the low-level robot control system acts only as an executor of the instructions given by the high-level control system, the low-level control system in this method may seize the "privilege to control" from the high-level control system in critical situations such as when an obstacle is encountered.

Some of the attempts made in building robots in this way are: *TouringMachines* [15], which integrate Brooks' concept of task-decomposed layering structure and Bratman *et al.*'s view of using partial hierarchical plans to limit the amount of future deliberation while maintaining consistency across actions; InteRRaP [41], which uses a competence-driven layered architecture in which the reactive module and the planning module run in parallel; and *ATLANTIS* [20], which employs Agre *et al.*'s [2] plans-as-communication view of plans to link up the traditional AI planner and the reactive control mechanism. In Agre *et al.*'s *plans-as-communication view*, plans are used as one of the resources that the agent can draw upon when deciding what actions to take rather than as something that completely determines the agent's action. The executor retains the right to decide whether the "advise" from the planner is appropriate.

Although there have been lots of work on building agents based on the hybrid approaches, it is still not clear how the reactive and deliberative approaches are best reconciled so that the resulting agent can perform complex tasks in a dynamic domain robustly [23].

2.2 Runtime Failure and Exogenous Event Handling

This section tries to summarize some existing approaches to handle runtime failures and exogenous events. For an agent to handle runtime failures and exogenous events, it must not only be able to monitor its environment and identify the failure on time, but also be able to recover from it in a timely manner.

To deal with runtime failures and exogenous events, different techniques are employed by the different approaches mentioned above.

2.2.1 Reactive Approaches

Most researchers in the reactive school, such as Brooks [8, 9] and Maes [37], structure this problem as an architectural issue. Information about the agent's environment is collected continuously by various sensors and fed into a tightly coupled perception-action cycle to produce a suitable action. Since the cycle is actually a hardwired circuit, the robot can always respond to runtime failures with actions, which may be sub-optimal, in a timely manner. Although this technique is well suited for handling low-level runtime failures, it is unable to deal with failures whose solution requires the switching of goals, or failures which arise from situations that have not been anticipated.

2.2.2 Deliberative and Hybrid Approaches

In deliberative and hybrid approaches, exogenous event and runtime failure handling is usually framed partly as a problem of replanning and partly as a problem of whether sensing should be continuous or discrete. Since reasoning and planning are involved in this formulation, switching to different goals is now a possible solution to the problems caused by runtime failures.

The simplest approach to runtime failure handling is to generate a new plan to achieve the global goal when a failure or an exogenous event occurs. This strategy is implemented in *HOMER* [60]. A related approach is to generate a corrective plan when an action fails to achieve its expected effects, perhaps due to a runtime failure or an exogenous event; this corrective plan is to return the robot to a state where execution of the original plan can resume [12]. For an agent to make use of these strategies, it must contain a separate module to monitor its environment, either on a continuous or discrete basis. In addition, that module must also be able to trigger the planner once a failure or an exogenous event is detected. Replanning in this way can be very inefficient. Moreover, this technique lacks reactivity because the time involved in formulating a new plan may be very long.

In order to speed up the replanning process, some researchers suggest to use the current situation to index into a plan library and extract the most relevant plan. The plan will then be expanded and executed. This method is used in systems such as *PRS* [23] and *TouringMachines* [15]. Two types of plan that are commonly used by researchers to speed up the replanning process are hierarchical plans and universal plans.

2.2.3 Hierarchical Plans

A plan is hierarchical if it is specified at varying levels of details [55]. Details of the plan are filled out by recursively expanding into sub-plans stored in a plan library or generated at runtime as necessary. At any one time, only one part of the plan is being executed and only this part needs to be fully expanded into executable primitive actions.

Execution of a hierarchical plan usually starts with a top-level plan. At points in the plan, data about the environment can be collected and used to select a sub-plan to expand. Let us call these points the "expansion points" of a hierarchical plan. Execution of the plan is suspended when expansion of a sub-plan is required, and resumed when that expansion has been completed. This selection and execution of sub-plans will go on until the execution of the top-level plan is completed. Therefore, if a hierarchical plan is viewed graphically, it looks like a tree. Each node in the tree represents an expansion point which can be expanded into a sub-plan or an action. This type of plan is used by approaches such as *PRS* [23], InteRRap [20] and *ATLANTIS* [41].

However, since runtime failures can occur at any time, one should avoid sensing the environment only at the expansion points of a hierarchical plan. If a failure occurs between two expansion points, then there are chances that the failure may not be detected at all, or it may take a long time for the robot to realize that it has run into trouble.

2.2.4 Universal Plans

A plan is universal if it covers all possible situations that the agent is likely to face [57]. When a runtime failure occurs, the agent indexes the current situation into a set of plans, extracts the most relevant one, and executes the selected plan. Since the set of plans covers all possible situations, it is, in some sense, guaranteed that there must exist a plan that can recover from the failure. In addition, the selection of the next plan to expand or next action to execute is partly based on the current raw sensor data which is collected on a continuous basis, and partly based on some pre-coded rules. The *Teleo-Reactive Approach* [45] employs this sort of approach to handle runtime failures. For agents based on the situated-automata paradigm, it is this type of plan that gets compiled down to digital circuitry.

However, work involved in formulating a universal plan is tedious because the programmer has to iterate all possible situations mentally or with help of tools before the plan can be written up. In addition, since the world is always changing and there are always situations that cannot be foreseen, it is almost impossible to write down a complete set of plans to deal with all kinds of runtime failures. Even if all the possible situations could be written down completely, the set of possible situations and the resulting universal plan may be extremely large. Lastly, since reaction to the same situation may be different for different tasks, we will need different universal plans for each set of goals one may have. Therefore, it seems that universal plans are a bit impractical!

2.2.5 Concluding Remarks on Exogenous Event and Runtime Failure Handling

After reviewing previous work on exogenous event and runtime failure handling, I would like to point out two things. First, hierarchical plans and universal plans are not mutually exclusive. Second, in order to handle runtime failures robustly, sensing must be performed on a continuous basis [45]. Only in this way can failures be detected by the robot when they occur, and the failure handling routines be triggered on time.

2.3 Searching For Objects or People

If the robot is asked to find an object x within a given time period and it does not have any knowledge about the location of x, then the obvious approach to accomplish the task is to search every possible location in the environment. Since the robot has to visit every possible place one by one, the task may not be completed by its deadline, and the top-level schedule may fail.

Work has been done to speed up search by limiting the size of the search space. In *heuristic search*, one attaches a value to every search state and search is performed by traversing the search space according to these values [48, 44, 6]. However, heuristic search is for search in a model, not the real world. For these search algorithms to work well, the programmer must know the domain very well so that a complete world model can be built up at the time the program is being written up. In addition, the world must be static throughout the execution of the program. However, this is often impossible if the robot is looking for an object or a person in the real world for the following reasons: first, both people and objects are mobile and can be at different locations at different times; second, the real world, of which no complete model can be built, is dynamic in nature; and third, if there are lots of objects, it may be impossible to collect all the information beforehand.

Some researchers have proposed an alternative, knowledge-based approach, to search. The knowledge-based approach emphasizes the use of the agent's knowledge to guide search, which limits the size of the search space in most cases. The knowledge-based approach, in this sense, is like heuristic search. However, it differs from traditional heuristic search by using declarative knowledge instead of numbers to guide the search.

Various attempts have been made to conduct search using the knowledge-based approach and two of them are presented in [22] and [39]. Both of them emphasize the use of sensing to obtain information about the environment and hierarchical plans to improve reactivity. In the *SodaJack project* [22], the agent starts with a set of "currently known locations where an object may be located" and is asked to explore these locations systematically. At each location, a plan is developed to collect information about the environment. This phase of observation allows the agent not only to determine whether the object is found, but also to determine which is the best location to explore next if the object is not found in the current location. Obviously, sensing in this case is discrete because it is done only when SodaJack has reached a location. The ideas presented in [39] are similar except that the agents' behaviors are driven by some finite state machines and sensing is performed on a continuous basis.

2.4 Summary: On High-Level Control, Runtime Failure Handling and Search

As recognized by many researchers, while agents built according to the deliberative school lack reactivity, agents in the reactive school are unable to deal with complex tasks that require high-level reasoning and planning. The hybrid approaches are aimed at building agents from an architecture that combines these two approaches. In hybrid approaches, deliberation may or may not be part of the control loop. If deliberation is part of the control loop, the agent has to interleave the formulation, selection and execution of plans. If deliberation and reactivity run in parallel, then the agent must contain two separate modules, namely a low-level reactive and a high-level control module, that interact with each other in some ways. The low-level reactive module is responsible for all reactive actions that are time-critical or require little or no deliberation. The high-level control module is responsible for all reasoning and planning tasks.

While the reactive school has viewed exogenous event and runtime failure handling as something that is compiled into the control mechanism, other schools have framed it as a problem of replanning and sensing which is handled by dedicated mechanisms in the architecture instead. Perhaps our approach is like the reactive approach in that exogenous event and runtime failure handling must be written in the ConGolog program. We have observed that hierarchical plans alone are not sufficient to handle exogenous events and runtime failures and universal plans can never be complete. For an agent to handle exogenous events and runtime failures elegantly, it should monitor the environment continuously. In addition, it must possess facilities to stop the current action and change goals if necessary.

Suppose an agent is asked to find an object in its environment. The traditional search strategy is to search every possible place one by one until the object is found. Heuristic search helps in shrinking the size of the search space, but is not suitable for tasks such as searching for objects or people in dynamic domain. Therefore, some researchers have come up with an alternative called the knowledge-based approach of which the main idea is to use the agent's knowledge to guide the search. However, it seems that most of the work done so far has used static knowledge bases, i.e. knowledge bases that do not evolve with time, and where no attempt to obtain new information from any external source is made.

Chapter 3

The Situation Calculus and the ConGolog Framework

An agent's behavior can be viewed as a function of such parameters as abilities, knowledge, commitments, goals, etc. If the complex relations between every two elements are explored, one should not have difficulty in imaging how complicated the task of modeling and designing agents will be. In order to perform the task correctly, it is very important that the tools used be based on solid theoretical foundations. ConGolog [33, 32], which is an agent programming language based on a theory of action expressed in the Situation Calculus, is our choice for this.

In this chapter, we will examine the logical foundations on which ConGolog is built. Since it is a language based on the Situation Calculus, the first half of this chapter will provide a brief introduction to the Situation Calculus and the second half will cover the ConGolog framework.

3.1 The Situation Calculus

The *Situation Calculus* [38] is a first-order language for representing dynamically changing worlds. The world, at any time, is thought to be in a certain situation and all changes to the world are the result of the execution of some named *actions*. Actions in the Situation Calculus are instantaneous actions. They are denoted by function symbols and can be parameterized. A *world history*, which is a sequence of named actions, is represented by a first-order term called a *situation*. The *initial situation*, a situation in which no action has yet occurred, is denoted by the constant S_0. The term $do(\alpha, s)$ is used to represent the situation resulting from performing an action α in situation s. For example, $pickup(x)$ may stand for the action of picking up an object x, in which case $do(pickup(x), s)$ stands for the situation after the action $pickup(x)$ has been performed in situation s. The term $do(drop(x), do(goto(L), do(pickup(x), S_0)))$ denotes the world history consisting of the sequence of actions $[pickup(x), goto(L), drop(x)]$.

Relational fluents and *functional fluents* are relations and functions, respectively, whose values will vary from situation to situation. Since fluents may have different values in different situations, it is necessary for every fluent to take a situation term as their last argument so that a value can be associated with a situation. Therefore, the current "state" of the world is represented by the values of all the fluents in the current situation. For instance, $holding(robot, x, s)$ may mean an object x is being held by the robot in situation s and $loc(robot, s)$ may denote the position of the robot in situation s.

A domain of application is specified by three sets of axioms [35]. The first set contains the foundational axioms for situations, which specify the structure of situations. The second set contains the axioms for actions, which specify the preconditions of actions and axiomatize how the world changes when an action is performed. The last set contains the initial state axioms, which describes the world state when no action has yet occurred. The rest of this section gives a more detailed description of the Situation Calculus and can be skipped on a first reading.

3.1.1 Foundational Axioms for Situations

The *foundational axioms for situations* specify the structure of situations and are independent of the application domain. Situations in the Situation Calculus can be characterized in a similar way to natural numbers in Number Theory. Inspired by the Peano axioms for natural number, Lin and Reiter [36] came up with the following set of axioms:

$$S_0 \neq do(a, s), \tag{3.1}$$

$$do(a_1, s_1) = do(a_2, s_2) \ \supset \ (a_1 = a_2 \ \wedge \ s_1 = s_2), \tag{3.2}$$

$$\forall P. \, P(S_0) \ \wedge \ (\forall \, a, s) \, [P(s) \supset P(do(a, s))] \ \supset \ (\forall \, s) \, P(s), \tag{3.3}$$

$$\neg s < S_0, \tag{3.4}$$

$$s < do(a, s') \ \equiv \ (Poss(a, s') \ \wedge \ s \leq s'). \tag{3.5}$$

As explained in [36]: Axioms (3.1) and (3.2) are *unique names axioms for situations*, which help to eliminate finite cycles and merging when doing inference and induction. Axiom (3.3) is the *second-order induction axiom for situations.*

This axiom says that every state in the world history can be obtained by applying the binary function *do* to S_0 repeatly. Axioms (3.4) and (3.5) altogether define the relation "$<$" inductively. $s_1 < s_2$ means s_2 can be reached from s_1 by performing one or more actions in s_1. $s_1 \leq s_2$ is an abbreviation for $s_1 < s_2 \vee s_1 = s_2$.

3.1.2 Axiomatizing Actions

Actions in a domain are axiomatized by three sets of axioms: the action precondition axioms, successor state axioms, and unique names axioms for actions.

Action Precondition Axioms

The *action precondition axiom* of an action specifies all the major qualifications which must be true for the action to be possible. These axioms are of the following form:

$$Poss(\alpha(\vec{x}), s) \ \equiv \ \pi_\alpha(\vec{x}, s) \tag{3.6}$$

where $\pi_\alpha(\vec{x}, s)$ is a formula specifying the preconditions of the action $\alpha(\vec{x})$.

For example, the precondition axiom for the action $pickup(x)$ may look like

$$Poss(pickup(x), s) \ \equiv \ [(\forall\, z)\, \neg holding(robot, z, s)] \ \wedge \ \neg heavy(x).$$ [1]

This says that it is possible to pickup an object x in situation s if and only if the robot is holding nothing in s and x is not heavy.

[1] Examples in this section are taken from [35]

Successor State Axioms

As the execution of an action will not change the value of every fluent, therefore, for each action, there exists two sets of fluents. One of them contains the fluents whose value will be affected by the performed action and the other contains the fluents whose value is left unchanged. Changes to fluents in the former set are axiomatized by effect axioms. The fact that fluents in the latter set are invariant is specified by frame axioms. As we will see, under some assumptions, it is possible to generate a single successor state axiom for each fluent from the effect axioms. The *successor state axiom* for a fluent describes how the fluent is affected by all possible actions.

The *effect axioms* for an action specify how the action changes the world. They establish the "causal laws" for the application domain. There are two sets of effect axioms: the positive effect axioms and negative effect axioms. Each *positive effect axiom* of a fluent specify the conditions under which the execution of a particular action can cause the fluent to become true in the successor state. For example, the following positive effect axiom says that a robot dropping a fragile object x causes it to be broken:

$$Poss(drop(r,x),s) \land fragile(x) \supset broken(x, do(drop(r,x),s)). \qquad (3.7)$$

As a second example, the following positive effect axiom says that a bomb exploding next to an object causes it to become broken:

$$Poss(explode(b),s) \land nexto(b,x,s) \supset broken(x, do(explode(b),s)). \qquad (3.8)$$

Each *negative effect axiom* of a fluent specify the conditions under which the execution of a particular action can cause the fluent to become false in the successor state. For example, the effect on the fluent *broken* when an object is repaired by somebody is specified in the following negative effect axiom:

$$Poss(repair(r, x), s) \supset \neg broken(x, do(repair(r, x), s)). \tag{3.9}$$

Frame axioms specify the action invariants of the domain, i.e. aspects of the world state which will not be affected by the execution of an action. For example, a robot dropping things will not affect the color of an object:

$$Poss(drop(r, x), s) \wedge color(y, s) = c \supset color(y, do(drop(r, x), s)) = c.$$

In axiomatizing actions in an application domain, a huge number of axioms have to be written down and most of them are actually frame axioms. This is known as the "*frame problem*" [38]. This huge number of axioms makes inference and reasoning inefficient and the process of axiomatization is prone to errors and omissions. A number of attempts have been made to solve the frame problem in the last two decades. Since the approach used by ConGolog to handle the frame problem is based on Reiter's, only this solution will be discussed here.

By synthesizing proposals made by Pednault [46], Haas [25] and Schubert [58], and by making the Causal Completeness Assumption, Reiter [51] proposes a solution to the frame problem, which drastically reduces the number of axioms. His solution starts by collecting all effect axioms about a given fluent and then combining all positive effect axioms into a single axiom which takes the form

$$Poss(a, s) \wedge \gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)), \tag{3.10}$$

and combining all the negative effect axioms into another single axiom which is of the form

$$Poss(a,s) \wedge \gamma_F^-(\vec{x},a,s) \supset \neg F(\vec{x}, do(a,s)), \tag{3.11}$$

where $\gamma_F^+(\vec{x},a,s)$ and $\gamma_F^-(\vec{x},a,s)$ are first-order formula with free variables $\vec{x}, a$ and s. Axiom (3.10) is called *the General Positive Effect Axiom* for fluent F and Axiom (3.11) is called *the General Negative Effect Axiom* for fluent F. They specify under what conditions the performance of action a in situation s will make fluent F true and false, respectively, in the successor state $do(a,s)$. For example, the general form of the positive effect axioms (3.7) and (3.8) is

$$\begin{aligned} Poss(a,s) \wedge {} & [(\exists\, r)(a = drop(r,x) \wedge fragile(x)) \vee \\ & \quad (\exists\, b)(a = explode(b) \wedge nexto(b,x,s))] \\ & \supset broken(x, do(a,s)). \end{aligned}$$

Similarly, the negative effect axiom (3.9) can be transformed into

$$Poss(a,s) \wedge (\exists\, r) a = repair(r,s) \supset \neg broken(x, do(a,s)).$$

The *Causal Completeness Assumption* says that the axioms (3.10) and (3.11) capture all conditions in situation s under which an action a can cause fluent F to become true (or false) in the successor situation, i.e. the set of effect axioms provided by the axiomatizer is "complete". Given this assumption, it is reasonable to take the axioms (3.10) and (3.11) for a fluent and transform them into a single axiom called "the successor state axiom"[2] which is of the form

$$\begin{aligned} Poss(a,s) \supset {} & [F(\vec{x}, do(a,s)) \equiv \\ & \gamma_F^+(\vec{x},a,s) \vee (F(\vec{x},s) \wedge \neg\gamma_F^-(\vec{x},a,s))]. \end{aligned} \tag{3.12}$$

[2]This formulation assumes there are no ramifications (state constraints).

For example, the successor state axiom of the fluent *broken* would be

$$
\begin{aligned}
Poss\,(a,s) \;\supset\; [&broken(x, do(a,s)) \;\equiv \\
&(\exists\, r)\,(a = drop(r,x) \;\wedge\; fragile(x,s)) \vee \\
&(\exists\, b)\,(a = explode(b) \;\wedge\; nexto(b,x,s)) \vee \\
&broken(x,s) \;\wedge\; \neg(\exists\, r)\; a = repair(r,x)].
\end{aligned}
\tag{3.13}
$$

This says that an object x will be broken in the successor state if and only if one of the following cases applies: first, x is fragile and has just been dropped by some r; second, a bomb b which is next to x has just exploded; and third, x is broken in the previous state and has not been repaired by any r. With this, there is no need for any frame axioms. Thus, for example, if the action $paint(r,x)$ is executed in situation s, then (3.13) can be instantiated and, by the unique names axioms for actions, simplified to

$$Poss\,(paint(r,x),s) \;\supset\; [broken(x, do(paint(r,x),s)) \;\equiv\; broken(x,s)].$$

and the result implies the frame axioms

$$Poss(paint(r,x),s) \;\wedge\; broken(x,s) \;\supset\; broken(x, do(paint(r,x),s)$$

and

$$Poss(paint(r,x),s) \;\wedge\; \neg broken(x,s) \;\supset\; \neg broken(x, do(paint(r,x),s).$$

The conciseness and perspicuity of Reiter's solution, as mentioned in [51], relies on:

- quantification over actions,
- the assumption that a fluent is affected only by relatively few actions, and
- the Causal Completeness Assumption.

Unique Names Axioms For Actions

The *unique names axioms for actions* guarantee that actions with different names or arguments are different. To express these formally, we have (as stated in [50]):

For distinct action names A and B,

$$A(\vec{x}) \neq B(\vec{y}). \tag{3.14}$$

Identical actions have identical arguments:

$$A(x_1, \cdots, x_n) = A(y_1, \cdots, y_n) \supset x_1 = y_1 \ \wedge \ \cdots \ \wedge \ x_n = y_n. \tag{3.15}$$

3.1.3 Initial State Axioms

The *initial state axioms* describe the world in the initial situation S_0. As mentioned, the initial situation is a situation in which no action has yet occurred. For example, the initial state axiom

$$nexto(b, x, S_0)$$

says that an object b is next to an object x in the initial situation and the initial state axiom

$$color(y, S_0) = red$$

means that the color of an object y is initially red.

Let us also mention in passing that the knowledge of agents can be modeled explicitly in the Situation Calculus [40]. Moreover, Reiter's solution to frame problem can be extended to knowledge-producing actions [56]

3.2 ConGolog

ConGolog (*CONcurrent GOLOG*) [33, 32, 13] is an agent programming language based on the Situation Calculus. It is an extended version of the Golog (*alGOl in LOGic*) [35] language, to which it adds concurrent processes, prioritized interrupts, and exogenous actions handling.

Both Golog and ConGolog are designed as a compromise between classical planning and detailed programming [32]. The programmer provides both axioms describing the domain and a possibly non-deterministic program specifying the agent's behavior. The interpreter can reason about how to get to a final state of the program using the axioms. The programmer need not specify in detail how and when an action should be performed because the details can be filled in by the interpreter when the program is executed.

3.2.1 The Relation between Planning and ConGolog Program Execution

Given a set of background axioms, which defines an application domain, a classical planning task is to find a sequence of legal action $\vec{a}$ that will lead us to a state s where the goal formula ϕ is satisfied [13]. Writing this down formally, we have

$$Axioms \models Legal(\vec{a}, S_0) \wedge \phi(do(\vec{a}, S_0)), \tag{3.16}$$

where $do(\vec{a}, S_0) \stackrel{\text{def}}{=} do([a_1, \cdots, a_n], s)$ is an abbreviation for

$$do(a_n, do(a_{n-1}, \cdots, do(a_1, s)) \cdots))$$

and where $Legal([a_1, \cdots, a_n], s)$ stands for

$$Poss(a_1, s) \wedge \cdots \wedge Poss(a_n, do([a_1, \cdots, a_{n-1}], s)).$$

To obtain such a sequence of legal action as a side-effect of the process of finding a proof for (3.16) is one way to specify the *plan synthesis* task.

However, classical planning is infeasible in complex dynamic domains where incomplete knowledge and exogenous events are a common phenomenon. In [35], Levesque *et al.* argued that high-level program execution is a more practical alternative. Their idea is that, "instead of searching for a sequence of actions that takes the robot from an initial state to some goal state, the task is to find a sequence of actions that constitutes a legal execution of some high-level program." [34] Therefore, the execution of a ConGolog program σ can be defined as

$$Axioms \models Do(\sigma, S_0, do(\vec{a}, S_0)) \tag{3.17}$$

where $Do(\sigma, s, s')$ is a formula of the Situation Calculus saying that executing the high-level program σ in state s may lead to situation s'. A high-level program in this context is one "whose primitive instructions are domain-dependent actions of the robot, whose tests involve domain-dependent predicates that are affected by the actions, and whose code may contain non-deterministic choice points where lookahead is necessary to make a choice that leads successful termination" [34].

3.2.2 The Constructs of ConGolog

A ConGolog program is defined in terms of a set of primitive actions and a set of primitive fluents, using a set of constructs described below. The programmer provides an action precondition axiom for each action and a successor state axiom for each primitive fluent, as well as initial state axioms. Programs are usually composed of a number of procedures that accomplish particular tasks.

The syntax of the ConGolog constructs is as follows:

α,	primitive action
$\phi?$,	wait for condition ϕ to become true
$(\sigma_1 \,;\, \sigma_2)$,	sequence[3]
$(\sigma_1 \mid \sigma_2)$,	nondeterministic choice between actions σ_1 and σ_2
$\pi\vec{x}.\,\sigma$,	nondeterministic choice of arguments, $\vec{x}$
σ^*,	nondeterministic iteration
if ϕ **then** σ_1 **else** σ_2,	conditional
while ϕ **do** σ,	loop until $\neg\phi$
proc $\beta(\vec{x})\ \sigma$,	procedure definition
$(\sigma_1 \parallel \sigma_2)$,	concurrent execution of σ_1 and σ_2
$(\sigma_1 \gg \sigma_2)$,	concurrency with different priorities
$\sigma^{\parallel}$,	concurrent iteration of two or more instances of σ
$< \vec{x} \,:\, \phi \rightarrow \sigma >$,	interrupt[4]

A formal definition of the semantics of the above constructs based on the Situation Calculus is provided in [13]. For the sake of simplicity and brevity, we will not discuss the formal semantics here. Instead, we provide an informal overview of the meaning of the constructs.

A Simple Example of a ConGolog Program

Before we go on, let us look at a very simple program written in ConGolog. The program gets a simulated robot to perform a mail-delivery task. The declarative

[3] σ ranges over programs, i.e. complex actions

[4] $< \vec{x} \,:\, \phi \rightarrow \sigma >$ can be defined in terms of the interrupts in [13] as $< (\exists\,\vec{x})\phi \rightarrow (\pi\,\vec{x})[\phi?;\sigma] >$.

part of the program goes as follows. It contains five primitive actions: *goTo(p)*, which directs the robot to go to a place p, *pickUpShipment(n)* and *dropOffShipment(n)*, which get the robot to pick up and drop off a shipment n, and *ackShipmentOrder(n)*, which gets the robot to acknowledge shipment order n; also there is the exogenous action *orderShipment(c,r)* that is performed by a user c to order a shipment for a recipient r. Below are the action precondition axioms for the actions:

$$Poss(goTo(p), s) \equiv p = jeff \lor p = michael \lor p = yves \lor \\ p = hector \lor p = centralOffice,$$

$$Poss(pickUpShipment(n), s) \equiv (\exists c).shipmentToPickUp(n, s) \land \\ sender(n, s) = c \land robotPlace(s) = c,$$

$$Poss(dropOffShipment(n), s) \equiv (\exists c).shipmentOnBoard(n, s) \land \\ recipient(n, s) = c \land robotPlace(s) = c,$$

$$Poss(ackShipmentOrder(n), s) \equiv shipmentOrdToAck(n, s).$$

$$Poss(orderShipment(c, r), s) \equiv person(c) \land person(r),$$

The second of these, for instance, says that the robot may pick up shipment n when it is ready for pickup and the robot is at the sender's location.

The state of the world is modeled by seven fluents. These include *shipmentOrdToAck(n,s)*, which holds when shipment order n is waiting for acknowledgment in situation s, *shipmentToPickUp(n,s)*, which holds when shipment n is ready for pickup, and *shipmentOnBoard(n,s)*, which holds when shipment n is onboard the robot. The sender and recipient of a shipment n are kept track of by the functional fluents *sender(n,s)* and *recipient(n,s)*, respectively. Finally, the *robotPlace(s)* functional fluent represents the robot's position and the *shipmentCtr(s)* functional fluent denotes the number of the next shipment order. Their successor state axioms are as follows:

$$
\begin{aligned}
&shipmentOrdToAck(n, do(a, s)) \equiv \\
&\quad (\exists\, c, r)\, [\, a = orderShipment(c, r) \ \wedge \ shipmentCtr(s) = n \,] \vee \\
&\quad [\, shipmentOrdToAck(n, s) \ \wedge \ a \neq ackShipmentOrder(n) \,],
\end{aligned}
$$

$$
\begin{aligned}
&shipmentToPickUp(n, do(a, s)) \equiv \\
&\quad a = ackShipmentOrder(n) \vee \\
&\quad [\, shipmentToPickUp(n, s) \ \wedge \ a \neq pickUpShipment(n) \,],
\end{aligned}
$$

$$
\begin{aligned}
&shipmentOnBoard(n, do(a, s)) \equiv \\
&\quad a = pickUpShipment(n) \vee \\
&\quad [\, shipmentOnBoard(n, s) \ \wedge \ a \neq dropOffShipment(n) \,],
\end{aligned}
$$

$$
\begin{aligned}
&sender(n, do(a, s)) = c \equiv \\
&\quad [\, (\exists\, r)\ a = orderShipment(c, r) \ \wedge \ shipmentCtr(s) = n \,] \vee \\
&\quad c = sender(n, s),
\end{aligned}
$$

$$
\begin{aligned}
&recipient(n, do(a, s)) = r \equiv \\
&\quad [\, (\exists\, c)\ a = orderShipment(c, r) \ \wedge \ shipmentCtr(s) = n \,] \vee \\
&\quad r = recipient(n, s),
\end{aligned}
$$

$$
\begin{aligned}
&robotPlace(do(a, s)) = p \equiv \\
&\quad a = goTo(p) \vee \\
&\quad (\forall\, p')[\, robotPlace(s) = p \ \wedge \ a \neq goTo(p') \ \wedge \ p \neq p' \,].
\end{aligned}
$$

$$
\begin{aligned}
&shipmentCtr(do(a, s)) = n \equiv \\
&\quad [\, (\exists\, c, r)\ a = orderShipment(c, r) \wedge \\
&\quad\ \ shipmentCtr(s) = n' \ \wedge \ n = n' + 1 \,] \vee \\
&\quad [\, (\forall\, c, r)\, a \neq orderShipment(c, r) \ \wedge \ n = shipmentCtr(s) \,]
\end{aligned}
$$

For example, the fluent *shipmentToPickUp(n,s)* becomes true when shipment n is acknowledged, and stays true until it is picked up by the robot.

In addition, we also have the following initial state axioms in the declarative part of the program:

$$robotPlace(S_0) = robotHome,$$

$$shipmentCtr(S_0) = 1.$$

The procedural part of the program is as follows:

$$\begin{array}{l}
\langle n\colon shipmentOrdToAck(n) \rightarrow ackShipmentOrder(n)\rangle \\
\gg \\
\langle (\exists\, n) shipmentToPickUp(n) \rightarrow serveCustomer\rangle \\
\gg \\
\langle robotPlace \neq centralOffice \rightarrow goTo(centralOffice)\rangle
\end{array}$$

with *serveCustomer* being the following procedure:

$$\begin{array}{l}
\mathbf{proc}\ serveCustomer \\
\quad (\pi n, c, r)[(sender(n) = c \wedge recipient(n) = r \wedge shipmentToPickUp(n))?; \\
\qquad\qquad goTo(c); pickUpShipment(c); \\
\qquad\qquad goTo(r); dropOffShipment(r)] \\
\mathbf{end}
\end{array}$$

This program starts up with three threads at different priorities. The highest priority thread handles the task of acknowledging new shipment orders. The middle priority thread directs the robot to deliver shipments from one customer to another as long as there are some shipments to pick up. The lowest priority thread gets the robot to return to the *centralOffice* when it has nothing to do. Each of these is coded up as an interrupt. We explain the details below.

Concurrency and Priorities

When a thread in CONGOLOG encounters an action whose preconditions are false or a false condition in a test, its execution is suspended and another unblocked thread is picked for execution. An unblocked thread with the highest priority gets to execute. The suspended thread can become active again once the action becomes possible or the test condition becomes true. For example,

$$(\sigma_1\,;\,\phi?\,;\,\sigma_2) \gg \sigma_3$$

might be executed as follows: σ_1 is executed first and then the first thread blocks on the condition ϕ, which we assume is false; then σ_3 from the lower priority thread gets to execute at this point, which might make ϕ true; since ϕ has now become true, σ_2 can then be executed and the program will terminate.

The example CONGOLOG program presented earlier contains a more concrete example of this feature. The program starts up with three threads at different priorities. The first thread always gets executed when there is a shipment order to acknowledge and the second thread gets executed only if the first thread is blocked, i.e. if the conditon *shipmentOrdToAck* is false. The first thread will become active again once the condition becomes true. Similarly, the third thread gets executed only if the first and second thread are both blocked.

Interrupts

An interrupt $< \vec{x} : \phi \rightarrow \sigma >$ consists three parts: a sequence of variables $\vec{x}$, a trigger condition ϕ, and a body σ. Every interrupt monitors the environment on a continuous basis. The body of an interrupt will be executed as long as the interrupt gets control from higher priority processes and its trigger condition ϕ is satisfied for some binding of the variables $\vec{x}$. Once it has completed its execution, it can be triggered again. For instance, in our example program, we have the following interrupt:

$$\langle (\exists\, n) shipmentToPickUp(n) \rightarrow serveCustomer \rangle$$

This means that the procedure *serveCustomer* will be executed whenever there exists a shipment waiting for pickup, i.e. *shipmentToPickUp* is true (in this example, there is no variable to bind in the interrupt).

When interrupts are prioritized (for example, $\sigma_1 \gg \sigma_2$), the one with higher priority (i.e. σ_1) will always take control of the robot if its trigger condition is true. Interrupts with lower priority (i.e. σ_2) may take control of the robot only when σ_1 is blocked, or its trigger condition is false. If the trigger condition of σ_1 becomes true when the robot is controlled by σ_2, σ_1 will seize control from σ_2 which will be suspended until σ_1 blocks again. Therefore, in ConGolog, if exogenous events and runtime failures are handled by processes with higher priority, they will be dealt with in a timely manner. In our example program, we have another interrupt at the highest priority:

$$\langle\, n\colon shipmentOrdToAck(n) \rightarrow ackShipmentOrder(n)\, \rangle.$$

With this arrangement, whenever a new shipment order is detected, this thread will immediately be triggered and an acknowledgment will be made. Other enabled threads will be suspended while this occurs. This assures that new orders get acknowledged quickly.

Nondeterminism

Both GOLOG and CONGOLOG support nondeterministic choice between actions and of action arguments. Suppose there are two programs σ_1 and σ_2 and we use the construct $(\sigma_1 \mid \sigma_2)$. Then the interpreter is free to choose and execute either σ_1 or σ_2. The case of choosing arguments $(\pi \vec{x}.\, \sigma)$ is similar.

For instance, the example program presented above uses nondetermistic choice of arguments in the *serveCustomer* procedure. Suppose we have two shipment orders placed initially: $orderShipment(jeff, hector)$ and $orderShipment(michael, yves)$. Then, the nondeterministic choice operator $(\pi\, n, c, r)$ will select some bindings for the variables, and for the triples *<1,jeff,hector>* and *<2,michael,yves>*, the test will succeed. So one of these triples will be selected and the body executed for it.

Let us look at another example involving nondeterministic choice between actions. Suppose we have the following program:

$$(a_1; \phi?) \mid a_2$$

where both a_1 and a_2 are primitive actions that can always be executed successfully. Suppose we try to execute this starting in situation S_0. The interpreter will first pick either a_1 or a_2 to execute at random. If a_2 is selected and executed, then the program will terminate in situation $do(a_2, S_0)$. Suppose the interpreter first picks a_1 to execute, which is followed by a test of ϕ. If ϕ is true, the program terminates in $do(a_1, S_0)$. However, if ϕ is false, then execution cannot proceed (there are no other enabled threads here), so the interpreter will *backtrack* over the selection of a_1, then a_2 will be selected instead of a_1 and the program will terminate in situation $do(a_2, S_0)$.

Use of the Declarative Part of the Program

The declarative part of the program includes the action precondition axioms, the successor state axioms, and the initial state axioms. The successor state axioms and initial state axioms are used to determine whether a test condition in the program is true. For example, to determine the truth value of the condition

$$(sender(c) \wedge recipient(r) \wedge shipmentToPickUp(n))?$$

we have to determine the values of the fluents *sender*, *recipient*, and *shipmentToPickUp* using the successor state axioms and the initial state axioms. The three types of axioms are also used to determine whether primitive actions are executable. For example, to determine the executability of the action *pickUpShipment* in a given situation, we first determine that it depends on the values of the fluents *shipmentToPickUp*, *sender*, and *robotPlace* in the current situation using the action precondition axioms, and then we must use the other types of axioms to determine the current values of these fluents.

An Additional Iteration Construct

Finally, let us introduce an additional construct that will be used in chapter 5. Its syntax is:

$$\textbf{for } (x : P \textbf{ until } \phi) \textbf{ do } [\sigma] \textbf{ endFor}$$

This is a loop construct that executes the body σ for each element x of the set P. As well, the loop terminates immediately if, after a repetition, ϕ is true. It can be defined as follows:

$$
\begin{array}{l}
\textbf{for } (x : P \textbf{ until } \phi) \textbf{ do } [\sigma] \textbf{ endFor} \stackrel{\text{def}}{=} \\
\quad \textbf{proc } N(P') \\
\qquad \textbf{if } (P' \neq \emptyset) \textbf{ then} \\
\qquad \pi x[\,(x \in P')?; \\
\qquad\quad \sigma; \\
\qquad\quad \textbf{if } \neg\phi \textbf{ then } N(P' - \{x\}); \\
\qquad\quad \textbf{else noOp } ;] \\
\qquad \textbf{else noOp } ; \\
\quad \textbf{end} \\
\quad N(P)
\end{array}
$$

where N is a new procedure name.

3.2.3 Others Characteristics of ConGolog Programs

Other than non-deterministic choice of actions or arguments, concurrent execution of threads and prioritized interrupts, ConGolog programs also have the following characteristics:

Support Abstraction of the Tasks/Domains

ConGolog supports abstraction of tasks/domains at different levels. The higher the abstraction, the fewer details about a task have to be written into the program, and the more expressive and natural the "vocabularies" are. With abstraction, one needs not reason all the way down to the machine instruction level; this would be inefficient and unnecessary. One should choose the level of abstraction that is appropriate for the task. Of course, an implementation for the primitive actions in terms of the robot's architecture must be provided; but the ConGolog programs will be independent of these details.

Easier to Extend and Modify Behaviors

Since ConGolog supports high-level abstraction of the task/domain, programs written in the language are clean, natural, and highly modular. This in turn, makes extension and modification easier.

Easier to Debug and Verify

As programs written in ConGolog are expressed in ways that are natural for the domain, behaviors can be specified easily and accurately. Debugging and verification are easier because everything in the program reflects the domains or the tasks themselves.

3.2.4 The Implemented ConGolog Interpreter

The ConGolog interpreter used in this thesis is implemented in Quintus Prolog. Because some ConGolog operators are already used by the Prolog language, there are some differences in the syntax used in the implementation and that presented so far. These are as follows:

- Since semicolon (;) for sequential execution already has a special meaning in Prolog, we use Prolog lists instead;
- We use "$\sigma_1 \# = \ \sigma_2$" and "$\sigma_1 \# > \ \sigma_2$" instead of "$\sigma_1 \parallel \sigma_2$" and "$\sigma_1 \gg \sigma_2$", respectively;
- All fluents in our implementation are reified and appear as arguments to the predicate *holds*.

3.3 ConGolog as a Knowledge-Based Language

ConGolog can be considered a good knowledge-based language for the following reasons:

- For a language to follow the knowledge-based approach, it must be able to specify a robot's behavior at a high level of abstraction. ConGolog, which is able to support different levels of abstraction of tasks/domain, satisfies this requirement.
- Specifications are often transferable between tasks/domains. Being able to support specifications at different levels of abstraction, ConGolog helps in building knowledge-based agents that can be easily modified to do different tasks in different domains.
- ConGolog supports reasoning about action based on declarative specifications and its interpreter automatically maintains a world model. Therefore, the behavior of a knowledge-based agent built using ConGolog can be verified, debugged, extended and maintained easily.

3.4 Other Agent Languages

In this section, we will take a look at some other agent languages and see how ConGolog differs from them.

3.4.1 Agent-0

Shoham's *Agent-0* [59] is a language addressing the integration of belief, commitment and ability. As described by Wooldridge *et al.* in [61], the framework of this language is essentially "a quantified multi-modal logic, allowing direct reference to time". In Agent-0, an agent is specified in terms of a set of initial belief and commitments, and a set of commitment rules. In addition, the language also provides some built-in simple communication acts. All messages are constrained to be one of the following types: "inform", "request" and "unrequest". "Inform" messages cause modifications of belief; "request" and "unrequest" messages cause modifications of commitments.

However, as mentioned in [35], Agent-0 does not provide any mechanism to automatically maintain an agent beliefs as actions are performed as ConGolog does. In addition, it handles only a few types of complex actions.

3.4.2 Concurrent MetateM

Fisher's *Concurrent* MetateM [19] is another agent language that is based on logic. Agents in a *Concurrent* MetateM system run concurrently and communicate with each other via asynchronous broadcast message passing. Each agent's behavior is specified in a subset of temporal logic and the agent exhibits this behavior by building a logical model of the specification iteratively. However, as mentioned in [31], a "limitation of the approach is that neither the interactions between agents nor their mental states are modeled within the logic"[5].

[5] The use of ConGolog to model multi-agent systems is discussed in [31]. An extension of Golog that support temporal reasoning has also been developed (see [47, 52]).

3.4.3 Other Languages

Situated-Automata

ConGolog is similar to *situated-automata* [54] in the sense that both languages specify agents in a logical framework. However, one major difference is that programs in situated-automata are pre-compiled down to a digital circuit for immediate execution [28, 29] while programs in ConGolog are interpreted at runtime.

Baral *et al.*'s Language for Reactive Robot Control

The last piece of work we want to discuss here is Baral *et al.*'s work [5] on reactive robot control. Their framework is similar to ours in the sense that both languages are based on logical theories of action. Baral *et al.*'s programs are conditional plans, and each condition acts like an interrupt. However, their interrupts are not prioritized. In addition, as mentioned by Baral himself in [5], their control programs are unable to handle non-deterministic actions.

3.5 Summary: On ConGolog

In brief, there are many advantages to using a knowledge-based language like ConGolog to program a robot control module. Some of these are:

- ConGolog is a language based on a solid theoretical foundation so that the task of modeling and designing agents can be done correctly.

- It supports abstraction over the details of sensing and robot operation, which makes not only coding, debugging, and verification of the robot control program easier, but also facilitates the extension and modification of the robot's behavior.

- It is capable of filling in the details of when and how to achieve some goals as long as the application domain is properly axiomatized. Without paying much attention to the low-level details, the programmer can concentrate on developing a robust high-level controller for the agent.

- With the additional features of concurrent execution of threads, prioritized interrupts, and exogenous actions, ConGolog is able to handle problems autonomous agents usually encounter in dynamic domains reactively.

Therefore, ConGolog is a good candidate as a knowledge-based programming language for controlling agents in dynamic domains at a high level.

Chapter 4

Runtime Failure and Exogenous Event Handling

In the development of a high-level control module for robots or other autonomous agents in dynamic domains, it is unrealistic to assume that no exogenous events occur at runtime or that the robot never fails in all its courses of action. Furthermore, it is also unwise to delegate all failure-handling functions to the low-level reactive module. One needs a high-level reactive control module that can handle failures and significant exogenous events at a high level. To see this, let us take the following scenario as an example. Consider a robot going to a location X and which, on the way to X, encounters some obstacle which it cannot get around. We have a runtime failure at this point. Now, a reasonable solution is to try to find another path to X, or give up on going to X and start working on another task. This can only be accomplished by a high-level control module.

In this chapter, we investigate how high-level control modules that handle runtime failures and significant exogenous events can be designed using (software) prioritized interrupts which give better reactivity and resource allocation. In our case, the high-level control modules consists of a knowledge base and a control program. The knowledge base contains all action precondition axioms, successor state axioms, initial state axioms, etc. The control program is written in CONGOLOG using concurrency and prioritized interrupts. In addition, software interrupts help to keep the maintenance costs of the control program low. A high-level control module for a mail delivery robot is developed to test our approach. The full listing of the control module can be found in appendix B.

4.1 Modeling Non-Instantaneous Actions

Before we start our discussion of runtime failure and exogenous event handling, let us take a look at how actions that have a long duration and must be interleaved with other actions can be handled in the Situation Calculus. In order to write our high-level control module in CONGOLOG, we need to find a way to model such actions using instantaneous actions. This is necessary because actions in the Situation Calculus are assumed to be instantaneous and many actions in daily life take a long time to perform.

Notice that any non-instantaneous action is marked by two points in time. One is the time when the action starts and the other is the time when the action finishes. Both of these are points, not intervals. Therefore, our approach to handle actions with a long duration is to break the action up into two instantaneous actions. One corresponds to the beginning of the action and the other corresponds to the end of the action. The latter action may be treated as an exogenous event if the environment determines when the long duration action finishes. There is an example of this in the mail delivery program that we will discuss later in this chapter: the long duration action *goTo(Place)*. The beginning of the action is modeled by the action *startGoTo(Place)* and the end of the action is modeled by the exogenous event *msgFromRobot*. The reader is referred to chapter 6 of [50] for a detailed treatment of non-instantaneous actions and concurrency.

4.2 Why Prioritized Interrupts?

As mentioned in [45], it is necessary for a robot to monitor the environment on a continuous basis in order to handle runtime failures and exogenous events reactively and elegantly.

There are two ways to achieve this. The first way is to delegate all sensing tasks to the low-level reactive module. The high-level control program, which is typically written in an ordinary non-concurrent programming language, monitors the low-level reactive module to see whether failures or exogenous events occur. In this case, the high-level control program is usually structured as a reactive and hierarchical plan, which does busy waiting on signals sent from the low-level reactive module. Therefore, for most of the time, the whole control program is actually doing nothing but checking whether some conditions are met. This results in misallocation of scarce resources.

The second way is to write the high-level control program in a language that support concurrency and priority interrupts. The language interpreter monitors signals from the low-level reactive module and selects the appropriate thread to execute automatically. Therefore, the part of the high-level control program that is not interrupted can concentrate on other things. Although this method involves overhead in time and memory used for context switching, resources are used more efficiently than in the previous method. In addition, this framework hides all resource allocation management from the high-level control program. We will see in the following sections that this kind of abstraction is best fit for robotic applications.

If interrupts are implemented at the hardware level, then the arrangement of various threads/modules will be inflexible and the "hardwired program" becomes very task-specific. Therefore, in view of the flexibility in programming and the generality of the resulting program, software interrupts are preferred. In addition, software interrupts also help lower the maintenance costs of the control program.

Finally, reactivity and better resource allocation can be achieved by prioritizing interrupts. Higher priority interrupts that are enabled execute before lower priority ones. The priority of an interrupt should reflect the amount of resources it is allowed to utilize and how urgent the associated task is.

4.3 The High-Level Model Of the Robot

For a robot to accomplish the mail delivery task, it should not only be able to navigate through the environment, but also be able to carry out sub-tasks such as acknowledging a new shipment order, and picking up and dropping off a shipment, etc. In addition, it must be able to recover from runtime failures and handle significant exogenous events.

Let us assume that all actions in our domain are instantaneous except for *goTo(Place)*. For simplicity, we treat actions like *pickup* and *dropoff* as instantaneous. The high-level controller initiates them and waits for their completion.

With this assumption, we can illustrate how the long duration action *goTo(Place)* is handled in CONGOLOG on one hand, and on the other, we can show how runtime failures and exogenous events can be treated.

In order to support the interaction between various sub-tasks, we introduce a simple model of the robot in our high-level control module. The model is defined as follows. The robot is always thought to be in one of the states in RS_f:

$$RS_f = \{\, idle,\ moving,\ blocked \,\}$$

Figure 4.1 summarizes the described model. When the robot is in the *idle* state, it is assumed to be stationary. All pickup and dropoff sub-tasks can be carried out in this state. The robot jumps into the *moving* state once the primitive action *startGoTo* is executed. If the robot reaches the destination, it will switch back to *idle* upon sending back the message *msgFromRobot(reached)*. However, if the robot is blocked on the way to the destination, it will switch to the state *blocked* instead and the message *msgFromRobot(blocked)* is sent. In order to bring the robot back to *idle* state after the control module has found a way to recover from the failure, the primitive action *resetRobot* is executed. The primitive action *resetRobot* can also be used to abort the current navigation.

The state of the robot is maintained by the successor state axiom for the fluent *robotStatus*, which is as follows:

$$\begin{array}{l}
robotStatus(do(a,s)) = i \ \equiv \\
\quad [\,(\exists p)\ a = startGoTo(p) \ \wedge\ i = moving\,] \vee \\
\quad [\,(a = resetRobot \ \vee\ a = msgFromRobot(reached)) \ \wedge\ i = idle\,] \vee \\
\quad [\,a = msgFromRobot(blocked) \ \wedge\ i = blocked\,] \vee \\
\quad [\,(\forall p)\ a \neq startGoTo(p) \ \wedge\ a \neq resetRobot \wedge \\
\quad\ \ a \neq msgFromRobot(reached) \ \wedge\ a \neq msgFromRobot(blocked) \wedge \\
\quad\ \ i = robotStatus(s)\,]
\end{array}$$

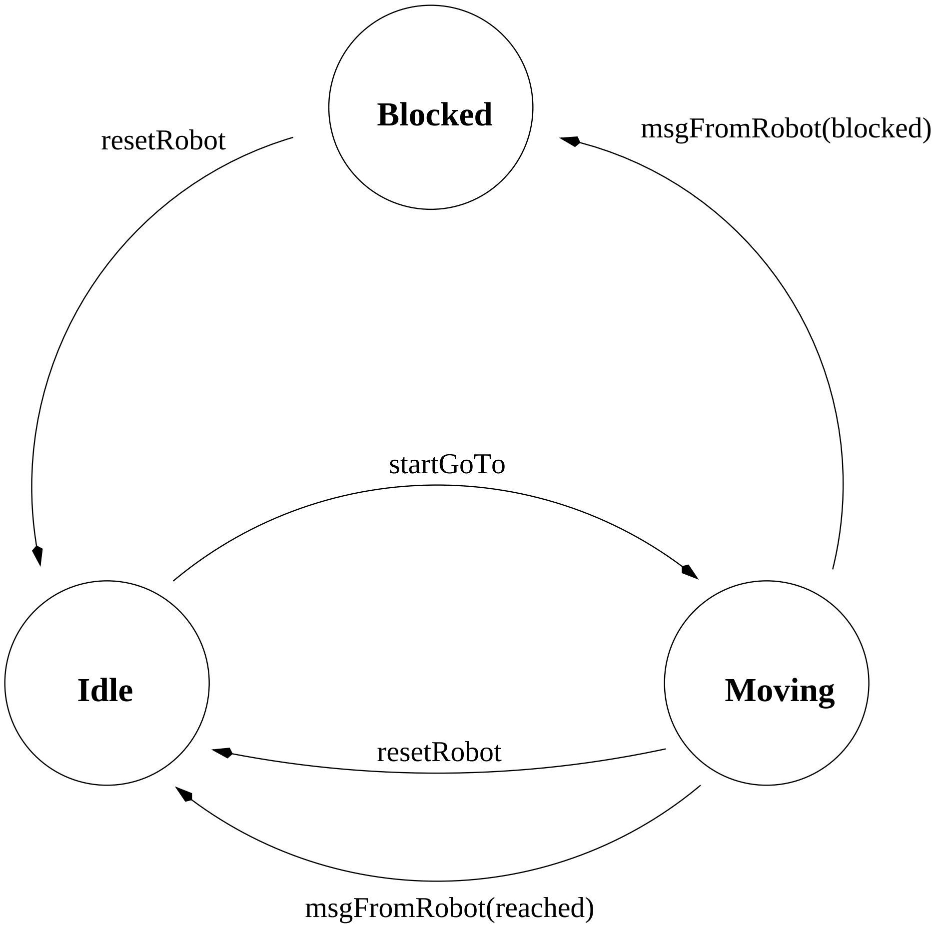


Figure 4.1: Model of our robot

In addition to the fluent *robotStatus*, we also use two functional fluents in modeling the current movement of the robot: *robotPlace* and *approaching*. *robotPlace* keeps track of the current position of the robot and *approaching* indicates the destination of the current movement. Their successor state axioms are as follows:

$$
\begin{array}{l}
robotPlace(do(a,s)) = p \;\equiv \\
\quad [\, a = msgFromRobot(reached) \;\wedge\; p = approaching(s)\,] \vee \\
\quad [\, (\exists p')\; a = startGoTo(p') \;\wedge\; p = unknown\,] \vee \\
\quad [\, p = robotPlace(s) \wedge \\
\quad\; (\forall p)\; a \neq startGoTo(p) \;\wedge\; a \neq msgFromRobot(reached)\,]
\end{array}
$$

$$
\begin{array}{l}
approaching(do(a,s)) = p \;\equiv \\
\quad a = startGoTo(p) \vee \\
\quad [\, p = approaching(s) \;\wedge\; (\forall p)\; a \neq startGoTo(p)\,]
\end{array}
$$

Note that one can distinguish between the robot successfully reaching a destination and motion being interrupted by a *resetRobot* according to the value of the *robotPlace* fluent, the destination in the former case and *unknown* in the latter case.

4.4 An Overview of the Mail Delivery Domain

A reactive mail delivery agent has been developed to test our approach to handling exogenous events and runtime failures. In this example domain, the robot lives in an office setting and its task is to pick up and drop off mail. There are two types of significant exogenous events and two types of runtime failures. The significant exogenous events our robot is expected to handle are:

- the generation of new shipment orders, which is represented by the event *orderShipment*; and

- the signals sent from the navigation module, namely the events *msgFrom-Robot(reached)* and *msgFromRobot(blocked)*.

New shipment orders with two different priority levels may come in at any time. Our agent should acknowledge each new order with an appropriate message. We use *ackShipmentOrder* for valid shipments and *ackWithRegret* for invalid shipments (a definition will be given later in this section). In addition, acknowledgment should be done in a timely manner.

Our robot is also expected to serve high-priority shipments prior to serving low-priority ones. Furthermore, messages from the navigation module may come at arbitrary times. Let us take the messages *msgFromRobot(reached)* and *msgFrom-Robot(blocked)* as an example. We are not sure when these messages will come after the robot has started moving from one place to another. This is because we do not know exactly how long will it take the robot to come to a stop, either at the destination or somewhere else if it gets blocked by an obstacle. When dealing with these events introduced by the navigation module, the high-level control module should be able to respond promptly with an appropriate action such as finding a new client to serve.

The runtime failures our robot should be able to recover from are:

- when the robot gets blocked on the way to a customer's mailbox, and
- when the recipient and/or sender of a shipment gets suspended (because it is considered unreachable — the criteria will be given in the next paragraph) before the shipment has been completely served.

When a robot is operating in a real office, it is possible for it to encounter obstacle that block its way to a customer's mailbox. In order to recover from such runtime failures, our control module should be able to detect such failures and find a new best client to serve next.

Since the obstacle encountered by the robot may be temporary, we want the robot to make several attempts to reach a customer's mailbox before giving up. This is accomplished by assigning every person in our office k credits initially, and reducing the customer's credit by one when an attempt to go to their mailbox fails. If it happens to be the case that it is the k^{th} time that the robot gets blocked on the way to a client's mailbox, the client's credit will reach zero and he/she will be notified that he/she will not be served by the robot anymore. In addition, from now on, the client will be treated as a "suspended client". As a consequence, some shipments may become undeliverable because their recipient has been suspended and/or unreturnable because their sender has been suspended. Let us call these shipments "invalid shipements" for short. Our robot is expected to handle these invalid shipments as well. It recovers from this type of failure as follows. For invalid shipments that have not been picked up yet, the pickup action will be cancelled. For invalid shipments already onboard, these will either be returned to their sender if the sender has not run out of credit or be delivered to the *centralOffice* for future collection if both the sender and recipient have been suspended. Furthermore, all subsequent new shipment orders made by or sent to a suspended client will be acknowledged with the action *ackWithRegret*, which tells the sender that the shipment order just placed will not be served.

4.5 Primitive Actions and Fluents

In order to carry out the described mail delivery task, the robot must be able to navigate through the world, acknowledge new shipment orders, and pick up and drop off shipments. All the primitive and exogenous actions used in our example are listed in Table 4.1.

Primitive Actions	Descriptions
Robot Actions :	
$startGoTo(l)$	Starts moving to place l.
$ackShipmentOrder(n, c, r)$	Acknowledges the reception of a valid shipment order n from client c to client r.
$pickUpShipment(n)$	Picks up shipment n.
$dropOffShipment(n)$	Drops off shipment n.
$shutdownRobot$	Shuts down the robot.
$cancelPickUp(n, c)$	Cancels the plan to pick up shipment n from client c.
$reduceCredit(c)$	Reduces client c's credit by one.
$resetRobot$	Halts and resets the robot.
$notifyStopServing(c)$	Notifies client c that he/she will not be served anymore.
$ackWithRegret(n, c, r)$	Acknowledges the reception of an invalid shipment order n from client c for client r.
Exogenous Actions :	
$orderShipment(c, r, p)$	Places a new shipment order from client c to client r with priority p.
$msgFromRobot(i)$	A message is received from the navigation module. i is either *reached* or *blocked*

Table 4.1: List of Primitive and Exogenous Actions

We have an action precondition axiom for each action. For example:

$$\begin{array}{l} Poss(pickUpShipment(n), s) \;\equiv\; (\exists\, c)\,.shipmentToPickUp(n, s) \wedge \\ \qquad sender(n, s) = c \;\wedge\; robotPlace(s) = mailBox(c) \end{array}$$

$$\begin{array}{l} Poss(dropOffShipment(n), s) \;\equiv\; (\exists\, c)\,.shipmentOnBoard(n, s) \wedge \\ \qquad (\,[\,(\,sender(n, s) = c \;\vee\; recipient(n, s) = c\,) \wedge \\ \qquad\quad robotPlace(s) = mailBox(c)\,] \;\vee\; robotPlace(s) = centralOffice\,) \end{array}$$

The complete set of action precondition axioms can be found in appendix C.1.

The state of the world is modeled using the primitive fluents listed in Table 4.2. We have a successor state axiom for each primitive fluent. For example:

$$\begin{array}{l} shipmentOrdToAck(n, do(a, s)) \;\equiv \\ \qquad (\exists\, c, r, p)\,[\,a = orderShipment(c, r, p) \;\wedge\; shipmentCtr(s) = n\,] \vee \\ \qquad [\,shipmentOrdToAck(n, s) \;\wedge\; (\forall\, c, r)(a \neq ackShipmentOrder(n, c, r) \wedge \\ \qquad\; a \neq ackWithRegret(n, c, r))\,] \end{array}$$

$$\begin{array}{l} priority(n, do(a, s)) = p \;\equiv \\ \qquad [\,(\exists\, c, r)\; a = orderShipment(c, r, p) \;\wedge\; shipmentCtr(s) = n\,] \vee \\ \qquad p = priority(n, s) \end{array}$$

The rest of successor state axioms can be found in appendix C.2.

For the sake of clarity, we introduce some defined fluents which are fluents defined in terms of the primitive fluents and/or other defined fluents. They are listed in Table 4.3. For example, we use the following set of defined fluents to specify the best customer(s) to serve:

$$\begin{array}{l} bestCustToServe(c) \stackrel{\text{def}}{=} \\ \qquad needServing(c) \wedge \\ \qquad (\exists\, p)\,.\,haveShptOfPriority(c) = p \wedge \\ \qquad\qquad isTheHighestPriority(p) \;\wedge\; haveHighestCredit(c, p) \end{array}$$

Primitive Fluents	Descriptions
Predicate Fluents:	
$shipmentOrdToAck(n)$	Shipment order n is waiting for acknowledgment.
$shipmentToPickUp(n)$	Shipment n is waiting to be picked up.
$shipmentOnBoard(n)$	Shipment n is being held by the robot.
$shipmentDelivered(n)$	Shipment n has been delivered to an appropriate location and has been served.
$clientSuspended(c)$	Client c has been suspended.
$readyToBeServed$	At least one shipment order has been made and is ready to be served.
$robotAlive$	Robot has not been shutdown.
Functional Fluents:	
$sender(n)$	The sender of shipment n.
$recipient(n)$	The recipient of shipment n.
$priority(n)$	The priority level of shipment n.
$shipmentCtr$	The shipment number of the next shipment order.
$robotStatus$	The robot's state; at any time, the robot can only be in one of the following states: *moving* , *idle* and *blocked*.
$robotPlace$	The current position of the robot; *unknown* is returned if the robot is moving or being blocked somewhere in the world.
$haveCredit(c)$	Client c's current amount of credit.
$approaching$	The destination of the current movement.

Table 4.2: List of Primitive Fluents

$$
\begin{aligned}
& needServing(c) \stackrel{\text{def}}{=} \\
& \quad (\exists\, n)\,.\,\{\, shipmentToPickUp(n) \;\wedge\; sender(n) = c \;\wedge \\
& \qquad\qquad shipmentDeliverable(n) \;\wedge\; shipmentReturnable(n)\,\} \;\vee \\
& \qquad\quad \{\, shipmentOnBoard(n) \;\wedge\; \neg\, clientSuspended(c) \;\wedge \\
& \qquad\qquad [\, sender(n) = c \;\wedge\; \neg\, shipmentDeliverable(n)\,] \;\vee \\
& \qquad\qquad recipient(n) = c\,\}
\end{aligned}
$$

$$
\begin{aligned}
& haveShptOfPriority(c) = p \stackrel{\text{def}}{=} \\
& \quad \{\, (\exists\, n)\,[\,(\, shipmentToPickUp(n) \;\wedge\; sender(n) = c \;\vee \\
& \qquad\qquad shipmentOnBoard(n) \;\wedge\; recipient(n) = c\,) \;\wedge \\
& \qquad\quad priority(n) = 1\,] \;\supset\; p = 1\,\} \;\wedge \\
& \quad \{\neg(\exists\, n)\,[\,(\, shipmentToPickUp(n) \;\wedge\; sender(n) = c \;\vee \\
& \qquad\qquad shipmentOnBoard(n) \;\wedge\; recipient(n) = c\,) \;\wedge \\
& \qquad\quad priority(n) = 1\,] \;\supset\; p = 0\,\}
\end{aligned}
$$

$$
\begin{aligned}
& isTheHighestPriority(p) \stackrel{\text{def}}{=} \\
& \quad (\exists\, c)\,[\, needServing(c) \;\wedge\; haveShptOfPriority(c) = p\,] \;\wedge \\
& \quad (\forall\, c', p')\,[\, needServing(c') \;\wedge\; haveShptOfPriority(c') = p' \;\supset\; p \geq p'\,]
\end{aligned}
$$

$$
\begin{aligned}
& haveHighestCredit(c, p) \stackrel{\text{def}}{=} \\
& \quad (\forall\, c', m, m')\,.\, needServing(c') \;\wedge\; haveShptOfPriority(c') = p \;\wedge \\
& \qquad\qquad c \neq c' \;\wedge\; haveCredit(c') = m' \;\wedge\; haveCredit(c) = m \\
& \qquad\qquad \supset\; m \geq m'
\end{aligned}
$$

This set of defined fluents essentially says that the best customer(s) to serve is/are the one(s) with the highest priority shipment(s) and the largest amount of credit available among those who have shipment(s) of the same priority level. This provides a declarative specification of the robot's scheduling of order service. The definitions for the rest of the defined fluents appears in appendix C.3.

Defined Fluents	Descriptions
Defined Fluents:	
$needServing(c)$	Holds if client c needs to be served, that is, there are some valid shipments in client c's mailbox waiting for pickup and/or some shipments onboard waiting for dropoff to c.
$haveShptOfPriority(c)$	Returns the highest priority level among all shipments sent from/to client c.
$haveHighestCredit(c, p)$	Holds if client c has the largest amount of credit among all clients who have shipment(s) of priority p.
$isTheHighestPriority(p)$	Holds if p is the highest priority among all shipments to be served.
$bestCustToServe(c)$	Holds if c is one of the best customers to serve; this is defined to be one who needs to be served, has the highest priority shipment(s), and has the largest amount of credits available among those who have shipment(s) of the same priority level.
$servingPriorityShpt$	Holds if the robot is serving some high priority shipments.
$shipmentReturnable(n)$	Holds if the sender of shipment n has not been suspended.
$shipmentDeliverable(n)$	Holds if the recipient of shipment n has not been suspended.

Table 4.3: List of Defined Fluents

4.6 Main Program and Procedures

As mentioned, we will handle runtime failures and exogenous events using prioritized interrupts. If we take a look at our task (mail delivery) domain carefully, we can see that there are three major activities that should execute concurrently: acknowledging new shipment order, cancelling pickup for invalid shipments and the normal delivery routine. In addition to these, we need another thread to prevent the program from aborting when there is nothing to do and the program must wait. Therefore, in the main control program, four threads are forked and executed concurrently at different priorities. Each thread involves an interrupt. An interrupt will be triggered when a given condition on the current world state is met. At least one thread must be running at all times; otherwise, the program will terminate. The four threads are as follows:

$$
\begin{array}{l}
\langle\, n : shipmentOrdToAck(n) \rightarrow acknowledgeShipmentOrder(n)\,\rangle \\
\gg \\
\langle\, n, c : (shipmentToPickUp(n) \wedge sender(n) = c \wedge \\
\qquad \neg(shipmentReturnable(n) \wedge shipmentDeliverable(n))) \\
\qquad\qquad \rightarrow cancelPickUp(n, c)\,\rangle \\
\| \\
\langle\, n : [\,(shipmentToPickUp(n) \vee shipmentOnBoard(n) \vee \\
\qquad robotPlace \neq centralOffice) \wedge readyToBeServed\,] \\
\qquad\qquad \rightarrow robotMotionControl\,\rangle \\
\gg \\
\langle\, (\,\neg readyToBeServed \vee robotAlive\,) \rightarrow \mathbf{noOp}\ \,\rangle
\end{array}
$$

In order to make our agent respond to new shipment orders reactively, a thread that acknowledges new shipment orders is forked at the highest priority level. The associated interrupt will be triggered whenever there is a new shipment order. The procedure executed is:

proc $acknowledgeShipmentOrder(n)$
$(\pi\, c, r)\, [\, (sender(n) = c\ \wedge\ recipient(n) = r)?;$
if $(shipmentReturnable(n)\ \wedge\ shipmentDeliverable(n))$ **then**
$ackShipmentOrder(n, c, r);$
if $(priority(n) = 1\ \wedge\ robotStatus = moving\ \wedge$
$\neg\, servingPriorityShpt)$ **then**
$resetRobot$ % abort current service
else noOp
else $ackWithRegret(n, c, r)\,]$
end

This procedure gets the robot to acknowledge new valid shipment orders with *ackShipmentOrder* and new invalid shipment orders with *ackWithRegret*. In addition, if the new shipment order is of high priority and the robot is on the way to serve some low-priority shipment, then this thread will issue a *resetRobot* command to stop the robot. This has the effect of terminating the current execution of *robotMotionControl* in the third thread. In the next round, the *robotMotionControl* thread will start executing all over again with respect to the new world state. This will result in the robot having the new goal of serving the customer with the high-priority shipment.

At the second level of priority, we have a thread responsible solely for cancelling the pickup action of invalid shipments, i.e. shipments of which either the sender and/or the recipient has been suspended. This thread will be triggered when there are invalid shipments waiting for pickup.

At the same level, we have another thread that is responsible for sub-tasks like navigating around the office, and picking up and dropping off shipments. The interrupt is triggered as long as there are shipments waiting for pickup or dropoff, or the robot is not located at the *centralOffice*, and at least one order has been made (which makes the fluent *readyToBeServed* true — we do not want the robot to go retire at the *centralOffice* unless it has served an order). This thread is made up of six procedures, namely *robotMotionControl, serveCustomer, handleGoToFailure, dropOffToCentralOffice, pickUpShipmentsFrom*, and *dropOffShipmentsTo*.

robotMotionControl is the top level control procedure for the robot's motion; it decides whether to go serve a person or to go to the *centralOffice*:

```
proc robotMotionControl
  if (∃ c) (needServing(c)) then
    serveCustomer
  else
    startGoTo(centralOffice);
    (robotStatus ≠ moving)?;
    if (robotPlace = centralOffice) then
       [ dropOffToCentralOffice;
        if (∃ c′) (needServing(c′)) then
          noOp
        else shutdownRobot ]
    else if (robotStatus = blocked) then
           resetRobot
         else noOp      % aborted service
end
```

In plain language, this means that if there are customers that need to be served, then the robot goes and serves them; otherwise, it starts going to the *centralOffice*. In the latter case, the controller then waits until the robot halts. If the robot halts at the *centralOffice*, then it drops off all invalid shipments that are onboard. If the robot halts and the robot status returned from the navigation module is *blocked*, then it abandons the current attempt to go to the *centralOffice* and resets the robot. Note that the procedure also terminates when the robot's movement is aborted by an order coming in (when *resetRobot* is executed in *acknowledgeShipmentOrder*). In addition, the robot shuts down only if there are no valid shipments.

serveCustomer is a procedure which is called to serve a customer. It is defined as follows:

proc $serveCustomer$
  $(\pi\, c)\, [\, bestCustToServe(c)?;$
    $startGoTo(mailBox(c))$
    $(robotStatus \neq moving)?;$
    if $(robotPlace = mailBox(c))$ **then**
      $[\, pickUpShipmentsFrom(c); dropOffShipmentsTo(c)\,]$
    else if $(robotStatus = blocked)$ **then**
        $handleGoToFailure(c)$
      else noOp]   % aborted service
end

This says that the robot first picks one of the best customers to serve. Then it starts going to his/her mailbox and waits until the robot halts. If the robot halts at the customer's mailbox, then it picks up and drops off the shipment(s) from/to the customer. However, if the robot halts and the returned robot status is *blocked*, then *handleGoToFailure* is executed. The procedure also terminates when the robot's movement is interrupted by the execution of the action *resetRobot* (which leaves *robotPlace* to have the value *unknown* and *robotStatus* the value *idle*).

To deal with failure to reach a customer c's mailbox, we execute:

proc $handleGoToFailure(c)$
  $reduceCredit(c);$
  if $(haveCredit(c) = 0)$ **then**
    $notifyStopServing(c);$
  else noOp;
  $resetRobot$
end

This first reduces c's credit by one and then checks whether c's remaining credit level has reached zero. If so, c will be notified with a message saying that service to him/her has been suspended. This is represented by the primitive action *notifyStopServing*. Otherwise, the robot is reset. This makes it possible to start serving a new customer at the next round.

dropOffToCentralOffice is the procedure that defines the robot's behavior when there are no shipments to pick up or deliver, and the robot has returned to the *centralOffice*. It proceeds to drop off all the onboard invalid shipments.

$$
\begin{array}{l}
\textbf{proc } dropOffToCentralOffice \\
\quad \textbf{while } (\exists\, n)\, (shipmentOnBoard(n) \wedge \\
\qquad\qquad\qquad \neg\, (shipmentDeliverable(n) \;\vee\; shipmentReturnable(n)) \\
\qquad (\pi\, n)\, [\, (shipmentOnBoard(n) \wedge \\
\qquad\qquad\qquad \neg\, (shipmentDeliverable(n) \;\vee\; shipmentReturnable(n))?; \\
\qquad\qquad dropOffShipment(n)\,] \\
\textbf{end}
\end{array}
$$

The procedures *pickUpShipmentsFrom* and *dropOffShipmentsTo* define the robot's behaviors at a client's mailbox. The procedure *pickUpShipmentsFrom* simply picks up shipments from customer c's mailbox as long as there are any:

$$
\begin{array}{l}
\textbf{proc } pickUpShipmentsFrom(c) \\
\quad \textbf{while } (\exists n)\, (shipmentToPickUp(n) \wedge sender(n) = c) \\
\qquad (\pi\, n)\, [(shipmentToPickUp(n) \wedge sender(n) = c)?; \\
\qquad\qquad pickUpShipment(n) \\
\textbf{end}
\end{array}
$$

Similarly, the procedure *dropOffShipmentsTo* drops off all onboard shipments, of which client c is either the sender (in which case the shipments are undeliverable) or recipient, to c's mailbox as long as there are any:

$$
\begin{array}{l}
\textbf{proc } dropOffShipmentsTo(c) \\
\quad \textbf{while } (\exists n)\,(shipmentOnBoard(n) \;\wedge\; (recipient(n) = c \,\vee \\
\qquad\qquad\qquad (sender(n) = c \;\wedge\; \neg shipmentDeliverable))) \\
\qquad (\pi\, n)\,[\,(shipmentOnBaord(n) \;\wedge\; (recipient(n) = c \,\vee \\
\qquad\qquad\qquad\qquad\qquad (sender(n) = c \;\wedge\; \neg shipmentDeliverable)))?; \\
\qquad\qquad dropOffShipment(n)\,] \\
\textbf{end}
\end{array}
$$

Finally, the last thread in the main program is an interrupt that prevents the program from halting prematurely, i.e. halting before the first shipment order has come or before all shipments have been delivered. After the program has returned to the *centralOffice* and dropped off all remaining shipments, it executes *shutdownRobot* which makes *robotAlive* false. This in turn makes the triggering condition of this thread false, upon which the whole program terminates.

4.7 Sample Runs

Let us take a look at some sample runs obtained in simulation mode. The scenario we used in these sample runs involves four persons: Yves, Michael, Jeff and Hector. We use the following initial state axioms:

$$
\begin{gathered}
robotPlace(S_0) = robotHome \\
approaching(S_0) = nowhere \\
robotStatus(S_0) = idle \\
robotAlive(S_0) \\
shipmentCtr(S_0) = 1 \\
(\forall\, c)\,.\,person(c) \supset haveCredit(c, S_0) = 2
\end{gathered}
$$

These axioms say the following. The robot is initially at *robotHome* and going to *nowhere*. The *robotStatus* is initially *idle* and the first shipment will be given “1” as its shipment number. Lastly, every person has a credit of two to start with. Notice that no shipment orders have yet been placed in the initial state.

In our first sample run, the probability of being blocked and arriving at the destination are 0.4 each. Here is the output:

```
| ?- run2.
orderShipment(jeff,michael,0)       % shipment 1
ackShipmentOrder(1,jeff,michael)
startGoTo(mailBox(jeff))
msgFromRobot(reached)
pickUpShipment(1)
orderShipment(yves,jeff,1)          % shipment 2
ackShipmentOrder(2,yves,jeff)
orderShipment(michael,hector,0)     % shipment 3
orderShipment(yves,jeff,1)          % shipment 4
ackShipmentOrder(3,michael,hector)
ackShipmentOrder(4,yves,jeff)
startGoTo(mailBox(yves))
msgFromRobot(blocked)
orderShipment(michael,hector,0)     % shipment 5
ackShipmentOrder(5,michael,hector)
reduceCredit(yves)
resetRobot
startGoTo(mailBox(yves))
msgFromRobot(blocked)
reduceCredit(yves)
notifyStopServing(yves)
cancelPickUp(4,yves)
cancelPickUp(2,yves)
resetRobot
startGoTo(mailBox(michael))
msgFromRobot(reached)
pickUpShipment(5)
pickUpShipment(3)
dropOffShipment(1)
startGoTo(mailBox(hector))
msgFromRobot(blocked)
reduceCredit(hector)
resetRobot
startGoTo(mailBox(hector))
```

```
36: msgFromRobot(reached)
37: dropOffShipment(3)
38: dropOffShipment(5)
39: startGoTo(centralOffice)
40: orderShipment(michael,hector,0)     % shipment 6
41: msgFromRobot(reached)
42: ackShipmentOrder(6,michael,hector)
43: orderShipment(michael,hector,0)     % shipment 7
44: ackShipmentOrder(7,michael,hector)
45: startGoTo(mailBox(michael))
46: msgFromRobot(reached)
47: orderShipment(yves,jeff,1)          % shipment 8
48: ackWithRegret(8,yves,jeff)
49: pickUpShipment(7)
50: pickUpShipment(6)
51: startGoTo(mailBox(hector))
52: msgFromRobot(blocked)
53: reduceCredit(hector)
54: notifyStopServing(hector)
55: resetRobot
56: startGoTo(mailBox(michael))
57: msgFromRobot(reached)
58: dropOffShipment(6)
59: dropOffShipment(7)
60: startGoTo(centralOffice)
61: msgFromRobot(blocked)
62: resetRobot
63: startGoTo(centralOffice)
64: msgFromRobot(blocked)
65: resetRobot
66: startGoTo(centralOffice)
67: msgFromRobot(reached)
68: shutdownRobot
69:
70: yes
```

In this test, notice that acknowledgments take place immediately after every new shipment order. Furthermore, the robot always serves the highest priority shipment first even when it is not the earliest order. For example, shipment 2 and 4 are served prior to shipment 3 (line 13 in the trace). Note also that in the experiment in chapter 7, unlike in this test, the case where the robot's motion is interrupted by a high-priority shipment coming in does arise and the control program performs correctly. This shows that our approach is able to handle the exogenous events in this domain properly.

Furthermore, the robot behaves as expected in cases in which it gets blocked on the way to somebody's mailbox. The credit of the customer is reduced and he/she is suspended if he/she runs out of credit; for example, Yves and Hector are suspended in our example. Moreover, the action *cancelPickUp* is correctly carried out. In our example, the pickup for shipment 2 and 4 are cancelled after Yves has been suspended.

In addition, this test shows that even if a shipment order is placed at the very last minute before the robot is shut down, the robot will still serve that shipment. For example, shipment 6 and 7 are served even though they are ordered after the robot has reached the *centralOffice* and is about to retire.

We also obtained another sample run which shows that the robot can handle shipments for which both the sender and recipient are suspended after they have been picked up. The run appears in appendix D. To get this, the chance of being blocked is deliberately set to 0.6 while that of arriving at the destination is 0.2. In this sample run, shipments 2 and 4 are this kind of shipment. As expected, they are dropped off to the *centralOffice* at the end of our test run.

4.8 Summary: On Runtime Failures and Exogenous Events Handling

By dividing the mail delivery task into sub-tasks and having each sub-task handled by a separate thread, concurrent execution of the sub-tasks is possible. Using software interrupts, the robot is able to react to failures and exogenous events and do some productive things such as acknowledging a new shipment order and cancelling pickups while it is moving. Despite of the overhead of context switching, resources can be utilized more efficiently. In addition, software interrupts also help to lower the costs involved in the development of a robust robot control module. Lastly, reactivity is achieved by prioritizing interrupts because critical conditions in the environment can now be monitored more closely than the less critical one. This in turn results in prompt response to runtime failures and significant exogenous events.

Chapter 5

Knowledge-Based Search For Objects

Many robotics tasks require the robot to find an object or a person. Searching for an object with incomplete information about the task domain can be very difficult. To see this, let us look at the various types of object search tasks. First, suppose that the robot is looking for an immobile object and the location of the object is known. This is simple. The robot can use a path-finding algorithm to find a path to that location and then use its onboard sensors to pinpoint the object there.

Next, suppose that the robot is asked to find an immobile object whose location is not known. Here, the robot has incomplete information about the domain. In this case, there is a simple algorithm that is guaranteed to work: search every possible alternative location. However, this may be very inefficient because the search space may be large and it can take a long time for the robot to just go to one location and examine it. Acquiring information about the object as the robot searches will shrink the search space, which in turn gives a more time-efficient search.

Finally, suppose that the robot has to find a mobile object in its environment. Now, the robot has incomplete information about the domain and the mobility of the object keeps the information from ever becoming complete. For example, suppose that an object Q is at location X, that the robot is at location Y, and that it is known that Q is now at X. Without any knowledge about where Q will go, the robot may try to catch Q at X. However, Q may have moved to another location Z by the time the robot reaches X. Clearly, the robot might never catch Q. So, there is no algorithm that is guaranteed to find a mobile object in a domain. The robot can never have complete information about such a domain. However, there are ways to improve chances of finding the object. One of them is to acquire information at runtime and use it to guide search.

In this chapter, we will try to cope with the problems posed by having incomplete information about objects' locations in a domain by using a knowledge-based approach to search. As pointed out in the above, acquiring information at runtime and making use of one's knowledge to select locations to search might help in achieving a more time-efficient search and improving the chance of finding the object. So in this chapter, the high-level control module developed in chapter 4 will be extended to incorporate a knowledge-based approach to search.

It should be pointed out that the work in this chapter is somewhat preliminary. The technical problems involved in dealing with sensing and knowledge base update in an agent programming language are complex. We do not try to develop a general approach to these problems[1] Instead we examine how a control module for an application involving these problems can be successfully developed.

[1]It is difficult to even come up with a formal specification of the problem: what the robot knows might be viewed as a probability assignment to the nodes of a graph, where these probabilities have to be revised when new information is acquired as well as updated when the object moves.

Obtaining general solutions is left for future work.

5.1 Why a Knowledge-Based Approach?

The simplest approach to search is to look into every possible alternative location. This is inefficient because the search space may be so large that it takes the agent a very long time to find the object. Furthermore, this approach often fail when the search task involves mobile objects. This is because the mobility of objects keeps the search space and the robot's knowledge about the task domain from ever shrinking and complete, respectively. So the objects may never be found and the search may never terminate.

Although researchers such as Pohl [48], Nilsson [44], Barr and Feigenbaum [6] have studied the use of heuristics to speed up search, these approaches are only suitable for search in a model where information is assumed to be complete; they are inappropriate for search with incomplete information.

As observed above, the mobility of objects and incomplete information about the domain are two main reasons why the exhaustive and the heuristic approaches to search may fail. However, if more information about the location of an object can be acquired as the robot conducts the search, it will have a better chance of successfully completing the search task and doing so in a timely manner. In this chapter, we try to do this by applying a knowledge-based approach to search for objects.

The *knowledge-based approach to search* (or simply called "*knowledge-based search*") uses a knowledge base to support reasoning and a knowledge-based robot programming language where actions can be selected according to what can be inferred from the knowledge base. The problems posed by incomplete information can be dealt with by acquiring information at runtime, updating the knowledge base, and performing reasoning on the knowledge base to select actions. For example, we can use reasoning to select the next location to search or the next person to query for additional information. Knowledge-based search is especially appropriate in domains where complete information is not available.

Since knowledge-based search depends on the amount of information in the knowledge base, the robot is forced to seek relevant information about the current location of the object it is looking for. If more information can be obtained, we have a smaller search space and a better chance of finding the object; the time required to do so can be shortened.

5.2 Our Extended Model of the Robot

In order to experiment with knowledge-based search, we will develop an application where the robot is used to search for people in an office setting for delivering mail by hand. Therefore, in addition to the capabilities presented in last chapter, our robot should now also be able to detect the presence of people and obtain information from people nearby. In our example, we assume that the robot is equipped with visual sensors for detecting people and a key-pad for collecting information from people nearby.

As a consequence, the model of our robot's lower level architecture described in section 4.3 is extended by adding two more states, namely *scanning* and *querying*.

The set of possible robot states in the extended model is as follows:

$$RS_s = \{\, idle,\ moving,\ blocked,\ scanning,\ querying \,\}$$

and Figure 5.1 gives its corresponding state diagram.

The robot puts itself in the *scanning* state upon the execution of the primitive action *scanPlace*. In this state, visual sensors and schemas are activated to find people in the surroundings. It switches back to state *idle* after the environment has been scanned and sensory data have been collected.

If the person the robot is looking for, say A, is not found but people who may know where to find A are spotted instead. The robot goes into the *querying* state when the action *askAbout* is performed to ask a person about where A is. The robot remains in the *querying* state until any of the following events happens:

- when a person knows the current location of A and tells the robot,
- when all people have been asked and no one knows where A is, and
- when the action *resetRobot* is performed.

Upon the occurrence of any of the above events, the robot will return to the state *idle*.

In all states other than the state *idle*, execution of the primitive action *resetRobot* terminates all the robot's current actions, resets the robot, and returns the robot's state to *idle* in which it will wait for a new instruction from the high-level control module.

As before, information on the movement of the robot is maintained using the fluents *robotStatus*, *robotPlace*, and *approaching*. The successor state axioms

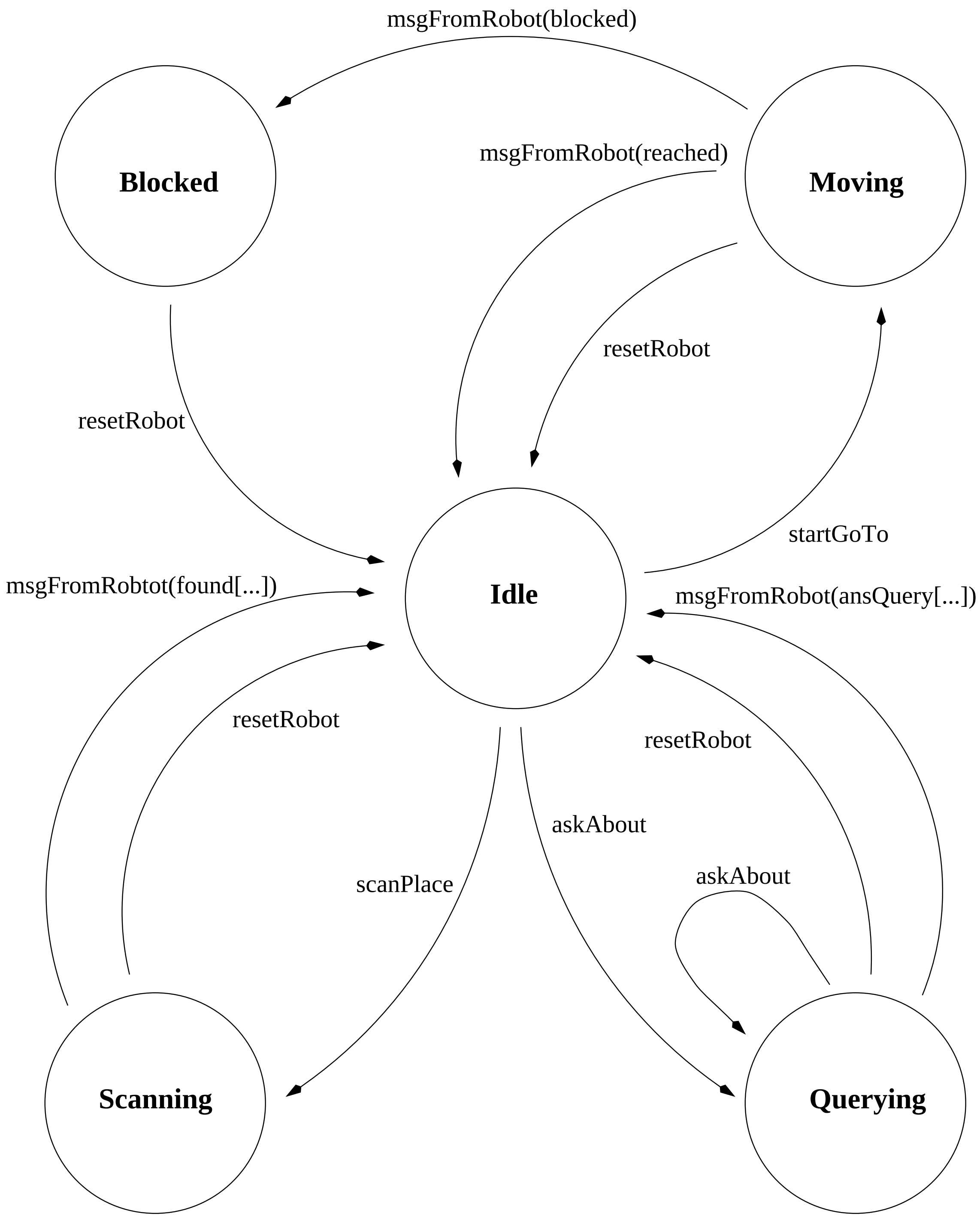


Figure 5.1: Extended model of our robot

of the latter two are as before. However, the successor state axiom of the fluent *robotStatus* is modified in order to reflect the extensions to the model. The revised successor state axiom is as follows:

$$
\begin{array}{l}
robotStatus(do(a,s)) = i \;\equiv \\
\quad [\,(\exists\, p)\; a = startGoTo(p) \;\wedge\; i = moving\,] \vee \\
\quad [\,(a = resetRobot \;\vee\; a = msgFromRobot(reached) \vee \\
\qquad (\exists\, t)\; a = msgFromRobot(found(t)) \vee \\
\qquad (\exists\, k)\; (a = msgFromRobot(ansQuery(k)) \;\wedge\; k \neq \texttt{[ ]})) \wedge \\
\qquad i = idle\,] \vee \\
\quad [\,a = msgFromRobot(blocked) \;\wedge\; i = blocked\,] \vee \\
\quad [\,a = scanPlace \;\wedge\; i = scanning\,] \vee \\
\quad [\,(\exists\, c', c)\; a = askAbout(c', c) \;\wedge\; i = querying\,] \vee \\
\quad [\,(\forall\, p)\; a \neq startGoTo(p) \;\wedge\; a \neq resetRobot \;\wedge\; a \neq scanPlace \wedge \\
\qquad (\forall\, m)\; (a \neq msgFromRobot(m) \;\vee\; m = ansQuery(\texttt{[ ]})) \wedge \\
\qquad (\forall\, c', c)\; a \neq askAbout(c', c) \;\wedge\; i = robotStatus(s)\,]
\end{array}
$$

where k is a set of places and t is a set of people.

5.3 Our Revised Domain Specification

In this chapter, our robot is operating in an office in which people can move from place to place. In addition to failure and exogenous event handling behaviors, the robot should exhibit intelligent behaviors to search for people in the office. Thus, it must be able to detect the presence of people, query people nearby about the location of the person it is searching for, and update its knowledge base when appropriate.

The knowledge base in our module contains two types of facts (or knowledge): fluents and non-fluents. Non-fluents are facts provided by the programmer when the knowledge base is written up and their meanings/contents never change during execution. In constrast, fluents are facts that can change as the agent operates and can be updated as sensing is performed. There are a lot of ways to represent knowledge about relations between people and places. For the sake of simplicity, we represent such knowledge using ordered lists. The places in an ordered list are assumed to be ordered with the first element having the highest chance of being the place where the person is and the last element having the least. Places that have probability zero of being the location of the client are simply left out of the list. The lists are treated as sets, i.e. a place can only appear once in the list.

In the revised domain specification, we have two additional non-fluents for every person c in our office. The first non-fluent represents the agent's expectations about where c may be found in the absence of other information. For this, we use the predicate *possiblePlacesFor(c,l)*, where c is a client and l is an ordered list of places. It can be viewed as a function that takes c's name as input and returns an ordered list l containing places where c may be found. The second new non-fluent is the predicate *mightKnowBy(c,t)* which represents the fact that the people in the set t are those who might know about the location of client c.

What the robot knows about the location of a client c in a situation s is represented by the functional fluent *knownPossPlacesFor(c,s)* whose value is an ordered list of places. Every query about the best place to find a person c made to the knowledge base in a situation s is answered with the first element of the ordered list *knownPossPlacesFor(c,s)*.

The successor state axiom of *knownPossPlacesFor* is as follows:

$$
\begin{array}{l}
knownPossPlacesFor(c, do(a, s))) = l \ \equiv \\
\quad [\,(\exists\, p, l')\ (\, a = delFromKnownPossPlaces(c, p) \wedge \\
\qquad\qquad knownPossPlacesFor(c, s) = l' \wedge \\
\qquad\qquad l = removeElement(p, l')\,)\,] \vee \\
\quad [\,(\exists\, t, p, l', l'')\ (\, a = restorePlaceList(c)\ \wedge\ vSensorReadings(s) = t \wedge \\
\qquad\qquad\quad c\ \in\ t\ \wedge\ possiblePlacesFor(c) = l' \wedge \\
\qquad\qquad\quad robotPlace(s) = p\ \wedge\ removeElement(p, l') = l'' \wedge \\
\qquad\qquad\quad l = queueAtFront(p, l'')\,)\,] \vee \\
\quad [\,(\exists\, t)\ (a = restorePlaceList(c) \wedge \\
\qquad\qquad \neg\ (vSensorReadings(s) = t\ \wedge\ c\ \in\ t) \wedge \\
\qquad\qquad l = possiblePlacesFor(c)\,)\,] \vee \\
\quad [\,(\exists\, k, l', l'')\ (\, a = msgFromRobot(ansQuery(k))\ \wedge\ k \neq \texttt{[ ]} \wedge \\
\qquad\qquad\quad serving(s) = c\ \wedge\ knownPossPlacesFor(c, s) = l' \wedge \\
\qquad\qquad\quad removeList(k, l') = l''\ \wedge\ l = queueAtFront(k, l'')\,)\,] \vee \\
\quad [\, a \neq restorePlaceList(c) \wedge \\
\quad\ (\forall\ p)\ a \neq delFromKnownPossPlaces(c, p) \wedge \\
\quad\ (\forall\ k)\ \neg\ (\, a = msgFromRobot(ansQuery(k))\ \wedge\ k \neq \texttt{[ ]} \wedge \\
\qquad\qquad serving(s) = c) \wedge \\
\quad\ l = knownPossPlacesFor(c, s)\,]
\end{array}
$$

In this we use the following list manipulation functions: *queueAtFront(k,l)* which returns a list of places, i.e. the list k concatenated to the front of the list l; *removeElement(p,l)* which returns the list l with the place p removed; and *removeList(k,l)* which returns the list l with all elements in list k removed.

The above successor state axiom says that the fluent *knownPossPlacesFor* will be updated as follows. The first disjunct handles the case where the action *delFromKnownPossPlaces(c,p)* is performed; this happens when the robot has reached a place p, whether the person it is looking for, c, is found or not; in this case, place p is removed from *knownPossPlacesFor(c,s)*. The second disjunct deals with the case where the client sought, c, is found at the robot's current location *robotPlace(s)*. The action *restorePlaceList* resets the value of c's *knownPossPlacesFor* to that of the corresponding *possiblePlacesFor* with the robot's current location moved to the front; then the robot knows that this is the best place to find c. The third disjunct deals with the case where c is not found at the robot's current position, c's *knownPossPlacesFor* has become empty, and c has not run out of credit; then the *knownPossPlacesFor* will be reinitialized to the value of c's *possiblePlacesFor* by the primitive action *restorePlaceList* so that another attempt at finding c can be made. The fourth disjunct handles the case where c is not found and information on c's location has been obtained from a bystander; then the locations suggested by the bystander will be moved to the front of the existing *knownPossPlacesFor*. If none of the above cases applies, the value of the fluent will be the same as that in the previous situation.

Let us also point out one more aspect of the new specification of our robot's behavior. As mentioned, the robot can scan the current location to find out whether somebody is there (by action *scanPlace*). The information sent back from the visual sensors is stored in the functional fluent *vSensorReadings* for further processing; we will return to this later.

In order to test our approach to search, changes have been made to the program described in chapter 4. Getting blocked on the way to a place or the person not being found there no longer means that the current delivery task has failed. This is because the person may be found in another room that is reachable. Therefore, it is inappropriate to reduce the person's credit in either of these cases. Only when the the robot has searched all possible locations for the person, i.e. when the list *knownPossPlacesFor* becomes empty, will the credit of the person be reduced.

5.4 Primitive Actions and Fluents

Since this is an extension of the program presented in chapter 4, only those new or modified primitive actions and fluents will be mentioned. All new primitive actions are mainly for the purpose of either sensing persons nearby, querying, or manipulating the knowledge base. They are listed in Table 5.1.

Primitive Actions	Description
Sensing Action	
$scanPlace$	Scan for people around the current location.
Query Action	
$askAbout(c', c)$	Ask client c' where to find client c.
Operations on Knowlege Base	
$delFromKnownPossPlaces(c, p)$	Delete a place p from *knownPossPlacesFor* for client c.
$restorePlaceList(c)$	Set *knownPossPlacesFor* of c equal to the value in *possiblePlacesFor* of c.
Exogenous Actions :	
$msgFromRobot(found(t))$	A set of people t has been found during scanning by the robot.
$msgFromRobot(ansQuery(k))$	The customer has entered the set of places k as answer to our query.

Table 5.1: List of New Primitive Actions

As in the previous chapter, each action comes with an action precondition axiom, for example:

$$Poss(askAbout(c', c), s) \equiv (\exists t) \,.\, person(c) \wedge person(c') \wedge vSensorReadings(s) = t \wedge c' \in t$$

where t is a set of people (represented as a list). This says that the robot can ask client c' about client c if and only if c' is found in the current location. Appendix F.1 lists all new and modified action precondition axioms.

The new primitive fluents are listed in Table 5.2.

Primitive Fluents	Description
Predicate Fluents:	
waitingForAnswer	Holds if the robot is waiting for the client's answer. It becomes false if either the action *resetRobot* is performed or the client's answer is received.
highPriorityShptJustPlaced	Holds if a customer places a high-priority shipment when the robot is querying people nearby.
Functional Fluents:	
serving	The client who is currently being served.
vSensorReadings	The current readings of the visual sensors.
$knownPossPlacesFor(c)$	The current set of possible places where client c may be found.

Table 5.2: List of New Primitive Fluents

Every new primitive fluent also comes with a successor state axiom. For example, that of *vSensorReadings* is as follows:

$$
\begin{aligned}
& vSensorReadings(do(a,s)) = t \;\equiv \\
& \quad [\, a = msgFromRobot(found(t)) \,] \lor \\
& \quad [\, a = scanPlace \;\land\; t = \bot \,] \lor \\
& \quad [\, (\forall\, t')\, a \neq msgFromRobot(found(t')) \;\land\; a \neq scanPlace \land \\
& \quad\;\; t = vSensorReadings(s) \,]
\end{aligned}
$$

This says that the value of the fluent *vSensorReadings* after an action is a set of people (for example, $\{\, Jeff,\ Yves \,\}$) if the robot has just found this set of people in the current location, is undefined ($\bot$) if the action *scanPlace* is executed, and is the same as in the previous situation otherwise. The successor state axiom of *vSensorReadings* provides an example of how inputs from the robot are cached before they are processed by the control module. As we have already seen in section 5.2, the successor state axiom of *robotStatus* has been modified. Please refer to appendix F.2 for all new and modified successor state axioms.

In addition to the primitive actions and fluents, we also have three new defined fluents which are listed in Table 5.3.

The defined fluent *bestPlaceToGet* is defined as follows:

$$
\begin{aligned}
& bestPlaceToGet(c) = p \overset{\text{def}}{=} \\
& \quad (\forall\, l)\, [\, knownPossPlacesFor(c) = l \;\land\; l \neq [\;] \supset p = front(l) \,] \land \\
& \quad [\neg(\exists\, l)\, (\, knownPossPlacesFor(c) = l \;\land\; l \neq [\;]\,) \;\supset\; p = \bot \,]
\end{aligned}
$$

where *front(l)* returns the first element of the list l without removing it.

This essentially says that the best place to reach a client c is the first place found in c's *knownPossPlacesFor*. The definitions for the rest of the defined fluents appear in appendix F.3.

Defined Fluents	Description
Defined Fluents:	
interruptAllowed	Holds if the robot is in one of the following states: *moving, scanning* and *querying.*
$bestPlaceToGet(c)$	Returns the best place to reach client c, which is the first element in the list *known-PossPlacesFor* of c.
$robotFoundSomething$	Holds if the robot detects someone in the environment.

Table 5.3: List of Defined Fluents

5.5 Modified and New Procedures

Although the same four threads and the same procedures are used in this revised example, some new procedures have been added to implement the knowledge-based search. These new procedures are *isCustHere, somebodyHere, seekHelpFrom, checkPlaceList,* and *adjustCredit.* The procedure *serveCustomer* has been modified in order to make use of these new procedures. Here is the new version:

proc $serveCustomer$
$(\pi\, c, p)$ [$(bestCustToServe(c)\ \wedge\ bestPlaceToGet(c) = p)?;$
$startGoTo(p, c)$
$(robotStatus \neq moving)?;$
if $(robotPlace = p)$ **then**
$isCustHere(c, p)$
else if $(robotStatus = blocked)$ **then**
$resetRobot$
else noOp]
end

First notice that, it is now necessary to select both who is the best customer to serve as well as what is the best place to reach this customer. Second, when the robot arrives at the destination, the procedure *isCustHere* is called rather than doing pickups or dropoffs immediately. Lastly, when the robot gets blocked on the way to a place, the procedure now simply resets the robot instead of calling the procedure *handleGoToFailure* to reduce a client's credit. As mentioned earlier, a client's credit is now reduced only when the robot has searched all the possible locations for him[2]. These are all the modifications required to integrate knowledge-based search for people into our original program presented in chapter 4.

In order to implement knowledge-based search for our application, our control module must have plans to sense what people are present in a location, query bystanders about the location of people, and update the knowledge base appropriately. In our example, procedures *isCustHere* and *somebodyHere* together define a plan for sensing, while procedure *seekHelpFrom* defines a plan for querying and knowledge base updating.

The procedure *isCustHere* defines the plan for scanning a place and dealing with the results. It is as follows:

proc $isCustHere(c, p)$
  $scanPlace$;
  $(robotStatus \neq scanning)$?;
  if $(vSensorReadings = \perp)$ **then**
    noOp
  else $[\,delFromKnownPossPlaces(c, p)$;
    $somebodyHere(c)]$
end

[2]This modification results in inadequate behavior in some cases: the robot will keep trying to get to the same place when it gets blocked and the obstacle is never removed.

The procedure first initiates scanning and waits for the robot to come out of the *scanning* state. In the application, if the action *resetRobot* (done by the thread dealing with new orders) is performed while the robot is scanning a place, the scanning process will be interrupted and the robot will switch its state back to *idle* while leaving *vSensorReadings* undefined ($\perp$). In this case, there is nothing left to do and the procedure quits immediately. On the other hand, if the scanning process completes successfully, then a value (either an empty set or a set of people) is returned from the visual sensors. In this case, the robot should first remove p from *knownPossPlacesFor*[3] and then call procedure *somebodyHere* to process the input from the visual sensors.

The procedure *somebodyHere* determines what to do next depending on the results of the sensing:

proc $somebodyHere(c)$
$(\pi\, t)\,[\,(vSensorReadings = t)?;$
if $(c \in t)$ **then**
$[\,pickUpShipmentsFrom(c);$
$dropOffShipmentsTo(c);$
$restorePlaceList(c)\,]$
else if $(t \neq \emptyset)$ **then**
$[\,seekHelpFrom(t, c);$
if $(highPriorityShptJustPlaced)$ **then**
noOp
else $checkPlaceList(c)\,]$
else $checkPlaceList(c)$
end

It says that if the robot is looking for a client c and c has been found by the sensors, then c is served (i.e. the robot picks up shipments from and/or drops off shipments

[3]Execution of this action can be deferred to the beginning of the outermost **else** block of the procedure *somebodyHere*. We put it here just for the sake of convenience.

to c) and then the contents of *knownPossPlacesFor* for c is restored to the value of c's *possiblePlacesFor* for a possible later search. If, on the other hand, the sensors determine that c is not there but other people are, then the robot will first seek help from these people, and then call the procedure *checkPlaceList* to check c's credit and see whether c's *knownPossPlacesFor* has become exhausted. Furthermore, when a high-priority shipment order is received, the thread which is responsible for handling new orders will perform the action *resetRobot*. If this happens while the robot is in the state *querying*, the fluent *highPriorityShptJustPlaced* will become true. As a consequence, the querying process will be interrupted and the procedure will terminate. This allows the robot to switch its goal to serving the high-priority shipment.

The procedure *seekHelpFrom* defines the plan to seek help from the set of people t at the current location, to find out where a client c is:

proc $seekHelpFrom(t, c)$
    for $(p : t$ **until** $robotStatus = idle)$ **do**
        if $(\exists\, m)(mightKnowBy(c) = m \;\wedge\; p \;\in\; m)$ **then**
            $[\, askAbout(p, c);$
             $(\neg waitingForAnswer)?;\,]$
        else noOp ;
    endFor
    if $(robotStatus \neq idle)$ **then**
        $resetRobot$
    else noOp
end

The procedure directs the robot to query all people in t who might know c's location (as represented by the *mightKnowBy* relation) one after the other. This stops when new information is obtained, or all such people have been queried, or the process is interrupted by the action *resetRobot* performed by the thread responsible for dealing with new orders (*waitingForAnswer* becomes false and *robotStatus* becomes *idle* if this happens). For each person, the robot makes a query and waits for their answer. If the answer contains places to find client c, it updates the knowledge base and the procedure terminates. Otherwise, another person is queried.

As mentioned in the description of the procedure *somebodyHere*, we need another procedure *checkPlaceList* to check whether the robot has run out of possible locations for a client c (as modeled by the fluent *knownPossPlacesFor*) and deal with this:

proc $checkPlaceList(c)$
  if $(knownPossPlacesFor(c) = \texttt{[ ]})$ **then**
    $[\, adjustCredit(c);$
     if $(clientSuspended(c))$ **then**
       noOp
     else $restorePlaceList(c)\,]$
  else noOp
end

If this is the case, the procedure first calls procedure *adjustCredit* to handle the credit reduction and possible client suspension for c. Then, if c has not run out of credit, the list *knownPossPlaceFor* for c will be restored to its initial value, i.e. c's *possiblePlacesFor*; this will cause the search for c to start over.

The following is the definition of the procedure *adjustCredit*:

proc $adjustCredit(c)$
  $(\pi\, k)\, [\, (haveCredit(c) = k \;\wedge\; k > 0)?;$
     $ReduceCredit(c);$
     if $(haveCredit(c) = 0)$ **then**
       $notifyStopServing(c)$
     else noOp $]$
end

This procedure simply reduces a client's credit and suspends the client if his/her credit level has reached zero.

5.6 A Sample Test Run

In order to show that the control program really does what we want, we have tested it in simulation mode with an environment having the following characteristics. The people working in our office are Yves, Michael, Jeff and Hector. In the knowledge base, in addition to facts that define places and people working in the office, we have the following new facts:

$$possiblePlacesFor(Yves) = [\,Lab1,\ Lab2,\ Lab3\,]$$
$$possiblePlacesFor(Michael) = [\,Lab3,\ Lab4\,]$$
$$possiblePlacesFor(Jeff) = [\,Lab4,\ Lab1\,]$$
$$possiblePlacesFor(Hector) = [\,Lab2,\ Lab4,\ Lab3\,]$$

This says that Yves may be found (in decreasing order of likelihood) in Lab1, Lab2 and Lab3; Michael may be found in Lab3 and then Lab4, etc. We also have the following:

$$mightKnowBy(Yves) = \{\,Michael,\ Jeff\,\}$$
$$mightKnowBy(Michael) = \{\,Jeff\,\}$$
$$mightKnowBy(Hector) = \{\,Yves\,\}$$
$$mightKnowBy(Jeff) = \{\,\}$$

The above means that the location of Yves might be known by Michael and Jeff, the location of Michael might be known by Jeff, etc.

In addition, we have the following initial state axioms:

$$robotPlace(S_0) = robotHome$$
$$approaching(S_0) = nowhere$$
$$robotStatus(S_0) = idle$$
$$robotAlive(S_0)$$
$$shipmentCtr(S_0) = 1$$
$$vSensorReadings(S_0) = \perp$$
$$(\forall\, c) \,.\, person(c) \supset haveCredit(c, S_0) = 1$$
$$(\forall\, c) \,.\, person(c) \supset knownPossPlacesFor(c, S_0) = possiblePlacesFor(c)$$

These altogether say the following. The robot is at *robotHome* and going to *nowhere*. That is, the robot stays stationary at *robotHome*. The *robotStatus* is initially set to *idle* and the first shipment will be given the shipment number "1". Every person has only one credit to start with. Nothing is available from visual sensors and the contents of the fluent *knownPossPlacesFor*, is initially set to the value of the corresponding non-fluent *possiblePlacesFor*.

The full listing of the program written in ConGolog and a trace of the primitive actions produced in one execution of the program given the above initial state specification can be found in appendix E and G, respectively.

If we examine the trace and the actions performed in providing service to Yves carefully, it is not difficult to see that our module does exhibit the expected behavior. First, the place list gets restored after the robot has picked up or dropped off shipments (lines 14 and 21). Second, a detailed examination shows that the robot always goes to the first place in the *knownPossPlacesFor*. Third, shipments will be picked up or dropped off only if the sender or the recipient is found (lines 53 and 85). If someone else is found, only people for which the relation *mightKnowBy* holds will be queried (lines 74 and 95). Lastly, an interruption of service is demonstrated in line 34.

5.7 Summary: On Knowledge-Based Search

In this chapter, we have developed an application that uses the knowledge-based approach to search to locate people in an office setting. Our example has shown that this approach is able to deal with the problems caused by having incomplete information about the domain. The robot's knowledge about the location of people is updated appropriately after sensing is performed. This helps shrink the search space or, at least, increases the chance of finding the person sought.

We have tested the control program developed in a simulated environment (see section 5.6) and our "simulated robot" has proven to behave as expected. But it was impossible to extend the real robot's hardware and software to perform the additional sensing, which is somewhat complex as vision is required, in the available time for this project. We believe that it should not be too difficult to do so and test the control program in a real environment given more time.

Chapter 6

Our Robot Architecture

This chapter describes a simple, three-layered architecture that provides reactivity to runtime failures and significant exogenous events, and handles tasks with multiple goals. The architecture supports a knowledge-based approach, i.e. uses a knowledge base and reasoning to select actions. Since this thesis is focused on high-level robot control, we describe in detail how our high-level control module (with a knowledge base incorporated) is structured and how it fits within an overall robot architecture.

6.1 Overview

In chapter 2, we discussed the pros and cons of various approaches to robot architecture. We saw that the hybrid approaches, which combine aspects of the deliberative and reactive approaches, support the construction of robots that are able to handle complex tasks with multiple goals and respond to environmental changes reactively. Our proposed architecture is based on a hybrid approach.

There are two ways to combine the reactive and deliberative approaches. The first one is to make deliberation part of the control loop. Then work on the formulation, selection, and execution of plans may occur at every cycle. Since, at any time, either the high- or low-level control module is inactive, a lot of time is spent waiting. Utilization of the system resources is, thus, rather low in this kind of arrangement.

The second way is to make the deliberation and reactive modules run in parallel. Then the resulting agent can still react quickly to environmental changes while the high-level control module is doing some deliberative work. In addition, parallelism simplifies the allocation of resources. Therefore, we use the latter approach in our design. Note that we must ensure that computing resources are sufficient to provide the required response time. This may involve using separate processors or a single sufficiently fast host, that is allocated fairly between processes.

The architecture we use is composed of three layers and each layer corresponds to a module/module (see figure 6.1): a low-level robot control module connected to the robot hardware at the bottom, a path planning and navigation module in the middle, and a high-level robot control module at the top. Communication between layers is implemented using TCP/IP sockets. In our architecture, perceptual data and feedback on status of task goes from the bottom to the top while control (or instructions) goes mainly from top to bottom.

The high- and low-level robot control modules are structured in such a way that they run in parallel. That means the lower layers remain active when the upper layers are doing other work. This is achieved by making each layer an individual, standalone module which communicates with the adjacent layer through TCP/IP.

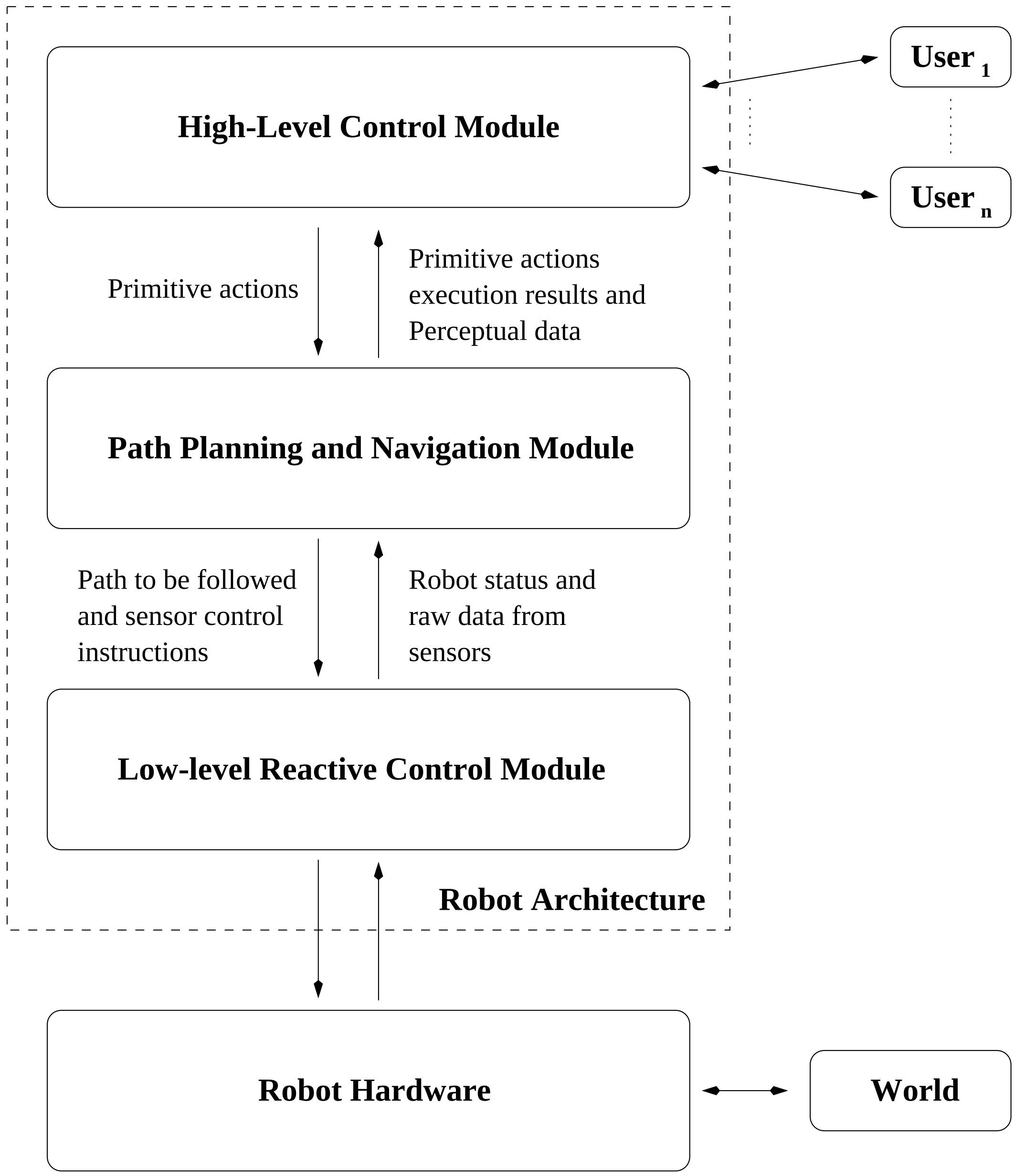


Figure 6.1: Our Robot Architecture

In some sense, other than the reception and sending of packets (either instructions or events), everything is done independently within each module.

Furthermore, note that the low-level control module can adapt instructions from a higher layer if it sees that they are not appropriate in the current situation. This is related to Agre *et al.*'s [2] notion of plans-as-communication.

6.2 The Robot Hardware

At the bottom of our architecture, we have a low-level robot control module connected to the robot hardware. The implementation of this module has been interfaced to a RWI B12 and a Nomad200, and is relatively portable. These robot platforms come with various kinds of sensor such as bumpers, infra-red sensors, sonars and/or visual sensors. These sensors are used for obstacle avoidance, collecting information about the robot's immediate environment, and correcting odometry error. In addition, the robot may also have some devices that make interaction with the outside world possible. These devices include user input devices such as key-pad mounted on the robot, and output devices such as sound-card and speaker, LCD display mounted on the robot, etc.

The robot used in our experiments is called "Mailmond" (pronounced as "mail-man"), which is an RWI B12-based mobile robot (see figure 6.2). It utilizes a three-wheel synchronous drive mechanical system; all wheels remain parallel at all time; and it has three translation ratios, namely high, medium, and low [49].

Mailmond comes with 6 bumper sensors to detect contact with objects. Twelve sonar sensors (one for each of its 12 side-panels) are used to determine the

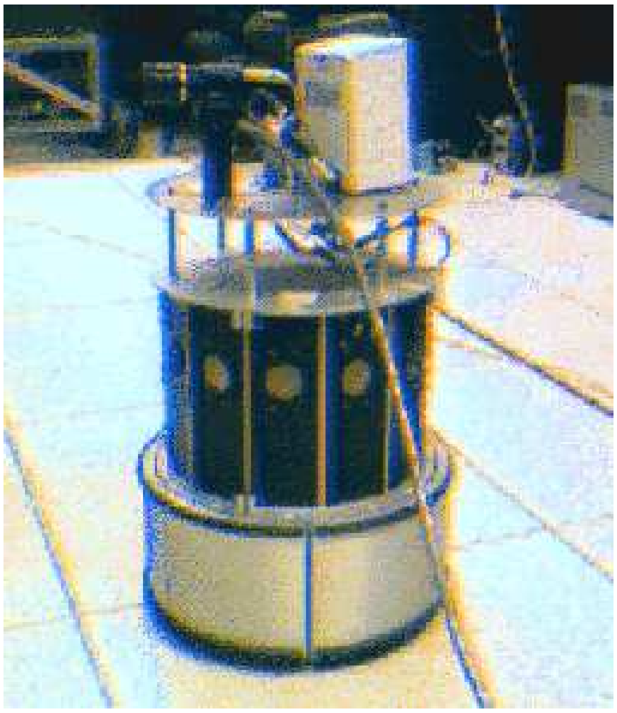


Figure 6.2: Mailmond

distance to objects and a speaker has been mounted to allow spoken messages to be issued to people nearby.

6.3 The Low-Level Robot Control Module

The *low-level robot control module* is mainly used to plan and control servo motions for the robot's wheels and the sensors. This control module can perform such primitive actions as "go forwards by 5cm", "take a picture", "say 'hello' ". Furthermore, it is also responsible for carrying out such "reflex" actions as obstacle avoidance, which are time-critical and must be performed with no deliberation because either the robot or people nearby will be hurt otherwise.

The low-level robot control module used on Mailmond is essentially the Brooksian-like reactive module developed for the *ARK* project [43, 27, 53].

6.4 The Path Planning and Navigation Module

The main function of the *path planning and navigation module* is to plan a path between any two points on an internal world model using various path-finding algorithms such as breadth-first search, A^* search, etc. The world model used in this module can be a geometric or topological map of the environment. With the presence of this layer, the high-level control module in section 6.5 can focus on high-level tasks without paying much attention to lower level details such as how to plan a path from a place to another and how to get the robot to follow it. In addition, the module also monitors the robot's operations on a continuous basis so that the current robot status can be reported and unexpected behavior can be detected.

The path planning and navigation module we use is called the "*Navigator*". It was also developed for the ARK project. As described in [11], the original Navigator

> "$\cdots$ is a program that provides a point and go user interface to a real or simulated mobile robot. The interface provides a 2D floorplan on which robot actions are planned, and a mechanism by which these plan can be executed by a mobile robot. $\cdots$ The major function of the navigator is to reason or plan motions which will be executed by the reactive subsystem. The navigator plans in a Good Old Fashioned AI

and Robotics (GOFAIR) paradigm. It assumes a static, full, modeled environment and counts on the reactive system to deal with unmodeled obstacles and objects."

In other words, the internal world model maintained by the Navigator is essentially a geometric map of the environment. All paths are planned by applying path-planning techniques using the map information. Figure 6.3 is a screen shot of the Navigator's graphical user interface window.

In the ARK project, users interacted with the robot directly using the Navigator's user interface. To allow the robot to be controlled by a high-level control module, the functionality of the original Navigator has been augmented with an interface for communication with the high-level control module. In this arrangement, the Navigator is responsible for path-planning and navigation tasks while leaving task-level control decisions to the high-level control module. As a result, the robot can be smarter and more autonomous. In addition, this helps in abstracting the low-level details, such as where a place is and how to get there, away from the high-level control module. Therefore, the high-level control program can be written as cleanly as possible.

With this interfacing, the new Navigator still plans in a GOFAIR manner, but unmodeled obstacles and objects are no longer handled only by the low-level reactive module. They are now also dealt with by the high-level control module. This is very important because, as mentioned in chapter 4, there are some situations that cannot be handled by the reactive module.

As mentioned, perceptual data in our architecture goes mainly from the bottom to the top while control goes mainly from the top to the bottom. Therefore,

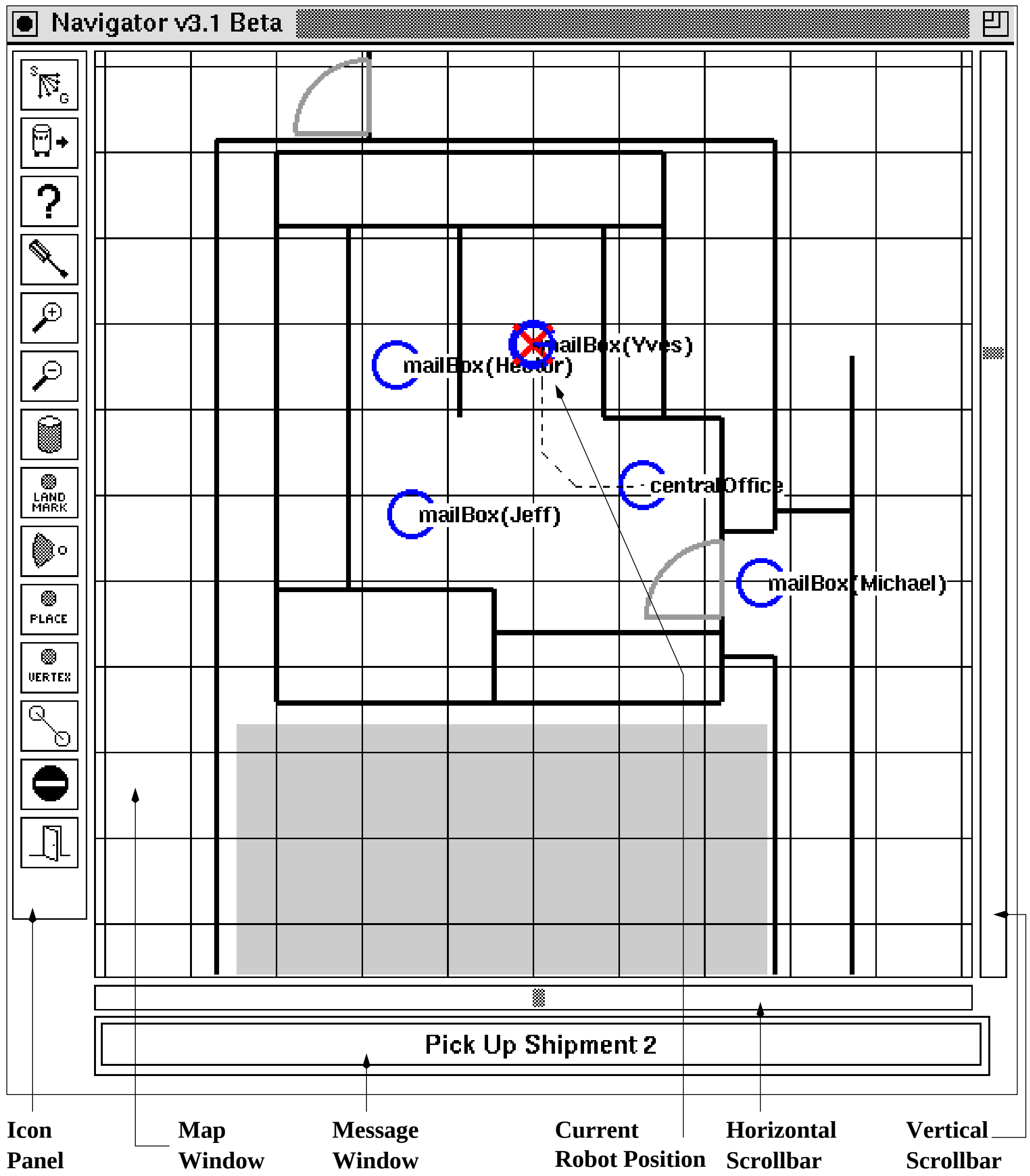


Figure 6.3: Navigator Screen

both control and perceptual information have to go through this layer. This layer takes inputs from the high- and low-level control module. All control instructions from the high-level control module other than those that require a change of the robot's position are passed on to the robot hardware or the low-level control module without modification. For instructions that request a change in the robot position, the path planner first plans a path which is a sequence of way-points between the source and destination according to the map. These way-points are then sent to the low-level robot control module for execution. Messages from the low-level control module are forwarded back to the high-level control module after relevant information has been extracted for the operations of the path planner.

6.5 The High-Level Robot Control Module

The high-level robot control module is responsible for deliberation, high-level plan execution, maintaining a knowledge base, and communication with users in the outside world at a high level. It is made up of a *knowledge base*, a *high-level control program*, an *interpreter* for the programming language used, and *interfaces to users and the Navigator*. The architecture of our high-level robot control module is depicted in figure 6.4.

In our work, the high-level control program is written in ConGolog and therefore, the ConGolog interpreter is used.

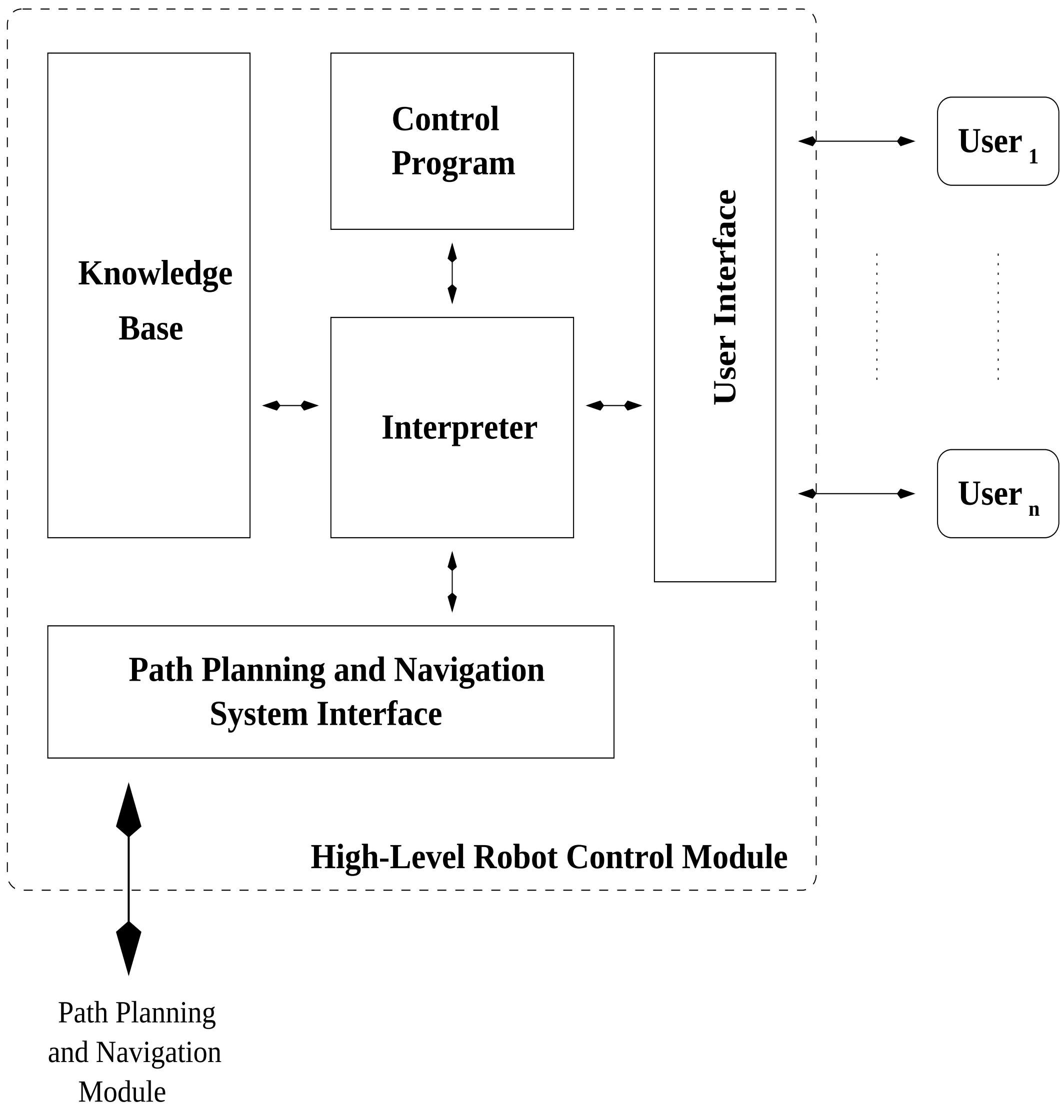


Figure 6.4: The Architecture of the High-Level Robot Control Module

6.5.1 The Knowledge Base

The knowledge base contains a domain theory, i.e. all domain-dependent facts known by the robot, inference rules about the facts, and "preconditions" and "postconditions" for every action as described in section 3.1.2. Our knowledge base contains two types of facts (or knowledge): fluents and non-fluents. Distinctions between these two types of facts have been discussed in section 5.3. In our implementation, all information obtained from outside the high-level control module is treated as exogenous events. These events cause the knowledge base to be updated when information is received. In the experiment in chapter 7, the high-level control module is built with a knowledge base containing all axioms and fluent definitions presented in section 4.5.

6.5.2 The High-level Control Program

The robot's behaviors are specified by a high-level control program. We use the ConGolog programming language to express high-level programs. The advantages of using ConGolog were explained in chapter 3. The programs we use are typically expressed as a set of prioritized interrupts running in parallel so that exogenous events and runtime failures can be handled. An example of such a program was presented in section 4.6 and this is the one that we use in our experiment in chapter 7. The body of each interrupt contains calls to procedures and each procedure is designed to attain certain goals. An interrupt will be triggered when certain conditions in the world are true. This may result in a suspension of the current goal and selection of new plan. Therefore, the robot is highly reactive and able to cope with changes in the environment.

6.5.3 The interpreter

The ConGolog interpreter is used to interpret the high-level control program. Interpretation proceeds as explained in section 3.2.2. The standard version of the interpreter in [13] has "to look all the way to a final goal state before even a single action can be taken in the world" [14]. This may result in long delays before a response to an (anticipated or unanticipated) event can occur. Such deferral is unacceptable in most robotic applications. To cope with this problem, a new incremental interpreter is developed in [14], which is described as "a new incremental way of interpreting such a high-level programs ... (which) allows much more control to be exercised over when actions can be executed" [14]. In other words, it tries to minimize delay by interpreting the program in an incremental way. However, in our implementation, the standard version of the interpreter is used. These issues are sidestepped by having the interpreter commit to and execute all primitive actions immediately as it encounters them in the program. Moreover, the interpreter continuously monitors for exogenous events and inserts them in the execution as soon as they are detected [34].

6.5.4 Interfaces to Users and the Navigator

This sub-module makes interaction between the high-level control module, and the path planning and navigation module, and users in the outside world possible. It is responsible for communication tasks such as connection and acceptance of sessions, sending and receipt of packets, buffering, validation of messages, acknowledgment, etc. All messages flowing between the module and the rest of the system or users must go through this sub-module before they are read by the interpreter or sent out to the appropriate recipient. In addition, this sub-module can only be called by the interpreter. The interface is written in Quintus Prolog.

6.6 The User Control Panel

It is not reasonable to restrict all users of a robotic application like mail delivery to have access to the same host computer. We have developed a *user control panel*, through which a user can place shipment orders, that is executable on multiple platforms and able to communicate with the system over a network. Our user control panel is implemented in *Java*, a programming language that has strong support for network communication and graphical user interfaces (GUI). As long as there is a Java virtual machine on all hosts, the control panel can be run there. Thus, the user can place shipment orders from anywhere over the network. Figure 6.5 is an annotated screen shot of user Jeff's control panel.

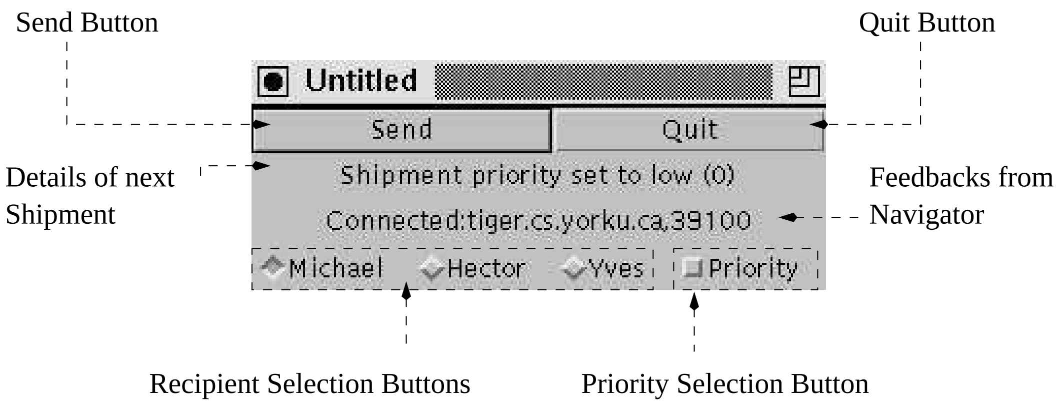


Figure 6.5: Jeff's Control Panel

6.7 Advantages of our Approach

In contrast to architectures such as *TouringMachines* [15] and InteRRap [41], we do not include in our architecture a complex mediation scheme or complicated layering structure. Our simple design makes robot construction easier.

By making the three layers run in parallel, i.e. having the low-level reactive module remains active while the high-level control module is deliberating and the Navigator is planning paths, the robot is capable of making instant responses to perils in the environment. Since there is a deliberation layer, the robot system can also reconsider plans in order to recover from runtime failures, deal with significant exogenous events, and handle tasks that involve reasoning and multiple goals. Parallelism also helps to achieve a better allocation of resources.

6.8 Summary: On Our Robot Architecture

In this chapter, we have proposed a simple, three-layered architecture that is both reactive, and capable of handling tasks involve reasoning and multiple goals. The three layers are a low-level robot control module connected to the robot hardware at the bottom, a path-planning and navigation module in the middle, and a high-level robot control module at the top. Layers are connected through TCP/IP sockets and are designed to run in parallel. The resulting robot is reactive to runtime failures and significant events, and is capable of handling tasks involving multiple goals.

Chapter 7

Experimentation

In this chapter, we will describe experiments in a real environment with our exogenous event and runtime failure handling control module (see chapter 4 for details), and the robot architecture introduced in chapter 6. We first describe the expected behavior of our robot, then describe the experiment itself, and finally conclude by pointing out some major issues for our implementation.

7.1 Setting and Expected Behavior

The experiment is done in a laboratory setting consisting of two rooms, the robotics lab and graphics lab, which are connected by a long hallway. We assume that there are four people working in the laboratory: Yves, Hector, Jeff and Michael. Yves and Hector are working in one of the rooms while Jeff and Michael are working in the other. Figure 7.1 is a map of our laboratory area in which the locations of the "legal places", namely *mailBox(Jeff), mailBox(Yves), mailBox(Hector), mailBox(Michael)* and *centralOffice*, are shown. We arrange for these customers to place shipment orders at runtime to test our robot's reaction to exogenous events and runtime failures.

For every valid shipment order[1], the robot goes and picks up the shipment from the sender's mailbox, and then goes and drops it off at the recipient's mailbox. When the robot is at someone's mailbox, it will tell the person being served to pick up a shipment from or put a shipment on the robot via the speaker mounted on it. The playback of the pre-recorded messages is controlled by the software.

When an obstacle is encountered, the robot halts and waits for 15 seconds. Then if the obstacle is no longer there, the current delivery task will be resumed. Otherwise, the robot will reduce the credit of the person being served by one, and find an alternative goal to pursue, i.e. find another person to serve. Every person in the laboratory starts with a credit of two and people who have run out of credit will no longer be served.

All other behaviors, such as how to choose customers to serve, what to do with a new shipment order, when to reduce a customer's credit, etc., and the overall strategy used by the robot to accomplish the mail delivery task are specified as in chapter 4, where the control program used in this experiment is described in detail.

7.2 Experiments

The following two experiments are aimed at testing whether our system as implemented on the real robot (with the control program to handle exogenous events

[1]A valid shipment order is a shipment order for which both the sender and recipient have not been suspended.

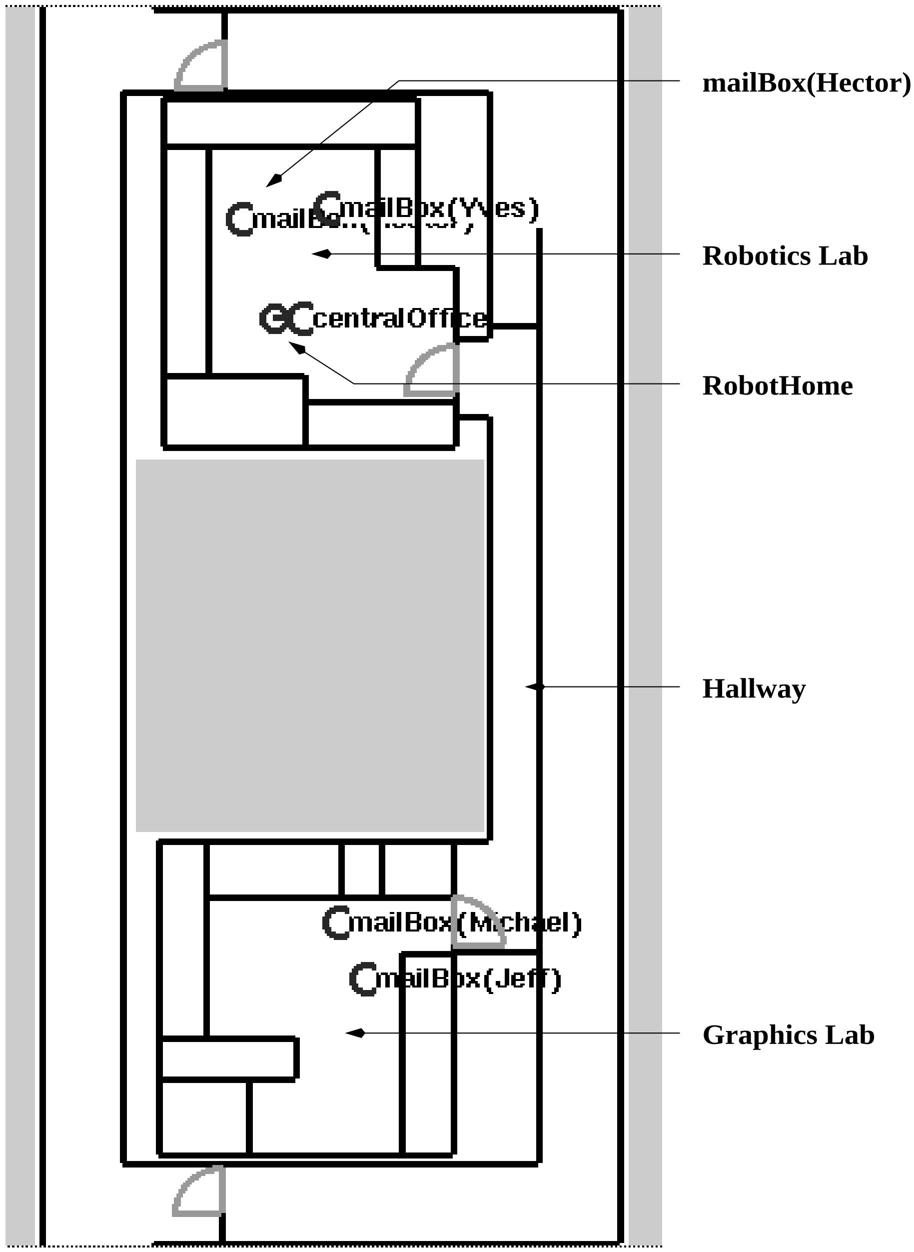


Figure 7.1: Map of our laboratory

and runtime failures, which is depicted in figure 6.1) is able to exhibit the specified behavior.

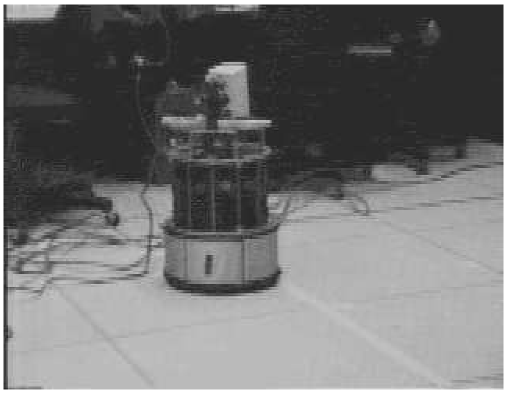
7.2.1 Experiment 1: Reaction to Exogenous Events

The objective of this experiment is to show that our system can react to exogenous events at runtime. To test this, we arrange for a new shipment order with high priority to be made while the robot is serving a low-priority shipment. We then see how the robot behaves.

The experiment goes as follows. The robot is initially in the robotics lab at location *robotHome*. First, Hector places a low-priority shipment order to Yves at the very beginning. The robot reaches Hector's mailbox; picks up the shipment; and sets off to Yves' mailbox. On the way to Yves' mailbox, the robot finds that a high-priority shipment order has been placed by Michael which is addressed to Hector. The robot will halt and start serving Michael's order. It will go to Michael's mailbox in the graphics lab, pick up the high-priority shipment from Michael, then return to the robotics lab, and drop the shipment off to Hector's mailbox. It will then resume serving the original order, i.e. delivering the low-priority shipment onboard to Yves. After the shipment has been dropped off to Yves' mailbox, the robot goes to *centralOffice* and retires. Figure 7.2 to 7.6 contain pictures taken from the experiment.

7.2.2 Experiment 2: Reaction to Runtime Failures

This experiment is aimed at showing reactivity in response to runtime failures. In the context of this experiment, an obstacle is any object encountered on the path

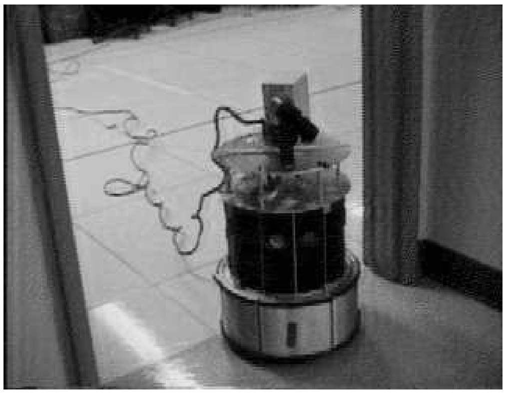


a. Mailmond at *robotHome*

b. Mailmond at Hector's mailbox to pick up shipment 1

c. Mailmond going to Michael's mailbox after being interrupted

Figure 7.2: Pictures Taken From Experiment 1 – Part I

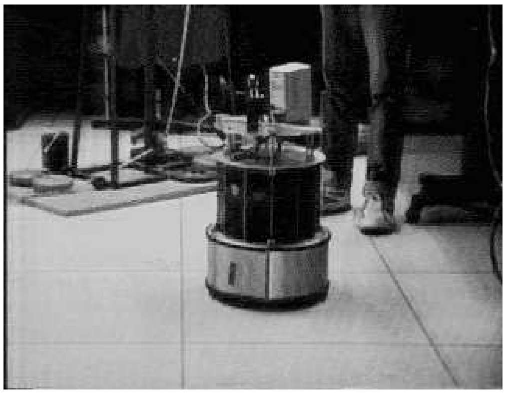


d. Mailmond leaving the robotics lab

e. Mailmond in the hallway heading to the graphics lab

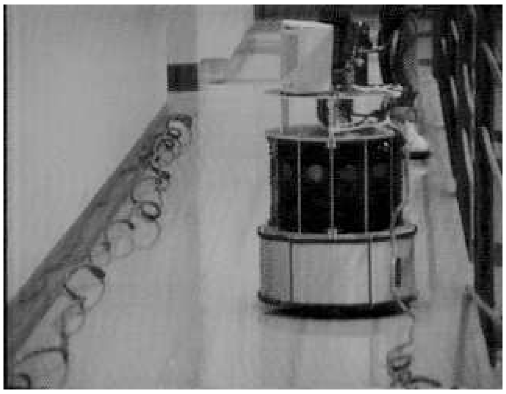
f. Mailmond entering the graphics lab

Figure 7.3: Pictures Taken From Experiment 1 – Part II



g. Mailmond at Michael's mailbox to pick up shipment 2

h. Mailmond leaving the graphics lab



i. Mailmond in the hallway heading back to the robotics lab

Figure 7.4: Pictures Taken From Experiment 1 – Part III

j. Mailmond entering the robotics lab

k. Mailmond at Hector's mailbox to drop off shipment 2

l. Mailmond at Yves' mailbox to drop off shipment 1

Figure 7.5: Pictures Taken From Experiment 1 – Part IV

m. Mailmond retires at *centralOffice*

Figure 7.6: Pictures Taken From Experiment 1 – Part V

followed by the robot where the distance between them is less than 25 centimeters. An obstacle is considered "temporary" if it blocks the robot's way for less than 15 seconds, otherwise, it is considered "permanent".

The experiment is conducted as follows. Hector first places a shipment order (all shipments are assumed to have the same priority level) addressed to Michael. On the way to Hector's mailbox, the robot finds that its path is blocked by someone. After a spoken request is made by the robot, the person gives way and the robot continues its trip to Hector's mailbox. The robot picks up the shipment from Hector and then proceeds to Michael's place. However, on the way to Michael's mailbox, the robot finds that someone has left something near the doorway of the robotics lab that blocks the path to Michael's mailbox. As well, a new shipment order is placed by Yves which is addressed to Hector. After being blocked for 15 seconds, the robot gives up the current plan (going to Michael's mailbox) and switches its goal to serving Yves. It goes and picks up the shipment from Yves, and then tries to deliver it to Hector's mailbox. Unfortunately, the path is blocked by another obstacle. After two unsuccessful attempts to "get around" it, Hector is suspended and the undelivered shipment is returned to Yves. After that, the robot tries to go to Michael's mailbox for a second time and finds that the path is still blocked. It then decides to suspend Michael. Since there are no more valid shipments (Hector has been suspended earlier), the robot returns to *centralOffice*, drops off the invalid shipment onboard, and then retires. Figure 7.7 to 7.10 show pictures taken from the experiment.

7.3 Issues Raised by the Experiments

Now, let us discuss some of the major issues raised by our implementation and the experiments performed.

The Situation Term Gets Large in Long Runs

The situation term used by the program grows in length each time an action is performed. This implies that the longer the experiment, the more time is required for reasoning[2], and the more inefficient the program's interpretation becomes.

To solve this problem, our implementation does "*rolling forward*" [36]. Rolling forward, in plain English, is a technique that turns the current situation into a new initial state. The axioms in the knowledge base must be updated to reflect the new state of the system. By rolling the current situation forward, we can limit the length of the situation terms involved in the execution of the

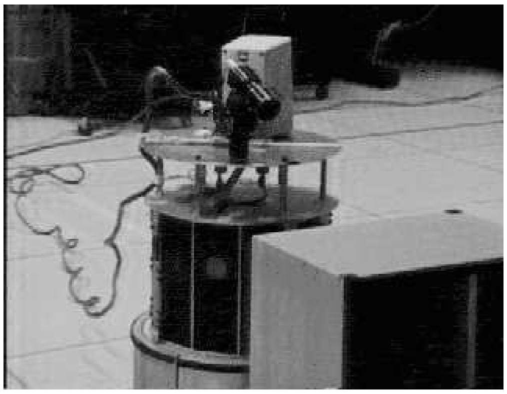
[2]Determining whether a test or precondition holds is done by regressing it to a condition on the initial state and querying the knowledge base [50]. The length of the regression process depends on the length of the situation term.

a. Mailmond at *robotHome*

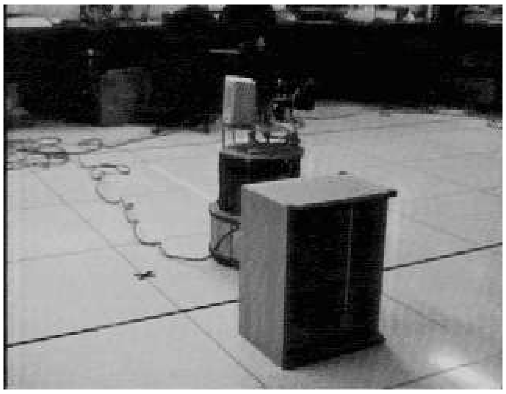
b. Mailmond is blocked by a temporary obstacle on the way to Hector's mailbox

c. Mailmond continues its trip to Hector's mailbox after the obstacle is removed and shipment 1 is picked up

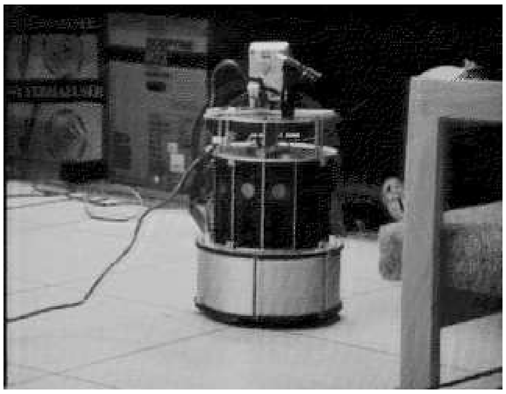
Figure 7.7: Pictures Taken From Experiment 2 – Part I



d. Mailmond is blocked by a permanent obstacle on the way to Michael's mailbox



e. Mailmond gives up the plan to go to Michael's mailbox and switches its goal to serving Yves

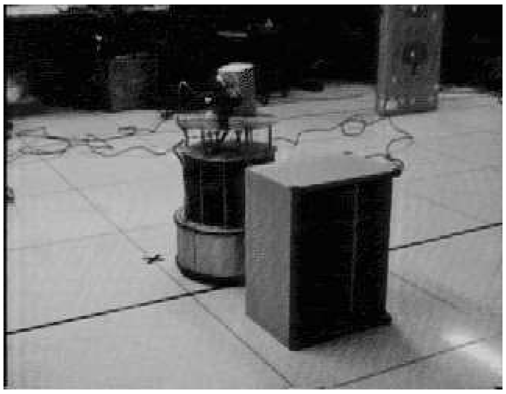


f. Mailmond at Yves' mailbox to pick up shipment 2

Figure 7.8: Pictures Taken From Experiment 2 – Part II

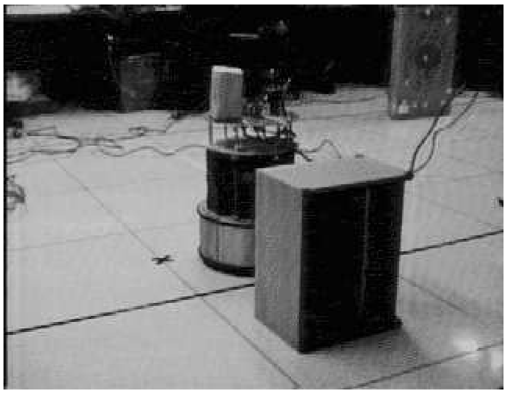
g. Mailmond is blocked by a permanent obstacle on the way to Hector's mailbox

h. Mailmond going to Yves' mailbox to return shipment 2



i. Mailmond is blocked again by a permanent obstacle on the way to Michael's mailbox

Figure 7.9: Pictures Taken From Experiment 2 – Part III



j. Mailmond gives up the goal to go to Michael's mailbox and returns to *centralOffice*

k. Mailmond retires after dropping off shipment 1

Figure 7.10: Pictures Taken From Experiment 2 – Part IV

program. Since the value of any fluent can now be obtained by reasoning with a shorter history, the time required for deliberation is shortened.

Low-level Reactivity

To achieve low-level reactivity, the low-level reactive module should reside onboard the robot because the time spent on data transmission between sensors and the reactive module can be shortened. However, in the current implementation, the low-level reactive module is located off-board. Communication between the robot hardware and the reactive module relies on serial links established via cables, which is both slow and inefficient.

Position Correction

For a robot to do navigation in some area using a map, it is necessary for it to know its position and orientation. If the robot operates for long periods, odometry error accumulates and the position estimate must be corrected. There are a lot of ways to do this. For example, landmarks can be put up in the environment and the robot can track them via active or passive sensing.

Computational position-correction techniques were not used in this experiment because of time limitations and because the issue is largely incidental to our concerns. However, any long experiment done on a mobile robot without position correction will run into difficulties. Therefore, in our experiment, the robot's position and orientation were corrected manually when the accumulated error in translation and rotation got too large.

Other Issues

There are some other issues that we will not discuss in details here because they are not within the focus of this thesis; these include noisy sensors, completeness and robustness of the auxiliary programs such as user's control panel, etc.

7.4 Concluding Remarks

In this chapter, we have shown that our approach to handling exogenous events and runtime failures can be realized on a real robot built on the architecture presented in chapter 6. The robot did exhibit the desired behaviors as specified by the control program presented in chapter 4.

The implementation used in the experiments was completed under time constraints and hardware configuration limitations. Due to these constraints, the control module for knowledge-based search for people from chapter 5 could not be tested with a real robot and environment.

Chapter 8

Conclusion

In this thesis, we have developed a simple, three-layered architecture based on a hybrid approach. Using this architecture, we have investigated how to handle runtime failures and significant exogenous events by prioritized software interrupts. Furthermore, we have also explored ways of implementing search for people using a knowledge-based approach to search. Now, let us conclude this thesis by discussing some advantages of our approach, the major contributions of this thesis, how this work relates to previous work, and some future research directions.

8.1 Summary of Our Approach

Let us first summarize our approach and its major advantages. Our hybrid architecture has low- and high-level robot control modules running in parallel; thus, our agent is capable of making instant response to perils in the environment, recovering from runtime failures, dealing with significant exogenous events, and handling tasks that involve reasoning and multiple goals. In addition, by reasoning with a knowledge base whose contents is kept current, better decisions can be made and the actions selected are more appropriate to the current situation.

By using prioritized software interrupts and threads, the agent is able to execute two or more tasks concurrently. Since different threads may engage in different tasks, the whole system will not be held up when a thread is waiting on signals from users or the robot because other threads are still proceeding with productive tasks. This results in a better allocation of system resources. In addition, software interrupts and threads keep the costs involved in the development of a robust robot control program low, as modifications and extensions can be made without rebuilding the whole system from scratch.

When searching for an object or person, the more knowledge the robot has about the location of object, the more the search space can be pruned and the more time-efficient the search can be. In addition, by using a knowledge base and maintaining its contents current, our approach has a better chance to find mobile objects.

Lastly, as shown in our examples in chapter 4 and 5, the robot's behaviors are specified at many levels of abstraction. The high-level abstractions consist of complex actions, which are structured as procedures, and a global strategy to achieve the top-level goals. The low-level abstractions consist of primitive actions, which are specified according to a theory of action. Control/scheduling knowledge, for example, who is the best customer to serve, what is the best place to reach this customer, etc., can be specified declaratively as defined fluents. This arrangement gives us two major advantages:

- the existence of a low-level abstraction of tasks enables the programmer to write control programs at a high level without paying much attention to the low-level details; and

- the different levels of abstraction make coding, modification, debugging, and verification of the program easier.

To illustrate the second point, let us take the following as an example. If one wants to change the strategy of serving customers to always serve the one with the smallest (oldest) shipment number. All one needs is to rewrite the defined fluent *BestCustToServe* rather than the whole program. In addition, as long as the definition of *BestCustToServe* is correct, the program will do what is expected.

8.2 Contributions

The major contributions of this thesis are as follows:

1. Previous work on ConGolog has shown that it can be used to build software agents [31] and a robot operating in a simulated world [13]. In this thesis, we have gone further to demonstrate its ability to control a robot operating in a real environment and show that programs written in ConGolog are able to accomplish realistic tasks.

2. Although much previous work has tried to tackle problems caused by the runtime failures either at a low- or high-level, only a few architectures or languages can deal with both exogenous events and runtime failures that require reconsidering the current plan and possibly adopting a new one. In this thesis, we have proposed a general approach to handle both runtime failures

and exogenous events using threads and prioritized software interrupts in a high-level control module.

3. It seems that previous work on search to localize objects/people does not fully apply the knowledge-based approach to search because search is done based on a set of facts written into the knowledge base once and for all. Information obtained at runtime is primarily used only for the expansion of current plans and the knowledge base is never updated. Our contributions in this aspect are two-fold:

   - we have explored how ConGolog can be used to implement a knowledge-based approach to search for locating objects/people when there is incomplete information;
   - we have demonstrated the importance of knowledge acquired at runtime in search and shown how it can be exploited; and

4. We developed a robot architecture based on a hybrid (reactive-deliberative) approach. It involves three layers of control with a knowledge base incorporated in the high-level control module. Control in our architecture flows in a top-down manner while perceptual data and signals/status reports flow bottom-up. Our high-level control module has a modular design and uses a knowledge-based approach. The high-level language supports deliberation and knowledge base update based on a theory of action. By using prioritized interrupts and Agre *et al.*'s [2] plans-as-communication model, we need not implement any mediation scheme to resolve conflicts in the selection of actions.

5. We have demonstrated the advantages of writing robot programs in a high-level knowledge-based language. The resulting programs or procedures should be more transferable between tasks or domains. In addition, our knowledge-based language supports different levels of abstraction, which in turn makes the resulting program easier to be understood and less error-prone. Therefore, extension, verification, and debugging of the program are easier.

6. Actions in the Situation Calculus are assumed to be instantaneous. This is unrealistic because in the real world many actions have a long duration and can occur concurrently with other actions. In this thesis, we have shown how an approach that models long duration actions with instantaneous actions can be used effectively in real applications.

8.3 Relation to Previous Work

In terms of the architectural design, our approach is similar to the InteRRap model [41] (see section A.10) in the sense that we both use a layered architecture and incorporate a knowledge base in the robot system. However, InteRRap focuses on interaction and cooperation between agents. We focus on more basic issues such as high-level reactivity.

Other than the above, the main difference between InteRRap and our model lies in the high-level control module. The plans used in the high-level control module of InteRRap are hierarchical plans. Therefore, it seems that interleaving the selection and execution of plans, and discrete sensing are inevitable because sensing can only happen at the expansion points in the hierarchical plan. In our model, since prioritized interrupts and threads are used, sensing can be done on a continuous basis.

Regarding continuous sensing, one may wonder about the difference between our model and Nilsson's *Teleo-Reactive Approach* [45] (see section A.7). In the *Teleo-Reactive Approach*, the plans used are essentially sets of condition-action rules structured hierarchically. The whole high-level control module is left idle when it is busy-waiting on signals from the low-level reactive module. Their low-level reactive module monitors the environment on a continuous basis and sends back all perceived information to the high-level control module for deliberation. Therefore, as pointed out by Nilsson himself in [45], some of the information sent back from the reactive module is irrelevant.

The low-level reactive module in our model also monitors the environment on a continuous basis. However, in contrast to the *Teleo-Reactive Approach*, only relevant information is collected and sent back to the high-level control module. The task of collecting and filtering information is done by the lower two layers in our architecture. Since interrupts and threads are used in our high-level control module, there is no busy-waiting. Thus, resource allocation in our model is better than that in Nilsson's.

Previous work on search for localizing people or object such as the SodaJack Project [22] and the agent designed to play "Hide and Seek" in [39], does not fully apply knowledge-based search. As far as we can determine, the knowledge base of the agents is not properly updated and the information obtained from sensors is only used for the current selection of an action. So it may become out-of-date. In our approach, information obtained at runtime is kept in our knowledge base for future use. Decision on the next place to look into is dependent on the contents of our knowledge base. In this respect, our approach is more appropriate for search tasks that involve incomplete knowledge, in which keeping the knowledge base up-to-date and using it to make decisions is critical.

8.4 Future Research Directions

This thesis only deals with controlling a single robot. Future work can extend this to tasks that involve multiple robots, in which the high-level control program should be able to reason about other robots' goals and intentions, and select an appropriate plan in this context.

During the development of the examples, the problem of how to represent uncertain information for perceptual search is encountered. For the sake of clarity, we represented our knowledge as ordered-lists which are then manipulated by functions provided by programmer. However, there are other representations of uncertain information. For example, knowledge can be represented as functions of quantitative probabilities or qualitative probabilities, or as propositional expressions. It is not clear what representation is best or most general for representing uncertain information depending on characteristics of the application. Therefore, future work can focus on working out suitable representation for different types of uncertain information and finding general representation for these.

Since the proper functioning of our system relies on readings from sensors, noisy sensors may pose problems. Therefore, future work may want to take sensor noise into account when modeling a robot and formulating the reasoning process. This can be done using the technique of incorporating subjective probabilities in knowledge [3, 4].

Although we have performed simple reasoning about relations between people and places, nothing has been done on reasoning about relations between these and time. Therefore, future work should explore how spatial and/or temporal reasoning can be used in the high-level control module. Pinto [47] and Reiter [52] has shown how temporal reasoning can be formulated in the Situation Calculus and GOLOG.

In our implementation, interrupts suspend a thread rather than terminate it. If one wants to terminate a thread when an exceptional condition arises, a bunch of test conditions have to be coded in the thread, which may result in code that is difficult to understand because of the excessive branching and nested conditions, and inefficient to execute. Therefore, future work should investigate how an additional throw/catch exception handling mechanism, which terminates the associated block, would help avoid these problems.

Although our program is able to react to runtime failures and exogenous events and uses its knowledge to select what plan to execute, it does little planning. Further work is necessary to test the adequacy of CONGOLOG's support of planning through non-deterministic search for practical robotic applications (see [50] for a discussion of planning in GOLOG).

Lastly, our work on knowledge-based perceptual search is very preliminary. There are still a lot of unsolved hard theoretical problems in this area awaiting for general solutions, for instance, how to reason with incomplete knowledge and perception, and how knowledge can be updated or revised. Therefore, future research work can focus on finding general solutions to these problems and applying them to practical tasks.

Bibliography

[1] Agre, Philip E. and Chapman, David. Pengi: An Implementation of a Theory of Activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 268–272, Seattle, WA, 1987.

[2] Agre, Philip E. and Chapman, David. What Are Plans For? *Robotics and Autonomous Systems*, 6:17–34, 1990.

[3] Bacchus, F., Grove, A.J., Halpern, J.Y., and Koller, D. Statistical foundations for default reasoning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambéry, France, 1993. Morgan Kaufmann Publishing.

[4] Bacchus, F., Halpern, J.Y., and Levesque, H. Reasoning about noisy sensors in the situation calculus. In C.S. Mellish, editor, *Proceeding of IJCAI-95*, pages 1933–1940, Montreal, August 1995.

[5] Baral, C. and Son, T. Regular and special sensing in robot control: relation with action theories. AAAI 97 Workshop on ROBOTS, SOFTBOTS, IMMOBOTS: Theories of Action, Planning and Control, Providence, Rhode Island, July 1997.

[6] Barr, A and Feigenbaum. *The Handbook of Artificial Intelligence.* William Kaufman, Los Altos, CA, 1981.

[7] Bratman, Michael E., Israel, David J., and Pollack, Martha E. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

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

[9] Brooks, R.A. Elephants Don't Play Chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[10] Brooks, R.A. Intelligence Without Reason. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991.

[11] Cowperthwaite, D., Evans, A., French, Lori, Jenkin, M., Luu, T., and Wong, H. *Navigator v3.0A*.

[12] De Giacomo, G., Reiter, R., and Soutchanski, M. Execution monitoring of high-level robot programs. *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, 1998. To appear.

[13] De Giacomo, Giuseppe, Lespérance, Yves, and Levesque, Hector J. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of IJCAI-97*, pages 1221–1226, Nagoya, Japan, August 1997.

[14] De Giacomo, Giuseppe and Levesque, Hector. An incremental interpreter for high-level programs with sensing. Technical report, Department of Computer Science, University of Toronto, 1998.

[15] Ferguson, Innes A. TouringMachines: Autonomous Agents with Attitudes. Technical Report 250, Computer Laboratory, University of Cambridge, New Museums Site, Cambridge CB2 3QG, England, UK, April 1992.

[16] Firby, R. James. Architecture, Representation and Integration: An Example from Robot Navigation. In *Proceedings of the 1994 AAAI Fall Symposium Series Workshop on the Control of the Physical World by Intelligent Agents*, New Orleans, LA, November 1994.

[17] Firby, R. James. Task Networks for Controlling Continuous Processes. In *Proceedings of the Second International Conference on AI Planning Systems*, Chicago, IL, June 1994.

[18] Firby, R. James. Modularity Issues in Reactive Planning. In Drabble, Brian, editor, *Proceedings of the Third International Conference on AI Planning Systems (AIPS-96)*, pages 78–85, Edinburgh, Scotland, May 1996. AAAI Press.

[19] Fisher, M. A survey of Concurrent METATEM – the language and its applications. In Gabbay, D.M. and Ohlback, H.J., editors, *Temporal Logic – Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag, July 1994.

[20] Gat, Erann. Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 809–815, San Jose, CA, July 1992. Morgan-Kaufmann.

[21] Geib, Christopher. Intentions in means-end planning. Technical Report MS-CIS-92-73, Department of Computer and Information Science, University of Pennsylvania, 1992.

[22] Geib, Christopher, Levison, Libby, and Moore, Michael B. SodaJack: an architecture for agents that search for and manipulate objects. Technical Report MS-CIS-94-16/LINC LAB 265, Department of Computer and Information Science, University of Pennsylvania, January 1994.

[23] Georgeff, Michael P. and Lansky, Amy L. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle,WA, 1987.

[24] Green, Cordell C. Theorem Proving by resolution as a basis for question-answering system. *Machine Intelligence*, 4:183–205, 1969.

[25] Haas, Andrew R. The case for domain-specific frame axioms. In Brown, F.M., editor, *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workship*, pages 343–348, Lawrence, KA, April 1987. Morgan Kaufmann Publishers, Inc.

[26] Hexmoor, Henry H. What are routines good for? *Robotics and Autonomous Systems*, 6:75–81, 1990.

[27] Jenkin, M., Bains, N., Bruce, J., Campbell, T., Down, B., Jasiobedzki, P., Jepson, A., Majarais, B., Milios, E., Nickerson, S., Service, J., Terzopoulos, D., Tsotsos, J., and Wilkes, D. Ark: Autonomous mobile robot for an industrial environment. In *IEEE/RSJ IROS*, Munich, 1994.

[28] Kaelbling, Leslie Pack. A Situated-Automata Approach to the Design of Embedded Agents. *SIGART Bulletin*, 2(4):85–88, 1991.

[29] Kaelbling, Leslie Pack and Rosenschein, Stanley J. Action and Planning in Embedded Agents. *Robotics and Autonomous Systems*, 6:35–48, 1990.

[30] Lespérance, Y., Levesque, H., Lin, F., Marcu, D., Reiter, R., and Scherl, R. Foundations of a Logical Approach to Agent Programming. In Wooldridge, M., Müller, J.P., and Tambe, M., editors, *Intelligent Agents Volume II – Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, pages 331–346. Springer-Verlag, 1996. Lecture Notes in Artificial Intelligence.

[31] Lespérance, Y., Levesque, Hector J., and Reiter, R. A Situation Calculus Approach to Modeling and Programming Agents. In preparation.

[32] Lespérance, Y., Levesque, H.J., Lin, F., Marcu, D., Reiter, R., and Scherl, R.B. A Logical Approach to High-Level Robot Programming — A Progress Report. In B. Kuipers, editor, *Control of the Physical World by Intelligent Systems, Papers from the 1994 AAAI Fall Symposium*, pages 109–119, New Orleans, LA, November 1994.

[33] Lespérance, Y., Levesque, H.J., and Ruman, S. An Experiment in Using Golog to Build a Personal Banking Assistant. In Cavedon, L., Rao, A., and Wobcke, W., editors, *Intelligent Agent Systems: Theoretical and Practical Issues (Based on a Workshop Held at PRICAI'96 Cairns, Australia, August 1996)*, pages 27–43. Springer-Verlag, 1997. LNAI volume 1209. To appear.

[34] Lespérance, Y., Tam, K., and Jenkin, M. Reactivity in a Logic-Based Robot Programming Framework. Submitted to the AAAI Fall Symposium on Cognitive Robotics, April 1998.

[35] Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming, 31*, pages 59–84, 1997.

[36] Lin, F. and Reiter, R. How to Progress a Database. *Artificial Intelligence, 92*, pages 131–167, 1997.

[37] Maes, Pattie. The Agent Network Architecture (ANA). *SIGART Bulletin*, 2(4):115–120, 1991.

[38] McCarthy, J. and Hayes, P. Some Philosophical problems from the standpoint of artificial intelligent. *Machine Intelligence*, 4:463–502, 1969.

[39] Moore, Michael B., Geib, Christopher, and Reich, Barry D. Planning for Reactive Behaviors in Hide and Seek. In *Proceeding of 5th Conference on Computer Generated Forces and Behavioral Representation*, Orlando, FL, May 1995. Institute for Simulation and Training.

[40] Moore, Robert C. A formal theory of knowledge and action. In Hobbs, J.R. and Moore, Robert C., editors, *Formal Theories of the Common Sense World*, pages 319–358. Ablex Publishing, Norwood, NJ, 1985.

[41] Müller, Jörg P. and Pischel, Markus. Modeling Interacting Agents in Dynamic Environments. In Cohn, A., editor, *ECAI-94. 11th European Conference on Artifical Intelligence*, pages 709–713. John, Wiley & Sons, Ltd, 1994.

[42] Newell, A. and Simon, H.A. Computer Science as Empirical Enquiry: Symbols and Search. *Communications of the ACM*, 19(3):113–126, March 1976.

[43] Nickerson, S. B., Jenkin, M., Milios, E., Down, B., Jasiobedzki, P., Jepson, A., Terzopoulos, D., Tsotsos, J., Wilkes, D., Bains, N., and Tran, K. Design of ARK, a sensor-based mobile robot for industrial environments. *Intelligent Vehicles*, pages 252–257, July 1993. Tokyo.

[44] Nilsson, N. J. *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.

[45] Nilsson, Nils J. Teleo-Reactive Programs for Agent Control. *Journal of Artificial Intelligence Research, 1*, pages 139–158, 1994.

[46] Pednault, E.P.D. ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus. In Brachman, R.J., Levesque, H., and Reiter, R., editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332, Toronto, ON, May 1989. Morgan Kaufmann Publishers, Inc.

[47] Pinto, J. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, January 1994.

[48] Pohl, I. *Bidirectional and Heuristic Search in Path Problems.* PhD thesis, Department of Computer Science, Stanford University, 1969.

[49] Real World Interface, Inc. *B12 Base Manual, Version 2.4*, 1994.

[50] Reiter, R. KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems. Draft version of January 5, 1998.

[51] Reiter, R. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, Vladimir, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380, San Diego, CA, 1991. Academic Press.

[52] Reiter, R. Sequential, temporal GOLOG. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, 1998. To appear.

[53] Robinson, M. and Jenkin, M. Reactive control of a mobile robot. In Archibald, C. and Kwok, P., editors, *Research in Computer and Robot Vision*, pages 55–70. World Scientific Press, Singapore, 1995.

[54] Rosenschein, Stanley J. and Kaelbling, Leslie Pack. The synthesis of digital machines with provale epistemic properties. In Halpern, Joseph Y, editor, *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 83–98, Monterey, CA, 1986. Morgan Kaufmann Publishers, Inc.

[55] Rusell, S. and Norvig, P. *Artificial Intelligence: A modern Approach.* Prentice Hall, Inc, 1995.

[56] Scherl, Richard B. and Levesque, Hector J. The Frame Problem and Knowledge-Producing Actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 689–695, Washington, DC, July 1993. AAAI Press/The MIT Press.

[57] Schoppers, M. J. Universal Plans for Reactive Robots in Unpredictable Domains. In *Proceedings of IJCAI-87*, San Francisco, CA, 1987. Morgan Kaufmann.

[58] Schubert, Len K. Monotonic solution to the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In Kyberg, H.E., Loui, R.P., and Carlson, G.N., editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, Boston, MA, 1990.

[59] Shoham, Yoav. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[60] Vere, Steven and Bickmore, Timothy. A basic agent. *Computational Intelligence*, 6:41–60, 1990.

[61] Wooldridge, Michael and Jennings, Nicholas R. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2), 1995.

Appendix A

Description of Various Architectures

This appendix describes most of the architectures mentioned in chapter 2.

A.1 IRMA

The *Intelligent Resource-bounded Machine Architecture (IRMA)* [7] addresses the fact that an agent's activities are resource-bounded. These resources include response time, memory, CPU time, stack size, etc. Deliberation to decide what actions to take is performed in a central module which consists of five sub-modules: a *reasoner* for reasoning about the world, a *means-end reasoner* for selecting plans that are consistent with the agent's current beliefs and intentions, an *opportunity analyzer* for exploiting further options for the agent, a *filter sub-module* for determining plans that are consistent with the agent's intentions, and a *deliberation sub-module* to resolve competing choices of plans.

In IRMA, the resource-bound problem is tackled by considering and selecting only those plans/intentions that are consistent with the agent's current beliefs and intentions. Therefore, deliberation is limited to relevant plans. IRMA works with plans that are partial, both temporally and from means-ends point of view[1]. By filtering plans according to the agent's current beliefs and intentions, and by overriding the filter according to pre-coded rules, appropriate plans or actions can be selected to deal with the "current" situation.

A.2 HOMER

HOMER [60] emphasizes the integration of natural language understanding, temporal reasoning and planning, episodic memory and some general world knowledge. The control language uses *state transition semantics*, in which things like actions, goals, and verbs are defined in terms of transitions in world state. Episodic memory plays an important role in allowing the machine to do inference and reflect on its own past experience.

For any given task, the planner will first perform backward and forward chaining to create a dynamic plan which satisfies both the given goals and time constraints. This plan will then be executed and modified as requested. Modification of plan or failure recovery in HOMER is very expensive. The part of the plan that is no longer feasible will first be removed; then the planner will be called to select some new goals through inference, and lastly, a new course of actions is planned to achieve these new goals.

[1] That is the plans contain steps that can be refined later.

A.3 The Subsumption Architecture

Brooks' *subsumption architecture* [8, 9] is considered to be a purely reactive control architecture. The architecture is composed of task-decomposed layers that run completely asynchronously. By inhibiting outputs and suppressing inputs, the upper layers can exercise control over the robot while the lower layers are still monitoring the environment. Each layer has a finite state machine with the ability to hold some data structures. Selection of action is based on the current raw sensor data and the internal state of the finite state machine.

Since everything is hardwired, and perception and actions are tightly coupled, response to environmental changes and runtime failures is very quick. There is no planning unit in this architecture. "Reasoning" is broken down into procedures and merged into the various layers. Therefore, high-level control system is almost impossible in this architecture.

A.4 Pengi

Pengi [1] is a hardwired control system designed to play the computer game Pengo. It is based on the idea that most activities are "routines" in the sense that they require little new abstract reasoning. Routines are neither plans nor procedures: they are the patterns of interaction between an agent and its environment. Once learnt, similar tasks can be done in a similar way, with little variation. Therefore, routines can be hardwired as circuits and updated periodically. In addition, routines are flexible and opportunistic because the agent can switch to different routines or goals in different situations. Since effects propagation (which is the propagation of the effects of performing an action) is relatively slow, it is sufficient to sense the properties of the immediate situation (called "indexical-functional aspects") rather than maintain a world model.

As mentioned in a later paper by Agre and Chapman [2], Pengi does not make nor use plans because it is only engaged in "improvisatory activities". However, some harder tasks do require planning. In these situations, plans shall be used as one of the resources that the agent can draw upon when deciding what actions to take rather than as something that completely determines the agent's action. The executor retains the right to decide whether the "advise" from the planner is appropriate. This is called the "plans-as-communication" view of plans.

A.5 Situated-Automata

The *Situated-Automata* approach [29, 28] stresses the importance of situatedness and goal-directed behaviors. In situated-automata theory, the world is modeled as a pair of interacting automata: one corresponds to the world, and the other corresponds to the agent. The relation between these two automata is established by the following specification:

> "... when the machine is in a state, s, it can be said to carry the information ϕ if and only if whenever the machine is in state s, the proposition ϕ holds in the environment." [28]

The agent is specified by a pair of declarative programming languages, namely Ruler and Gapps. *Ruler* is used to specify the perception component of an agent and *Gapps* is used to specify the action component.

The Ruler program takes as inputs three components: the semantics specifications of the sensors, a set of static facts, and a specification of the state transitions in the world. From these inputs, the compiler synthesizes a circuit that produces the desired, correct semantics. The output of the circuit will be used to update the internal state of the automata. All "declarative 'knowledge' (static facts provided by the programmer) has been reduced to a very simple circuit" [28] by the end of this compilation.

The Gapps program generates a program that can be translated into a digital circuit from a set of goal-reduction rules and a top-level goal. The generated program is a finite state-free program that maps the current perceptual state into actions.

Since the whole system is compiled down into a digital circuit, all decisions and responses are made in a timely fashion. In addition, by taking raw sensor data as inputs to the circuit, and by monitoring and evaluating the validity of the current action on each tick of the system, the agent can switch subgoals and actions in response to environmental changes promptly.

A.6 The Agent Network Architecture (ANA)

In the *Agent Network Architecture (ANA)* [37], the system is composed of groups of *competence modules* arranged in a network (somewhat like a neural network). There are two types of competence modules. The *action modules* make the robot to commit to some actions and the *belief modules* make the robot to adopt some beliefs. Each module has an *activation level*, which is a real number. The activation level of a module x is a function of the agent's top-level goals, its current protected goals, the current raw sensor data, and the executability of the predecessors or successors of x in the network. This number not only indicates the relevance of module x to the current situation, but also determines whether x has the right to exercise control over the robot.

As noted in [37], plans in ANA are used to express the agent's "intention" or "urge" to take a certain action. These "intentions" are indirectly determined by the activation levels. "Reasoning" is performed through the computation, assignment and selection of those activation numbers. By taking the current perceptual data and goals into account in the calculation of those activation levels and by spreading the activation forward, the agent is claimed to have "situation relevance and be opportunistic" [37].

A.7 The Teleo-Reactive Approach

The *Teleo-Reactive approach* [45] integrates the concept of continuous feedback in control theory with some ideas from computer science such as binding parameters at runtime, subroutines and recursion. The agent's control program is a teleo-reactive sequence which is actually a set of condition-action rules with variables. The sequence is claimed to be "universal" (see section 2.2.4) because it satisfies both the property of *goal-regression* and the requirement of *completeness*. The program can be recursive and hierarchically structured. Multiple actions can be triggered if more than one action are specified in the action part of the rule. The teleo-reactive program can be compiled down into a digital circuit prior to execution, or implemented using standard techniques such as writing it up in *LISP cond* statements and interpreting it at run time.

The *Teleo-reactive approach* deals with unanticipated events by monitoring the environment continuously. In addition, if the program is compiled down into a digital circuit, the action to perform can be computed quickly by the circuit and a prompt reaction to events is possible. However, the approach suffers from the fact that it may not discriminate irrelevant sensory data from relevant data.

A.8 The Procedural Reasoning System (PRS)

The *Procedural Reasoning System (PRS)* [23] combines reactivity with reasoning and planning at a high level. The *PRS* takes as input a set of beliefs (or facts from the agent's database) and a set of goals (or desires) for the agent. It then selects a relevant *Knowledge Area (KA)* from a *KA* library. *KA* is a plan which is both partial and hierarchical. It is an expression of the form condition-body, where body is a sequence of subgoals to be pursued to bring the agent to the goal(s) and condition is the situation in which the body shall be triggered. The sequence of subgoals in the body part constitutes a plan. Any goal in the system, unlike that in traditional AI, is a desired behavior such as 'walk' or 'pick' and the agent strives to make that behavior possible. In the course of executing the chosen *KA*, new beliefs may be derived. This may result in a situation where another *KA* is more relevant than the current one. In this case, the new *KA* will be chosen and executed. Therefore, the plan being executed is always relevant to the current situation. Also, new plans can be expanded from the old ones as the system operates.

By interleaving plan selection, formation, and execution, partial and hierarchical plans can be built. These are partial in the sense that in the pursuance of an end, the means to this end are left open for future deliberation. This allows switching goals or subgoals during the course of execution. Inferences, for example, deduction of new beliefs, can be made as the system operates but control is never relinquished to a separate reasoning module that may perform unbounded amounts of computation. In this way the system is resource-bounded.

A.9 TouringMachines

TouringMachines [15] is a layered agent architecture that is resource-bounded and goal-directed, and is capable of making hypotheses about such things as locations, goals, and intentions of other agents, while maintaining some geographical knowledge about the world. It uses partial and hierarchical plans to solve the resource-bound problem. It is reactive at both the low and high level: at the low level, the robot is made reactive by having a tight coupling of perception and action; at the high level, planning is made reactive by doing projection based on the agent's internal cognitive model.

The agent has three layers. Each is equipped with different task-oriented capabilities and direct connections from perception to action. Decisions on what action to take are deliberated upon independently by each individual layer and conflicting choices of actions are mediated by an enveloping control framework. This control framework allows a layer to examine, inject or remove data from other layers. With this mechanism, suppression of inputs and inhibition of outputs can be achieved and conflicts can be resolved.

The bottom layer of the agent is the *reactive* layer which provides the agent with a set of condition-action rules for performing time-critical tasks such as obstacle avoidance.

The middle layer is a *planner* that works with partial and hierarchical plans. The design of this layer is based on Bratman *et al.*'s view of limiting the amount of future deliberation by maintaining consistency across intentions (see section A.1). In addition, commitment to sub-plan execution is deferred until it is necessary and this allows switching of subgoals or actions during the course of execution. Similarly to the *PRS* (see section A.8), the planner instantiates template plans (or *schemata*) and retrieves relevant sub-plans from a *schema library* at expansion points according to the current condition.

The highest layer is the *modeling* layer that gives the agent reflective and predictive capabilities. This layer builds and modifies the agent's cognitive model of the world at runtime. This model is used to predict potential conflicts in goals with other agents and initiate re-planning.

A.10 InteRRaP

InteRRaP [41] is a model addressing situatedness, goal-orientation, interaction, and cooperation. It is layered, and all layers run in parallel. Goal-directed actions are produced by interleaving the selection and execution of plans.

The model is divided into a set of competence-driven functional layers. The *world interface (WIF)* models all the agent's low-level tasks such as perception, action, and communication. The *behavior-based component (BBC)* models the agent's control mechanism which is based on a set of dynamically prioritized patterns of behavior. Patterns of behavior cover the agent's basic reactive and procedural functionalities such as modifying the internal knowledge base and controlling the movement of the robot. The *cooperation component (CC)* models cooperation between agents and provides a mechanism for communication and cooperation. Planning is modeled by the *plan-based component (PBC)* which, upon request from the *BBC* or *CC*, devises a hierarchical plan to attain some specified goals. The hierarchical plan may be expanded into sub-plans according to some heuristic evaluation and the current perceptual information obtained from the *BBC*. Other than devising and expanding a plan, the *PBC* also determines the agent's role in a joint plan when multiple agents are involved in a task.

The InteRRap also contains a knowledge base which is structured into layers. There is a one-to-one relation between a layer in the knowledge base and a layer amongst the competence-driven functional layers described so far. In the knowledge base, we can find such things as the agent's internal world model, the patterns of behavior, local plans or goals, and strategies for cooperation. Lastly, control in the InteRRap flows in a top-down manner while perceptual data flows bottom-up.

A.11 The Reactive Action Package (RAP)

The *Reactive Action Package (RAP)* [17, 16] is an architecture that features hierarchical expansion of plans which allows an agent to achieve a goal along different paths of execution, and a highly modular structure so that modules can be used in the execution of many different tasks [18].

The system maintains a RAP library and a world model. A *RAP* is essentially "an autonomous process" that drives the agent to a particular goal. A RAP can be viewed as a set of hierarchical plans. Each expansion point in a hierarchical plan is either a primitive action or subgoal. This set of plans specifies how a goal can be achieved in different situations.

At the beginning of execution, a task is chosen for execution and a RAP is selected for this goal. This RAP will then be hierarchically expanded into actions, which will be executed by the reactive layer, or subgoals, which will further expand into other RAPs. All selected RAPs are placed in a queue. Whenever the goal of the current RAP is attained, the RAP will be removed and the next RAP will start execution. Sensor data are read at the beginning of every cycle through the hierarchical plans in a RAP. The current perceptual data are used to determine whether the goal of the current RAP has been attained, as well as which hierarchical plan shall be expanded.

Since the world model is assumed to be complete and RAPs are assumed to be universal, everything should be within the agent's "expectations" (otherwise, the system will fail [21]). Therefore, the top-level goal will be achieved by executing the planned RAPs in sequence. However, the system does "not do any significant look-ahead" [16], i.e. it takes decisions without considering future states. As a result, the system does not guarantee that the current goal can be achieved once a RAP has started execution. Also, as mentioned by Firby himself, this may cause problems in situations where conflicting goals or actions are involved [16].

A.12 ATLANTIS

ATLANTIS [20] integrates a traditional AI planner and a reactive control mechanism using Agre and Chapman's "plans-as-communication" view of plans (see section A.4). The resulting architecture exhibits goal-directedness and reactive behavior. In addition, it is also claimed to be "taskable", i.e. the resulting agent can be programmed to handle different tasks.

The system consists of three components. The *controller* is responsible for reactive sensorimotor actions and those that require little or no deliberation. It is implemented by embedding a classical control theory within a carefully designed computational framework in such a way that the control algorithm can be engineered effectively.

The *sequencer*, which is modeled after Firby's RAP system (see section A.11) is responsible for controlling the sequence of actions performed or does so by initiating and terminating appropriate modules in the controller. In addition, the sequencer is also responsible for resource allocation during execution. With the help of an algorithm to detect or predict failures, the system is able to deal with failures (sometimes in advance), say, by re-planning.

The *deliberator* is the third component where planning and world model updating actually take place. This component contains a traditional AI planning system. It is called and activated by the sequencer and the output of this component is a plan. This plan will be treated by the sequencer as a resource to draw upon when determining what action to take. By putting deliberation outside the control loop, the system can react to unanticipated events in a timely manner.

Appendix B

Program Listing: Runtime Failure and Exogenous Event Handling

This is the complete ConGolog program listing of the example given in chapter 4 about runtime failures and exogenous events handling.

```
/*************************************************************************
**                                                                     **
**                    Robot Mail Delivery Example                      **
**                 ConGolog version (For Simulation)                   **
**                                                                     **
**                    Implemented  by  Kenneth Tam                     **
**                           July 18, 1997                             **
**                                                                     **
*************************************************************************
*************************************************************************
**                                                                     **
**    This is the controlling program for a robot mail delivery        **
**    system in an office setting.  The program is designed to be      **
**    highly responsive to exogenous events and runtime failures       **
**                                                                     **
*************************************************************************
*************************************************************************
**                                                                     **
**                              Updates                                **
**                                                                     **
**    Aug 28, 97: The controller can start up without any shipment     **
**                order being placed initially.                        **
```

```
**                                                                      **
**    Feb 23, 98: Minor Bugs fixed.                                     **
**                                                                      **
**************************************************************************
**************************************************************************
**                                                                      **
**    Primitive Actions:                                                **
**                                                                      **
**      For Normal Operations and Runtime Failures Handling             **
**         startGoTo(Place)                                             **
**         ackShipmentOrder(ShNo,Sender,Recipient)                      **
**         pickUpShipment(ShNo)                                         **
**         dropOffShipment(ShNo)                                        **
**         shutdownRobot                                                **
**         cancelPickUp(ShNo,Sender)                                    **
**         reduceCredit(Person)                                         **
**         resetRobot                                                   **
**         notifyStopServing(Person)                                    **
**         ackWithRegret(ShNo,Sender,Recipient)                         **
**                                                                      **
**    Exogenous Actions:                                                **
**         orderShipment(Sender,Recipient,Priority)                     **
**         msgFromRobot(robotStatus)                                    **
**                                                                      **
**    **************************************************************    **
**                                                                      **
**    Predicate Fluents:                                                **
**                                                                      **
**      For Normal Operations and Runtime Failures Handling             **
**         shipmentOrdToAck(ShNo)                                       **
**         shipmentToPickUp(ShNo)                                       **
**         shipmentOnBoard(ShNo)                                        **
**         shipmentDelivered(ShNo)                                      **
**         clientSuspended(Person)                                      **
**         readyToBeServed                                              **
**         robotAlive                                                   **
**                                                                      **
**    **************************************************************    **
**                                                                      **
**    Functional Fluents:                                               **
**                                                                      **
**      For Normal Operations and Runtime Failures Handling             **
**         sender(ShNo)                                                 **
**         recipient(ShNo)                                              **
**         priority(ShNo)                                               **
**         shipmentCtr                                                  **
```

```
**         robotStatus                                                  **
**         robotPlace                                                   **
**         haveCredit(Person)                                           **
**         approaching                                                  **
**                                                                      **
***************************************************************************/



/* ************************************************************************
**                   Primitive Actions Declarations                      *
** ************************************************************************/

/* Primitive Actions for normal operations and handling runtime failures */
primAct(startGoTo(_)).
primAct(ackShipmentOrder(_,_,_)).
primAct(pickUpShipment(_)).
primAct(dropOffShipment(_)).
primAct(shutdownRobot).
primAct(cancelPickUp(_,_)).
primAct(reduceCredit(_)).      /* Reduce one credit from a client           */
primAct(resetRobot).           /* Halt and Reset robot                      */
primAct(notifyStopServing(_)). /* Stop serving a person                     */
primAct(ackWithRegret(_,_,_)). /* Acknowledge an invalid shipment order     */
                               /*   the shipment won't be served            */


/* Exogenous Actions Declarations */
primAct(orderShipment(_,_,_)).
primAct(msgFromRobot(_)).      /* simulating feedbacks from robot */



/* ************************************************************************
**                    Non-Fluent Declarations                            *
** ************************************************************************/

/* Non-Fluent Declarations */
non_fluent(person(_)).



/* ************************************************************************
**                    Action Precondition Axioms                         *
** ************************************************************************/

```

```
/* Primitive Actions for normal operations and handling runtime failures */

/* For Primitive Actions */
poss(startGoTo(Place),S):-
   legalPlace(Place),
   holds(val(robotStatus,idle), S).

poss(ackShipmentOrder(ShNo,Sndr,Rcpt),S):-
   holds(shipmentOrdToAck(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(recipient(ShNo),Rcpt),S).

poss(pickUpShipment(ShNo),S):-
   holds(shipmentToPickUp(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(robotPlace,mailBox(Sndr)),S).

poss(dropOffShipment(ShNo),S):-
   holds(shipmentOnBoard(ShNo),S),
   ( (holds(val(recipient(ShNo),Person),S);
        holds(val(sender(ShNo),Person),S)),
      holds(val(robotPlace,mailBox(Person)),S) );
   holds(val(robotPlace, centralOffice),S).

poss(shutdownRobot,_).

poss(cancelPickUp(ShNo,Sender),S) :-
   holds(shipmentToPickUp(ShNo),S), person(Sender).

poss(reduceCredit(Person),S) :-
   holds(val(haveCredit(Person),Credits),S),
   Credits > 0.

poss(resetRobot,S) :-
   not(holds(val(robotStatus,idle),S)).

poss(notifyStopServing(P),_) :-
   person(P).

poss(ackWithRegret(ShNo,Sndr,Rcpt),S):-
   holds(shipmentOrdToAck(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(recipient(ShNo),Rcpt),S).


/* For Exogenous Actions */
```

```

/* Priority=1 means high priority, Priority=0 means low priority */
poss(orderShipment(Sender,Recipient,Priority),S) :-
   holds(robotAlive,S),
   person(Sender),
   person(Recipient),
   Priority < 2.

poss(msgFromRobot(_),S) :-
   holds(robotAlive,S),
   holds(val(robotStatus,moving),S).



/* ************************************************************************
**                      Successor State Axioms                            *
** ************************************************************************/

/* For normal operations and handling runtime failures */
holds(shipmentOrdToAck(ShNo),do(A,S)):-
   (A = orderShipment(_,_,_), holds(val(shipmentCtr,ShNo),S));
   (holds(shipmentOrdToAck(ShNo),S),
    A \= ackShipmentOrder(ShNo,_,_), A \= ackWithRegret(ShNo,_,_)).

holds(shipmentToPickUp(ShNo),do(A,S)):-
   A = ackShipmentOrder(ShNo,_,_);
   (holds(shipmentToPickUp(ShNo),S),
   (A \= pickUpShipment(ShNo), A \= cancelPickUp(ShNo,_))).

holds(shipmentOnBoard(ShNo),do(A,S)):-
   A = pickUpShipment(ShNo);
   ( holds(shipmentOnBoard(ShNo),S),
     A \= dropOffShipment(ShNo) ).

holds(shipmentDelivered(ShNo),do(A,S)):-
   A = dropOffShipment(ShNo);
   A = ackWithRegret(ShNo,_,_);
   A = cancelPickUp(ShNo,_);
   holds(shipmentDelivered(ShNo),S).

/* Hold iff a client is no longer being served                          */
/* To make the action "cancelPickUp" executes concurrently with         */
/* robotMotionControl.  We need this as a sucessor state axiom as       */
/* well as haveCredit().                                                */
holds(clientSuspended(Cust),do(A,S)) :-
   A = notifyStopServing(Cust);
```

```
   holds(clientSuspended(Cust),S).

holds(readyToBeServed,do(A,S)) :-
   A = orderShipment(_,_,_);
   holds(readyToBeServed,S).

/* Robot is alive if it has not been shut down */
holds(robotAlive,do(A,S)) :-
   holds(robotAlive,S), A \= shutdownRobot.

holds(val(sender(ShNo),Sender),do(A,S)):-
   (A = orderShipment(Sender,_,_), holds(val(shipmentCtr,ShNo),S));
   holds(val(sender(ShNo),Sender),S).

holds(val(recipient(ShNo),Rec),do(A,S)):-
   (A = orderShipment(_,Rec,_), holds(val(shipmentCtr,ShNo),S));
   holds(val(recipient(ShNo),Rec),S).

/* Returns the priority of a shipment */
holds(val(priority(ShNo),Priority),do(A,S)):-
   (A = orderShipment(_,_,Priority), holds(val(shipmentCtr,ShNo),S));
   holds(val(priority(ShNo),Priority),S).

holds(val(shipmentCtr,N),do(A,S)):-
   (A = orderShipment(_,_,_), !,
       holds(val(shipmentCtr,M),S), N is M + 1);
   holds(val(shipmentCtr,N),S).

/* Returns the current status of robot */
holds(val(robotStatus,Status), do(A,S)):-
   (A = startGoTo(_), !, Status = moving);
   ( (A = resetRobot; A = msgFromRobot(reached)), !, Status = idle );
   (A = msgFromRobot(blocked), !, Status = blocked);
   holds(val(robotStatus,Status),S).

/* Returns the current position of the robot when not moving */
/* and returns unknown when the robot is moving              */
holds(val(robotPlace,Place),do(A,S)):-
   (A = msgFromRobot(reached), holds(val(approaching,Place1),S), !,
    Place = Place1);
   (A = startGoTo(_), !, Place = unknown);
   holds(val(robotPlace,Place),S).

/* Returns the number of credits a person left.         */
/* Credits of a person will decrease by 1 if the robot  */
/* is blocked on the way to the person's mailbox.       */
```

```
holds(val(haveCredit(Person),Credits),do(A,S)):-
   (A = reduceCredit(Person), !,
    holds(val(haveCredit(Person),CreditsLeft),S),
    Credits is CreditsLeft - 1);
   holds(val(haveCredit(Person),Credits),S).

/* Returns the place to which the robot is heading              */
/* Value changes when the agent starts moving towards another's   */
/* mailBox                                                         */
holds(val(approaching,Place), do(A,S)) :-
   (A = startGoTo(Place1), !, Place = Place1);
   holds(val(approaching,Place),S).



/* ***********************************************************************
**                            Defined Fluents                           *
** ***********************************************************************/

/* For normal operations and handling runtime failures */

/* Holds if Cust requires the services provided by the robot */
holds(needServing(Cust),S) :-
   ( holds(shipmentToPickUp(ShNo),S),
     holds(val(sender(ShNo),Cust),S),
     holds(shipmentDeliverable(ShNo),S),
     holds(shipmentReturnable(ShNo),S) );
   ( holds(shipmentOnBoard(ShNo),S),
     ( holds(val(recipient(ShNo),Cust),S);
       ( holds(val(sender(ShNo),Cust),S),
         not(holds(shipmentDeliverable(ShNo),S))) ),
       not(holds(clientSuspended(Cust),S)) ).

/* Returns 1 if Cust has some higher priority shipments, else returns 0 */
holds(val(haveShptOfPriority(Cust),P),S) :-
   ( holds(val(priority(ShNo),1),S),
     ((holds(shipmentToPickUp(ShNo),S), holds(val(sender(ShNo),Cust),S));
      (holds(shipmentOnBoard(ShNo),S), holds(val(recipient(ShNo),Cust),S))),
     !, P=1 );
   P=0.

/* Holds if Cust1 is the one with the largest amount of credits left */
/* among all those clients, including Cust1, who have some shipments */
/* of priority P                                                      */
holds(haveHighestCredit(Cust1,P),S) :-
   ( holds(val(haveCredit(Cust1),C1),S), holds(needServing(Cust2),S),
```

```
     Cust1 \= Cust2, holds(val(haveShptOfPriority(Cust2),P),S),
     holds(val(haveCredit(Cust2),C2),S), C2 > C1,
     !, not(holds(true,S)) );
   holds(true,S).

/* Holds if P1 is the highest priority amongst all available shipments */
holds(isTheHighestPriority(P1),S) :-
   ( holds(needServing(Cust),S),
     holds(val(haveShptOfPriority(Cust),P2),S),
     P2 > P1, !, not(holds(true,S)) );
   holds(true,S).

/* The best customer to serve is the one with highest priority and largest */
/* amount of credit left amongst those who are of the same priority level  */
holds(bestCustToServe(Cust),S) :-
   holds(needServing(Cust),S),
   holds(val(haveShptOfPriority(Cust),Priority),S),
   holds(isTheHighestPriority(Priority),S),
   holds(haveHighestCredit(Cust,Priority),S).

/* Holds if the robot is serving some higher priority shipment */
holds(servingPriorityShpt,S) :-
   holds(val(approaching,mailBox(Cust)),S),
   ( (holds(shipmentToPickUp(ShNo),S), holds(val(sender(ShNo),Cust),S));
     (holds(shipmentOnBoard(ShNo),S), holds(val(recipient(ShNo),Cust),S)) ),
   holds(val(priority(ShNo),1),S).

/* A shipment is returnable if its sender has not been suspended */
holds(shipmentReturnable(ShNo),S) :-
   holds(val(sender(ShNo),Cust),S), not(holds(clientSuspended(Cust),S)).

/* A shipment is deliverable if its recipient has not been suspended */
holds(shipmentDeliverable(ShNo),S) :-
   holds(val(recipient(ShNo),Cust),S), not(holds(clientSuspended(Cust),S)).



/* ***********************************************************************
**                         Non-fluents values                           *
** ***********************************************************************/

legalPlace([X,Y,T]):- number(X), number(Y), number(T), !.
legalPlace(P):- placeValue(P,_).

placeValue([X,Y,T],[X,Y,T]):- number(X), number(Y), number(T), !.
placeValue(mailBox(yves),[10.000000, 33.430626,0]).
```

```
placeValue(mailBox(michael),[11.273750, 31.120001,0]).
placeValue(mailBox(jeff),[8.583750, 30.780001,0]).
placeValue(mailBox(hector),[8.393750, 33.230000,0]).
placeValue(robotHome,[9.437200, 32.434600,0]).
placeValue(centralOffice,[10.490000, 29.958750,0]).

person(yves).
person(michael).
person(jeff).
person(hector).



/* ************************************************************************
**                        Initial States                                  *
** ***********************************************************************/

holds(val(robotPlace,Loc),s0) :- Loc = robotHome.
holds(val(shipmentCtr,N),s0) :- N = 1.
holds(val(robotStatus, Status),s0) :- Status = idle.
holds(robotAlive,s0).

/* Change the following to obtain the maximum number of blocking allowed */
holds(val(haveCredit(Person),C),s0):- person(Person),C=2.

holds(val(approaching,Place), s0) :- Place = nowhere.



/* ************************************************************************
**                        Exogenous Actions                               *
**                     Use for simulation only                            *
** ***********************************************************************/

/* This version relies on the "feedback" from robot         */
/* So, the predicate msgFromRobot() cannot be commented out */
exoDensity(100).

/* Setting for Test 1 */

exoAct(orderShipment(yves,jeff,1),1,2).
exoAct(orderShipment(jeff,michael,0),3,4).
exoAct(orderShipment(michael,hector,0),5,6).
exoAct(msgFromRobot(blocked),7,47).
exoAct(msgFromRobot(reached),48,88).

```

```

/* Setting for Test 2 */
/*
exoAct(orderShipment(yves,jeff,1),1,2).
exoAct(orderShipment(jeff,daniel,0),3,4).
exoAct(orderShipment(michael,hector,0),5,6).
exoAct(msgFromRobot(blocked),7,77).
exoAct(msgFromRobot(reached),78,88).
*/



/* ***********************************************************************
**                            Tracing Controls                           *
** ***********************************************************************/

/*
tracingProg.
tracingTest.
tracingExec.
*/



/* ***********************************************************************
**                         Procedure Definitions                         *
** ***********************************************************************/

/* Acknowledges a shipment if there are any*/
proc(acknowledgeShipmentOrder(ShNo),
   pi([sndr,rcpt], [ and(val(sender(ShNo),sndr),val(recipient(ShNo),rcpt))?,
        if(and(shipmentReturnable(ShNo),shipmentDeliverable(ShNo)),
            [ ackShipmentOrder(ShNo,sndr,rcpt),
              if(and(val(priority(ShNo),1),
                     and(val(robotStatus,moving),not(servingPriorityShpt))),
                    resetRobot,
                    no_op
              ) ],
            ackWithRegret(ShNo,sndr,rcpt)
        )
   ] )
).

/* Top level control of the robot: Serves a client if there are any,   */
/* otherwise, goes to central office and drops off all remaining       */
/* shipment onboard.  And shuts down the robot.                        */
```

```
proc(robotMotionControl,
   if(some(cust, needServing(cust)),
         serveCustomer,
         [ startGoTo(centralOffice),
           not(val(robotStatus,moving))?,
           if(val(robotPlace,centralOffice),
                 [ dropOffToCentralOffice,
                   if(some(cust2, needServing(cust2)),
                      no_op,
                      shutdownRobot)
                 ],
                 if(val(robotStatus,blocked),
                       resetRobot,
                       no_op
                 )
           ) ]
   )
).

/* Defines the procedure of what to do when a customer who requires */
/* serving is identified: Provides service if arrived at the        */
/* destination, otherwise, calls procedure ‘handleGoToFailure’      */
proc(serveCustomer,
   pi(cust, [ bestCustToServe(cust)?,
         [ startGoTo(mailBox(cust)),
           not(val(robotStatus,moving))?,
           if(val(robotPlace,mailBox(cust)),
                 [ pickUpShipmentsFrom(cust),
                   dropOffShipmentsTo(cust) ],
                 if(val(robotStatus,blocked),
                       handleGoToFailure(cust),
                       no_op
                 )
           ) ]
   ] )
).

/* Defines the procedure of what to do when blocked */
proc(handleGoToFailure(Cust),
   [ reduceCredit(Cust),
     if(val(haveCredit(Cust),0),
          notifyStopServing(Cust),
          no_op),
     resetRobot ]
).
```

```
/* Drops off all onboard invalid shipments to central office */
proc(dropOffToCentralOffice,
   while(some(shNo, and(shipmentOnBoard(shNo),
                        not(or(shipmentDeliverable(shNo),
                               shipmentReturnable(shNo))))),
       pi(shNo, [and(shipmentOnBoard(shNo),
                     not(or(shipmentDeliverable(shNo),
                            shipmentReturnable(shNo))))?,
          dropOffShipment(shNo) ]))).


/* Picks up all shipments from Cust */
proc(pickUpShipmentsFrom(Cust),
   while(some(shNo, and(shipmentToPickUp(shNo),val(sender(shNo),Cust))),
      pi(shNo, [ and(shipmentToPickUp(shNo),val(sender(shNo),Cust))?,
         pickUpShipment(shNo) ]))).


/* Drops off all shipments to Cust */
proc(dropOffShipmentsTo(Cust),
   while(some(shNo, and(shipmentOnBoard(shNo),
                        or(val(recipient(shNo),Cust),
                           and(val(sender(shNo),Cust),
                               not(shipmentDeliverable(shNo)))))),
      pi(shNo, [ and(shipmentOnBoard(shNo),
                     or(val(recipient(shNo),Cust),
                        and(val(sender(shNo),Cust),
                            not(shipmentDeliverable(shNo)))))?,
         dropOffShipment(shNo) ]))).



/* ************************************************************************
**                              Main Program                             *
** ***********************************************************************/

runSystem(Ini,Final) :-
   initialize(1),
   pdo(

   /* Acknowledges shipment order if there are any */
   (==>([shNo], shipmentOrdToAck(shNo),
           acknowledgeShipmentOrder(shNo)))

   #=
   /* Cancels pick up if there are any invalid shipment waiting for pickup */
```

```
   (==>([shNo,sender], and(and(shipmentToPickUp(shNo),
                               val(sender(shNo),sender)),
                           not(and(shipmentReturnable(shNo),
                               shipmentDeliverable(shNo)))),
           cancelPickUp(shNo,sender) ))

   #>
   /* Main Loop to serve a client */
   (==>([shNo], and(readyToBeServed,
                    or(or(shipmentToPickUp(shNo), shipmentOnBoard(shNo)),
                       not(val(robotPlace,centralOffice)))),
           robotMotionControl))

   #>
   /* Skips if the robot is moving */
   or(not(readyToBeServed), robotAlive) ==> no_op,

   Ini,Final).



/* ************************************************************************
**        An Auxiliary Program to break the final state down into        *
**         a sequence of actions starting from the initial state         *
** ***********************************************************************/

output(do(A,s0)) :- format('~N~w',A).
output(do(A,S)) :-  output(S), format('~N~w',A), nl.

run2 :- runSystem(s0,Final), output(Final).
```

Appendix C

The Complete Set of Axioms: Runtime Failure and Exogenous Event Handling

This is the complete listing of all the axioms (in Situation Calculus notation) of the mail delivery example given in chapter 4 about runtime failure and exogenous event handling.

C.1 Action Precondition Axioms

$$Poss(startGoTo(l), s) \equiv legalPlace(l) \wedge robotStatus(s) = idle$$

$$\begin{aligned}&Poss(ackShipmentOrder(n, c, r), s) \equiv shipmentOrdToAck(n, s) \wedge \\ &\quad sender(n, s) = c \wedge recipient(n, s) = r\end{aligned}$$

$$\begin{aligned}&Poss(pickUpShipment(n), s) \equiv (\exists c) .shipmentToPickUp(n, s) \wedge \\ &\quad sender(n, s) = c \wedge robotPlace(s) = mailBox(c)\end{aligned}$$

$$\begin{aligned}&Poss(dropOffShipment(n), s) \equiv (\exists c) .shipmentOnBoard(n, s) \wedge \\ &\quad ([(sender(n, s) = c \vee recipient(n, s) = c) \wedge \\ &\quad\quad robotPlace(s) = mailBox(c)] \vee robotPlace(s) = centralOffice)\end{aligned}$$

$$Poss(shutdownRobot, s) \equiv TRUE$$

$$Poss(cancelPickUp(n, c), s) \equiv shipmentToPickUp(n, s) \wedge person(c)$$

$$Poss(reduceCredit(c), s) \equiv haveCredit(c, s) > 0$$

$$Poss(resetRobot, s) \equiv robotStatus(s) \neq idle$$

$$Poss(notifyStopServing(c), s) \equiv person(c)$$

$$Poss(ackWithRegret(n, c, r), s) \equiv shipmentOrdToAck(n, s) \wedge \\ sender(n, s) = c \wedge recipient(n, s) = r$$

$$Poss(orderShipment(c, r, p), s) \equiv robotAlive(s) \wedge person(c) \wedge \\ person(r) \wedge (p = 0 \vee p = 1)$$

$$Poss(msgFromRobot(i), s) \equiv \\ robotAlive(s) \wedge robotStatus(s) = moving$$

C.2 Successor State Axioms

$$shipmentOrdToAck(n, do(a, s)) \equiv \\ (\exists c, r, p)\,[\,a = orderShipment(c, r, p) \wedge shipmentCtr(s) = n\,] \vee \\ [\,shipmentOrdToAck(n, s) \wedge (\forall c, r)(a \neq ackShipmentOrder(n, c, r) \wedge \\ a \neq ackWithRegret(n, c, r))\,]$$

$$shipmentToPickUp(n, do(a, s)) \equiv \\ (\exists c, r)\, a = ackShipmentOrder(n, c, r) \vee \\ [\,shipmentToPickUp(n, s) \wedge a \neq pickUpShipment(n) \wedge \\ (\forall c)\, a \neq cancelPickUp(n, c)\,]$$

$$shipmentOnBoard(n, do(a, s)) \equiv \\ a = pickUpShipment(n) \vee \\ [\,shipmentOnBoard(n, s) \wedge a \neq dropOffShipment(n)\,]$$

$$
\begin{aligned}
&shipmentDelivered(n, do(a,s)) \;\equiv \\
&\quad a = dropOffShipment(n) \lor \\
&\quad (\exists\, c, r)\, a = ackWithRegret(n,c,r) \;\lor\; (\exists\, c)\, a = cancelPickUp(n,c) \lor \\
&\quad shipmentDelivered(n,s)
\end{aligned}
$$

$$
\begin{aligned}
&clientSuspended(c, do(a,s)) \;\equiv \\
&\quad a = notifyStopServing(c) \lor \\
&\quad clientSuspended(c,s)
\end{aligned}
$$

$$
\begin{aligned}
&readyToBeServed(do(a,s)) \;\equiv \\
&\quad (\exists\, c, r, p)\; a = orderShipment(c,r,p) \lor \\
&\quad readyToBeServed(s)
\end{aligned}
$$

$$
\begin{aligned}
&robotAlive(do(a,s)) \;\equiv \\
&\quad robotAlive(s) \;\lor\; a \neq shutdownRobot
\end{aligned}
$$

$$
\begin{aligned}
&sender(n, do(a,s)) = c \;\equiv \\
&\quad [\,(\exists\, r, p)\; a = orderShipment(c,r,p) \;\land\; shipmentCtr(s) = n\,] \lor \\
&\quad c = sender(n,s)
\end{aligned}
$$

$$
\begin{aligned}
&recipient(n, do(a,s)) = r \;\equiv \\
&\quad [\,(\exists\, c, p)\; a = orderShipment(c,r,p) \;\land\; shipmentCtr(s) = n\,] \lor \\
&\quad r = recipient(n,s)
\end{aligned}
$$

$$
\begin{aligned}
&priority(n, do(a,s)) = p \;\equiv \\
&\quad [\,(\exists\, c, r)\; a = orderShipment(c,r,p) \;\land\; shipmentCtr(s) = n\,] \lor \\
&\quad p = priority(n,s)
\end{aligned}
$$

$$
\begin{aligned}
&shipmentCtr(do(a,s)) = n \;\equiv \\
&\quad [\,(\exists\, c, r, p)\; a = orderShipment(c,r,p) \land \\
&\quad\;\; shipmentCtr(s) = n' \;\land\; n = n' + 1\,] \lor \\
&\quad [\,(\forall\, c, r, p)\, a \neq orderShipment(c,r,p) \;\land\; n = shipmentCtr(s)\,]
\end{aligned}
$$

$$
\begin{aligned}
& robotStatus(do(a,s)) = i \ \equiv \\
& \quad [\,(\exists\, p)\ a = startGoTo(p) \ \wedge \ i = moving\,] \vee \\
& \quad [\,(a = resetRobot \ \vee \ a = msgFromRobot(reached)) \ \wedge \ i = idle\,] \vee \\
& \quad [\,a = msgFromRobot(blocked) \ \wedge \ i = blocked\,] \vee \\
& \quad [\,(\forall\, p)\ a \neq startGoTo(p) \ \wedge \ a \neq resetRobot \wedge \\
& \quad\ \ a \neq msgFromRobot(reached) \ \wedge \ a \neq msgFromRobot(blocked) \wedge \\
& \quad\ \ i = robotStatus(s)\,]
\end{aligned}
$$

$$
\begin{aligned}
& robotPlace(do(a,s)) = p \ \equiv \\
& \quad [\,a = msgFromRobot(reached) \ \wedge \ p = approaching(s)\,] \vee \\
& \quad [\,(\exists\, p')\ a = startGoTo(p') \ \wedge \ p = unknown\,] \vee \\
& \quad [\,p = robotPlace(s) \wedge \\
& \quad\ \ (\forall\, p')\ a \neq startGoTo(p') \ \wedge \ a \neq msgFromRobot(reached)\,]
\end{aligned}
$$

$$
\begin{aligned}
& haveCredit(c, do(a,s)) = m \ \equiv \\
& \quad [\,a = reduceCredit(c) \ \wedge \ haveCredit(c,s) = m' \ \wedge \ m = m' - 1\,] \vee \\
& \quad [\,a \neq reduceCredit(c) \ \wedge \ m = haveCredit(c,s)\,]
\end{aligned}
$$

$$
\begin{aligned}
& approaching(do(a,s)) = p \ \equiv \\
& \quad a = startGoTo(p) \vee \\
& \quad [\,p = approaching(s) \ \wedge \ (\forall\, p')\ a \neq startGoTo(p')\,]
\end{aligned}
$$

C.3 Defined Fluents

$$
\begin{aligned}
& bestCustToServe(c) \stackrel{\text{def}}{=} \\
& \quad needServing(c) \wedge \\
& \quad (\exists\, p)\,.\, haveShptOfPriority(c) = p \wedge \\
& \qquad isTheHighestPriority(p) \ \wedge \ haveHighestCredit(c,p)
\end{aligned}
$$

$$
\begin{aligned}
& needServing(c) \stackrel{\text{def}}{=} \\
& \quad (\exists\, n)\,.\, \{\, shipmentToPickUp(n) \ \wedge \ sender(n) = c \wedge \\
& \qquad\quad shipmentDeliverable(n) \ \wedge \ shipmentReturnable(n)\,\} \vee \\
& \qquad \{\, shipmentOnBoard(n) \ \wedge \ \neg\, clientSuspended(c) \wedge \\
& \qquad\quad [\,sender(n) = c \ \wedge \ \neg\, shipmentDeliverable(n)\,] \vee \\
& \qquad\quad recipient(n) = c\,\}
\end{aligned}
$$

$$
\begin{aligned}
&haveShptOfPriority(c) = p \stackrel{\text{def}}{=} \\
&\quad \{\, (\exists\, n)\,[\,(\,shipmentToPickUp(n) \ \wedge\ sender(n) = c \vee \\
&\qquad\qquad shipmentOnBoard(n) \ \wedge\ recipient(n) = c\,) \wedge \\
&\qquad\quad priority(n) = 1\,] \ \supset\ p = 1\,\} \wedge \\
&\quad \{\neg(\exists\, n)\,[\,(\,shipmentToPickUp(n) \ \wedge\ sender(n) = c \vee \\
&\qquad\qquad shipmentOnBoard(n) \ \wedge\ recipient(n) = c\,) \wedge \\
&\qquad\quad priority(n) = 1\,] \ \supset\ p = 0\,\}
\end{aligned}
$$

$$
\begin{aligned}
&isTheHighestPriority(p) \stackrel{\text{def}}{=} \\
&\quad (\exists\, c)\,[\,needServing(c) \ \wedge\ haveShptOfPriority(c) = p\,] \wedge \\
&\quad (\forall\, c', p')\,[\,needServing(c') \ \wedge\ haveShptOfPriority(c') = p' \ \supset\ p \geq p'\,]
\end{aligned}
$$

$$
\begin{aligned}
&haveHighestCredit(c, p) \stackrel{\text{def}}{=} \\
&\quad (\forall\, c', m, m')\,.\,needServing(c') \ \wedge\ haveShptOfPriority(c') = p \wedge \\
&\qquad\qquad c \neq c' \ \wedge\ haveCredit(c') = m' \ \wedge\ haveCredit(c) = m \\
&\qquad\qquad \supset\ m \geq m'
\end{aligned}
$$

$$
\begin{aligned}
&servingPriorityShpt(s) \stackrel{\text{def}}{=} \\
&\quad (\exists\, n, c)\,.\,approaching(s) = mailBox(c) \wedge \\
&\qquad\quad \{\,[\,shipmentToPickUp(n, s) \ \wedge\ sender(n, s) = c\,] \vee \\
&\qquad\quad\ \ [\,shipmentOnBoard(n, s) \ \wedge\ recipient(n, s) = c\,]\,\} \wedge \\
&\qquad\quad priority(n, s) = 1
\end{aligned}
$$

$$
\begin{aligned}
&shipmentReturnable(n, s) \stackrel{\text{def}}{=} \\
&\quad (\exists\, c)(sender(n, s) = c \ \wedge\ \neg clientSuspended(c, s))
\end{aligned}
$$

$$
\begin{aligned}
&shipmentDeliverable(n, s) \stackrel{\text{def}}{=} \\
&\quad (\exists\, r)(recipient(n, s) = r \ \wedge\ \neg clientSuspended(r, s))
\end{aligned}
$$

Appendix D

Sample Run: Runtime Failure and Exogenous Event Handling

This is the sample run of the example given in chapter 4 about runtime failures and exogenous events handling. Setting for test 2 is used to produce the following output. The program is listed in appendix B.

```
| ?- run2.
orderShipment(yves,jeff,1)            % shipment 1
ackShipmentOrder(1,yves,jeff)
orderShipment(michael,hector,0)       % shipment 2
orderShipment(yves,jeff,1)            % shipment 3
ackShipmentOrder(2,michael,hector)
ackShipmentOrder(3,yves,jeff)
startGoTo(mailBox(yves))
msgFromRobot(blocked)
orderShipment(michael,hector,0)       % shipment 4
ackShipmentOrder(4,michael,hector)
reduceCredit(yves)
resetRobot
startGoTo(mailBox(yves))
msgFromRobot(blocked)
reduceCredit(yves)
notifyStopServing(yves)
cancelPickUp(3,yves)
```

```
19: cancelPickUp(1,yves)
20: resetRobot
21: startGoTo(mailBox(michael))
22: msgFromRobot(blocked)
23: reduceCredit(michael)
24: resetRobot
25: startGoTo(mailBox(michael))
26: msgFromRobot(reached)
27: pickUpShipment(4)
28: pickUpShipment(2)
29: startGoTo(mailBox(hector))
30: msgFromRobot(blocked)
31: reduceCredit(hector)
32: resetRobot
33: startGoTo(mailBox(hector))
34: msgFromRobot(blocked)
35: reduceCredit(hector)
36: orderShipment(michael,hector,0)      % shipment 5
37: ackShipmentOrder(5,michael,hector)
38: notifyStopServing(hector)
39: orderShipment(michael,hector,0)      % shipment 6
40: cancelPickUp(5,michael)
41: ackWithRegret(6,michael,hector)
42: resetRobot
43: startGoTo(mailBox(michael))
44: msgFromRobot(blocked)
45: reduceCredit(michael)
46: orderShipment(yves,jeff,1)           % shipment 7
47: ackWithRegret(7,yves,jeff)
48: notifyStopServing(michael)
49: resetRobot
50: startGoTo(centralOffice)
51: msgFromRobot(blocked)
52: resetRobot
53: startGoTo(centralOffice)
54: msgFromRobot(blocked)
55: resetRobot
56: startGoTo(centralOffice)
57: msgFromRobot(blocked)
58: resetRobot
59: startGoTo(centralOffice)
60: msgFromRobot(blocked)
61: resetRobot
62: startGoTo(centralOffice)
63: msgFromRobot(blocked)
64: resetRobot
```

```
startGoTo(centralOffice)
msgFromRobot(blocked)
resetRobot
startGoTo(centralOffice)
msgFromRobot(reached)
dropOffShipment(2)
dropOffShipment(4)
shutdownRobot

yes
```

Appendix E

Program Listing: Knowledge-Based Search

This is the complete ConGolog program listing of the example given in chapter 5 about *knowledge-based search.*

```
/*************************************************************************
**                                                                      **
**                     Robot Mail Delivery Example                     **
**                          ConGolog version                           **
**                                                                      **
**                  Implemented  by  Kenneth B L Tam                   **
**                           July 18, 1997                             **
**                          February 23, 1998                          **
**                                                                      **
**************************************************************************
**************************************************************************
**                                                                      **
**    This is the controlling program for a robot mail delivery         **
**    system in an office setting.  The program is designed to be       **
**    highly responsive to exogenous events and runtime failures,       **
**    and use a knowledge base to searches for people.                  **
**                                                                      **
**    Runtime failure is handled by prioritized interrrupts while       **
**    the knowledge about the possible place a person can be found is   **
**    represented as ordered lists.  The lists will be updated with     **
**    addition information obtained at run time.                        **
**                                                                      **
```

```
**************************************************************************
**************************************************************************
**                                                                      **
**                            Updates                                   **
**                                                                      **
**    Aug 28, 97: The controller can start up without any shipment      **
**                order being placed.                                   **
**                                                                      **
**    Feb 23, 98: The controller conducts knowledge-based searches      **
**                                                                      **
**************************************************************************
**************************************************************************
**                                                                      **
**   Primitive Actions:                                                 **
**                                                                      **
**     For Normal Operations and Runtime Failures Handling              **
**         startGoTo(Place,Person)                                      **
**         ackShipmentOrder(ShNo,Sender,Recipient)                      **
**         pickUpShipment(ShNo)                                         **
**         dropOffShipment(ShNo)                                        **
**         shutdownRobot                                                **
**         cancelPickUp(ShNo,Sender)                                    **
**         reduceCredit(Person)                                         **
**         resetRobot                                                   **
**         notifyStopServing(Person)                                    **
**         ackWithRegret(ShNo,Sender,Recipient)                         **
**                                                                      **
**     For Knowledge-based search                                       **
**         askAbout(Person,Person)                                      **
**         delFromKnownPossPlaces(Person,Place)                         **
**         restorePlaceList(Person)                                     **
**                                                                      **
**   Sensing Actions:                                                   **
**         scanPlace                                                    **
**                                                                      **
**   Exogenous Actions:                                                 **
**         orderShipment(Sender,Recipient,Priority)                     **
**         msgFromRobot(robotStatus)                                    **
**         msgFromRobot(found([ListOfPerson])                           **
**         msgFromRobot(ansQuery([ListOfOption])                        **
**                                                                      **
**   ***************************************************************    **
**                                                                      **
**   Predicate Fluents:                                                 **
**                                                                      **
**     For Normal Operations and Runtime Failures Handling              **
```

```
**        shipmentOrdToAck(ShNo)                                         **
**        shipmentToPickUp(ShNo)                                         **
**        shipmentOnBoard(ShNo)                                          **
**        shipmentDelivered(ShNo)                                        **
**        clientSuspended(Person)                                        **
**        readyToBeServed                                                **
**        robotAlive                                                     **
**                                                                       **
**    For Knowledge-based search                                         **
**        waitingForAnswer                                               **
**        highPriorityShptJustPlaced                                     **
**                                                                       **
**  ***************************************************************      **
**                                                                       **
**  Functional Fluents:                                                  **
**                                                                       **
**    For Normal Operations and Runtime Failures Handling                **
**        sender(ShNo)                                                   **
**        recipient(ShNo)                                                **
**        priority(ShNo)                                                 **
**        shipmentCtr                                                    **
**        robotStatus                                                    **
**        robotPlace                                                     **
**        haveCredit(Person)                                             **
**        approaching                                                    **
**                                                                       **
**    For Knowledge-based search                                         **
**        serving                                                        **
**        vSensorReadings                                                **
**        knownPossPlacesFor(Person)                                     **
**                                                                       **
*************************************************************************



/* **********************************************************************
**                   Primitive Actions Declarations                     *
** *********************************************************************/

/* Primitive Actions for normal operations and handling runtime failures */
primAct(startGoTo(_,_)).
primAct(ackShipmentOrder(_,_,_)).
primAct(pickUpShipment(_)).
primAct(dropOffShipment(_)).
primAct(shutdownRobot).
primAct(cancelPickUp(_,_)).
```

```
primAct(reduceCredit(_)).      /* Reduce one credit from a client       */
primAct(resetRobot).           /* Halt and Reset robot                  */
primAct(notifyStopServing(_)). /* Stop serving a person                 */
primAct(ackWithRegret(_,_,_)). /* Acknowledge an invalid shipment order  */
                               /*   the shipment won't be served        */


/* Primitive Actions for Knowledge-based search */
primAct(askAbout(_,_)).               /* Ask a person about another      */
primAct(delFromKnownPossPlaces(_,_)). /* Remove a place from the         */
                                      /*   'knownPossPlacesFor'          */
primAct(restorePlaceList(_)).   /* Set the value of 'knownPossPlacesFor' */
                                /* to that of the 'possiblePlacesFor'    */

/* Primitive Sensing Actions Declarations */
primAct(scanPlace).                   /* Scan the surrounding            */


/* Exogenous Actions Declarations */
primAct(orderShipment(_,_,_)).
primAct(msgFromRobot(_)).           /* simulating feedbacks from robot   */



/* **********************************************************************
**                   Non-Fluent Declarations                            *
** **********************************************************************/

/* Non-Fluent Declarations */
non_fluent(person(_)).
non_fluent(possiblePlace(_,_)).
non_fluent(_\=_).
non_fluent(mightKnowBy(_,_)).



/* **********************************************************************
**                   Action Precondition Axioms                         *
** **********************************************************************/

/* Primitive Actions for normal operations and handling runtime failures */
poss(startGoTo(Place,Person),S):-           /* Redefined for KB's search */
   legalPlace(Place),
   ( person(Person); Person = clearMailOnBoard ),
   holds(val(robotStatus,idle), S).

```

```
poss(ackShipmentOrder(ShNo,Sndr,Rcpt),S):-
   holds(shipmentOrdToAck(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(recipient(ShNo),Rcpt),S).

poss(pickUpShipment(ShNo),S):-
   holds(shipmentToPickUp(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(vSensorReadings,ListOfPerson),S),         /* For KB'd search */
   member(Sndr,ListOfPerson).

poss(dropOffShipment(ShNo),S):-
   holds(shipmentOnBoard(ShNo),S),
   ( ( ( holds(val(recipient(ShNo),Person),S);
         holds(val(sender(ShNo),Person),S) ),
       holds(val(vSensorReadings,ListOfPerson),S),     /* For KB'd search */
       member(Person,ListOfPerson) );
     holds(val(robotPlace, centralOffice),S) ).

poss(shutdownRobot,_).

poss(cancelPickUp(ShNo,Sender),S) :-
   holds(shipmentToPickUp(ShNo),S), person(Sender).

poss(reduceCredit(Person),S) :-
   holds(val(haveCredit(Person),Credits),S),
   Credits > 0.

poss(resetRobot,S) :-
   not(holds(val(robotStatus,idle),S)).

poss(notifyStopServing(P),_) :-
   person(P).

poss(ackWithRegret(ShNo,Sndr,Rcpt),S):-
   holds(shipmentOrdToAck(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(recipient(ShNo),Rcpt),S).


/* Primitive Actions for Knowledge-based search */
poss(askAbout(Person1,Person2),S) :-
   person(Person1), person(Person2),
   holds(val(vSensorReadings,ListOfPerson),S),
   member(Person1,ListOfPerson).

```

```
poss(delFromKnownPossPlaces(Person,Place),_) :-
   person(Person),
   legalPlace(Place).

poss(restorePlaceList(Person),_) :-
   person(Person).

/* Primitive Sensing Actions */
poss(scanPlace,S) :-
   holds(val(robotStatus,idle),S).


/* Exogenous Actions */

/* Priority=1 means high priority, Priority=0 means low priority */
poss(orderShipment(Sender,Recipient,Priority),S) :-
   holds(robotAlive,S),
   person(Sender),
   person(Recipient),
   Priority < 2.

poss(msgFromRobot(A),S) :-
   holds(robotAlive,S),
   ( (A = reached; A = blocked),!,holds(val(robotStatus,moving),S) );
   ( A = found(_), !, holds(val(robotStatus,scanning),S) );
   ( A = ansQuery(_), !, holds(val(robotStatus,querying),S) ).



/* *********************************************************************
**                       Successor State Axioms                       *
** *********************************************************************/

/* For normal operations and handling runtime failures */
holds(shipmentOrdToAck(ShNo),do(A,S)):-
   ( A = orderShipment(_,_,_), holds(val(shipmentCtr,ShNo),S) );
   ( holds(shipmentOrdToAck(ShNo),S),
     A \= ackShipmentOrder(ShNo,_,_), A \= ackWithRegret(ShNo,_,_)).

holds(shipmentToPickUp(ShNo),do(A,S)):-
   A = ackShipmentOrder(ShNo,_,_);
   ( holds(shipmentToPickUp(ShNo),S),
     (A \= pickUpShipment(ShNo), A \= cancelPickUp(ShNo,_)) ).

holds(shipmentOnBoard(ShNo),do(A,S)):-
   A = pickUpShipment(ShNo);
```

```
   ( holds(shipmentOnBoard(ShNo),S),
     A \= dropOffShipment(ShNo) ).

holds(shipmentDelivered(ShNo),do(A,S)):-
   A = dropOffShipment(ShNo);
   A = ackWithRegret(ShNo,_,_);
   A = cancelPickUp(ShNo,_);
   holds(shipmentDelivered(ShNo),S).

/* Hold iff a client is no longer being served                         */
/* To make the action "cancelPickUp" executes concurrently with        */
/* robotMotionControl.  We need this as a sucessor state axiom as      */
/* well as haveCredit().                                               */
holds(clientSuspended(Cust),do(A,S)) :-
   A = notifyStopServing(Cust);
   holds(clientSuspended(Cust),S).

/* Hold once an order is made */
holds(readyToBeServed,do(A,S)) :-
   A = orderShipment(_,_,_);
   holds(readyToBeServed,S).

/* Robot is alive if it has not been shut down */
holds(robotAlive,do(A,S)) :-
   holds(robotAlive,S), A \= shutdownRobot.

holds(val(sender(ShNo),Sender),do(A,S)):-
   ( A = orderShipment(Sender,_,_), holds(val(shipmentCtr,ShNo),S) );
   holds(val(sender(ShNo),Sender),S).

holds(val(recipient(ShNo),Rec),do(A,S)):-
   ( A = orderShipment(_,Rec,_), holds(val(shipmentCtr,ShNo),S) );
   holds(val(recipient(ShNo),Rec),S).

/* Returns the priority of a shipment */
holds(val(priority(ShNo),Priority),do(A,S)):-
   ( A = orderShipment(_,_,Priority), holds(val(shipmentCtr,ShNo),S) );
   holds(val(priority(ShNo),Priority),S).

holds(val(shipmentCtr,N),do(A,S)):-
   ( A = orderShipment(_,_,_), !,
        holds(val(shipmentCtr,M),S), N is M + 1 );
   holds(val(shipmentCtr,N),S).

/* Returns the current status of robot */
/* Redefined for KB's search           */
```

```
holds(val(robotStatus,Status), do(A,S)):-
   ( A = startGoTo(_,_), !, Status = moving );
   ( (A = resetRobot; A = msgFromRobot(reached);
      A = msgFromRobot(found(_));
      (A = msgFromRobot(ansQuery(X)),X \= [0])), !,
      Status = idle );
   ( A = msgFromRobot(blocked), !, Status = blocked );
   ( A = scanPlace, !, Status = scanning );
   ( A = askAbout(_,_), !, Status = querying );
   holds(val(robotStatus,Status), S).

/* Returns the current position of the robot when not moving */
/* and returns unknown when the robot is moving              */
holds(val(robotPlace,Place),do(A,S)):-
   ( A = msgFromRobot(reached), holds(val(approaching,Place1),S), !,
     Place = Place1 );
   ( A = startGoTo(_,_), !, Place = unknown );
   holds(val(robotPlace,Place),S).

/* Returns the number of credits a person left.         */
/* Credits of a person will decrease by 1 if the robot  */
/* is blocked on the way to the person's mailbox.       */
holds(val(haveCredit(Person),Credits),do(A,S)):-
   ( A = reduceCredit(Person), !,
     holds(val(haveCredit(Person),CreditsLeft),S),
     Credits is CreditsLeft - 1 );
   holds(val(haveCredit(Person),Credits),S).

/* Returns the place to which the robot is heading               */
/* Value changes when the agent starts moving towards another's  */
/* mailBox                                                       */
holds(val(approaching,Place), do(A,S)) :-
   ( A = startGoTo(Place1,_), !, Place = Place1 );
   holds(val(approaching,Place),S).


/* For Knowledge-based search */
/* Returns the name of the person who is being served */
holds(val(serving,Cust), do(A,S)) :-
   ( A = startGoTo(_,Cust1), !, Cust = Cust1 );
   holds(val(serving,Cust),S).

/* Returns the most recent readings from the visual sensors            */
/* It returns a list of person found if there are any, or              */
/* the list [] if nothing is found after scanning the environment, or  */
/* [undefine] if the visual schema has not been triggered.             */
```

```
holds(val(vSensorReadings,What),do(A,S)) :-
   ( A = msgFromRobot(found(ListOfPerson)), !,
        What = ListOfPerson );
   ( A = scanPlace, !, What = [undefine] );
   holds(val(vSensorReadings,What),S).

/* Returns the current set of possible places where Cust might be  */
/* found.  All newly acquired information is queued at the front   */
/* of the lists and all place visited will be removed.             */
/* The list is maintained as an ordered list and all elements in   */
/* the list are unique.                                            */
holds(val(knownPossPlacesFor(Cust),Places),do(A,S)) :-
   ( (A = msgFromRobot(ansQuery(OriginalAns)), OriginalAns \= [0]),
        holds(val(serving,Cust),S), !,
        modifyPlacesList(OriginalAns,Ans),
        holds(val(knownPossPlacesFor(Cust),Places1),S),
        removeList(Ans,Places1,NewPlace),
        queueAtFront(Ans,NewPlace,Places) );
   ( A = restorePlaceList(Cust),
        (( holds(val(vSensorReadings,ListOfPerson),S),
           member(Cust,ListOfPerson), !,
           holds(val(robotPlace,CurPlace),S),
           possiblePlacesFor(Cust,Places1),
           removeElement(CurPlace,Places1,Places2),
           queueAtFront([CurPlace],Places2,Places) );
         possiblePlacesFor(Cust,Places) ));
   ( A = delFromKnownPossPlaces(Cust,Place), !,
        holds(val(knownPossPlacesFor(Cust),Places1),S),
        removeElement(Place,Places1,Places) );
   holds(val(knownPossPlacesFor(Cust),Places),S).

/* Waiting for answers */
holds(waitingForAnswer,do(A,S)) :-
   A = askAbout(_,_);
   ( holds(waitingForAnswer,S),
     A \= msgFromRobot(ansQuery(_)),
     A \= resetRobot ).

/* A high priority shipment has just been acknowledged */
holds(highPriorityShptJustPlaced,do(A,S)) :-
   ( A = resetRobot, holds(val(robotStatus,querying),S) );
   ( holds(highPriorityShptJustPlaced,S), A \= startGoTo(_,_) ).



/* ***********************************************************************
```

```
**                           Defined Fluents                             *
** **********************************************************************/

/* For normal operations and handling runtime failures */

/* Holds if Cust requires the services provided by the robot */
holds(needServing(Cust),S) :-
   ( holds(shipmentToPickUp(ShNo),S),
     holds(val(sender(ShNo),Cust),S),
     holds(shipmentDeliverable(ShNo),S),
     holds(shipmentReturnable(ShNo),S) );
   ( holds(shipmentOnBoard(ShNo),S),
     ( holds(val(recipient(ShNo),Cust),S);
       ( holds(val(sender(ShNo),Cust),S),
         not(holds(shipmentDeliverable(ShNo),S))) ),
       not(holds(clientSuspended(Cust),S)) ).

/* Returns 1 if Cust has some higher priority shipments, else returns 0 */
holds(val(haveShptOfPriority(Cust),P),S) :-
   ( holds(val(priority(ShNo),1),S),
     ((holds(shipmentToPickUp(ShNo),S), holds(val(sender(ShNo),Cust),S));
      (holds(shipmentOnBoard(ShNo),S), holds(val(recipient(ShNo),Cust),S))),
     !, P=1 );
   P=0.

/* Holds if Cust1 is the one with the largest amount of credits left */
/* among all those clients, including Cust1, who have some shipments */
/* of priority P                                                     */
holds(haveHighestCredit(Cust1,P),S) :-
   ( holds(val(haveCredit(Cust1),C1),S), holds(needServing(Cust2),S),
     Cust1 \= Cust2, holds(val(haveShptOfPriority(Cust2),P),S),
     holds(val(haveCredit(Cust2),C2),S), C2 > C1,
     !, not(holds(true,S)) );
   holds(true,S).

/* Holds if P1 is the highest priority amongst all available shipments */
holds(isTheHighestPriority(P1),S) :-
   ( holds(needServing(Cust),S),
     holds(val(haveShptOfPriority(Cust),P2),S),
     P2 > P1, !, not(holds(true,S)) );
   holds(true,S).

/* The best customer to serve is the one with highest priority and largest */
/* amount of credit left amongst those who are of the same priority level  */
holds(bestCustToServe(Cust),S) :-
   holds(needServing(Cust),S),
```

```
   holds(val(haveShptOfPriority(Cust),Priority),S),
   holds(isTheHighestPriority(Priority),S),
   holds(haveHighestCredit(Cust,Priority),S).

/* Holds if the robot is serving some higher priority shipment */
holds(servingPriorityShpt,S) :-
   holds(val(approaching,mailBox(Cust)),S),
   ( (holds(shipmentToPickUp(ShNo),S), holds(val(sender(ShNo),Cust),S));
     (holds(shipmentOnBoard(ShNo),S), holds(val(recipient(ShNo),Cust),S)) ),
   holds(val(priority(ShNo),1),S).

/* A shipment is returnable if its sender has not been suspended */
holds(shipmentReturnable(ShNo),S) :-
   holds(val(sender(ShNo),Cust),S), not(holds(clientSuspended(Cust),S)).

/* A shipment is deliverable if its recipient has not been suspended */
holds(shipmentDeliverable(ShNo),S) :-
   holds(val(recipient(ShNo),Cust),S), not(holds(clientSuspended(Cust),S)).

/* States in which interruption is allowed */
/* Redefined for KB'd search     */
holds(interruptAllowed,S) :-
   holds(val(robotStatus,State),S),
   ( State = moving; State = scanning; State = querying ).



/* For Knowledge-based search */

/* A best place to get Cust is the first place found in */
/* the Cust's ‘knowPossPlacesFor’                        */
holds(val(bestPlaceToGet(Cust),Place),S) :-
   holds(val(knownPossPlacesFor(Cust),Places),S),
        Places \= [],
        front(Places,Place).

/* Robot has found some objects */
holds(robotFoundSomething,S) :-
   holds(val(vSensorReadings,Info),S),
   Info \= [],
   Info \= [undefine].



/* **********************************************************************
**                          Non-fluents values                          *
```

```
**    Can also be treated as Adventitious Facts in our Knowledge Base  *
**                   Redefined for Knowledge-based search                *
** *******************************************************************/

/* Places in our domain */
legalPlace([X,Y,T]):- number(X), number(Y), number(T), !.
legalPlace(P):- placeValue(_,P,_).

/* placeValue is triples of <placeID#, placeName, placeCoord> */
placeValue(99, [X,Y,T],[X,Y,T]):- number(X), number(Y), number(T), !.
placeValue(81, robotHome,[1,1,0]).
placeValue(82, centralOffice,[5,5,0]).
placeValue(1, lab1,[2,3,0]).
placeValue(2, lab2,[5,3,0]).
placeValue(3, lab3,[1,5,0]).
placeValue(4, lab4,[3,4,0]).

/* People in our domain */
person(X) :-
   X = yves; X = michael; X = jeff; X = hector.

/* Relations of People and Places */
/* It represent the possible places to get a person */
possiblePlacesFor(yves,[lab1,lab2,lab3]).
possiblePlacesFor(michael,[lab3,lab4]).
possiblePlacesFor(jeff,[lab4,lab1]).
possiblePlacesFor(hector,[lab2,lab4,lab3]).

/* ‘mightKnowBy’ Relation between peoples */
/* mightKnowBy(A,[B]) means A might known by any element in [B] */
mightKnowBy(yves,[michael,jeff]).
mightKnowBy(michael,[jeff]).
mightKnowBy(hector,[yves]).

/* Helper functions to decode options into name of places */
modifyPlacesList([],[]).
modifyPlacesList([PlaceCode1|Rest],[Place|ListOfPlaces]) :-
   placeValue(PlaceCode1,Place,_),
   modifyPlacesList(Rest,ListOfPlaces).

/* Returns the first element of a queue without removing it */
front([Element|_],Element).

/* Element and ListOfElement must both be lists */
queueAtFront(Element,ListOfElement,NewList) :-
    append(Element,ListOfElement,NewList).
```

```

/* Removes a subset from a set */
removeList([],ListOfElement,ListOfElement).
removeList([First|Last],ListOfElement,NewList) :-
    removeElement(First,ListOfElement,NewList1),
    removeList(Last,NewList1,NewList).

/* Removes a single element from a set */
removeElement(_,[],[]).
removeElement(Element,[Element|ListOfElement],ListOfElement).
removeElement(Element,[Element1|ListOfElement],[Element1|NewList]) :-
    Element \== Element1,
    removeElement(Element,ListOfElement,NewList).



/* ********************************************************************
**                          Initial States                            *
** *********************************************************************/

holds(val(robotPlace,Place),s0) :- Place = robotHome.
holds(val(shipmentCtr,N),s0) :- N = 1.
holds(val(robotStatus, State),s0) :- State = idle.
holds(robotAlive,s0).
holds(val(vSensorReadings,Readings),s0) :- Readings = [undefine].
holds(val(approaching,Place), s0) :- Place = nowhere.

/* Change the following to obtain the maximum number of blocking allowed */
holds(val(haveCredit(Person),N),s0) :- person(Person), N = 1.

holds(val(knownPossPlacesFor(Person),Places),s0) :-
    person(Person),
    possiblePlacesFor(Person,Places).



/* ********************************************************************
**                          Exogenous Actions                         *
**                       Use for simulation only   *
** *********************************************************************/

/* This version relies on the "feedback" from robot         */
/* So, the predicate msgFromRobot() cannot be commented out */
exoDensity(110).

exoAct(orderShipment(yves,jeff,1),1,2).
```

```
exoAct(orderShipment(michael,hector,0),5,6).

exoAct(msgFromRobot(blocked),7,37).
exoAct(msgFromRobot(reached),48,88).

exoAct(msgFromRobot(found([])),93,94).
exoAct(msgFromRobot(found([jeff])),95,96).
exoAct(msgFromRobot(found([jeff,michael])),97,98).
exoAct(msgFromRobot(found([hector,yves])),99,100).

exoAct(msgFromRobot(ansQuery([0])), 101,102).
exoAct(msgFromRobot(ansQuery([3,4])), 103,104).
exoAct(msgFromRobot(ansQuery([1,2])), 105,106).



/* ***********************************************************************
**                             Tracing Controls                          *
** ***********************************************************************/

/*
tracingProg.
tracingTest.
tracingExec.
*/



/* ***********************************************************************
**                          Procedure Definitions                        *
** ***********************************************************************/

/* Procedures defined for normal operations and runtime failure handling */

/* Acknowledges a shipment if there are any */
proc(acknowledgeShipmentOrder(ShNo),
   pi([sndr,rcpt], [ and(val(sender(ShNo),sndr),val(recipient(ShNo),rcpt))?,
        if(and(shipmentReturnable(ShNo),shipmentDeliverable(ShNo)),
            [ ackShipmentOrder(ShNo,sndr,rcpt),
              if(and(val(priority(ShNo),1),
                     and(interruptAllowed, not(servingPriorityShpt))),
                    resetRobot,
                    no_op
              ) ],
            ackWithRegret(ShNo,sndr,rcpt)
        )
```

```
   ] )
).


/* Top level control of the robot: Serve a client if there are any,    */
/* otherwise, go to central office to drop off all remaining shipments */
/* on board                                                            */
proc(robotMotionControl,
   if(some(cust, needServing(cust)),
         serveCustomer,
         [ startGoTo(centralOffice,clearMailOnBoard),
           not(val(robotStatus,moving))?,
           if(val(robotPlace,centralOffice),
                 [ dropOffToCentralOffice,
                   if(some(cust2, needServing(cust2)),
                      no_op,
                      shutdownRobot)
                 ],
                 if(val(robotStatus,blocked),
                       resetRobot,
                       no_op
                 )
           ) ]
   )
).


/* Define the procedure of what to do when a customer who requires */
/* serving is identified: Goes to his/her mailbox and check        */
/* whether the customer is there if arrived at the destination,    */
/* otherwise, reset the robot.                                     */
proc(serveCustomer,
   pi([place,cust], [ and(bestCustToServe(cust),
                          val(bestPlaceToGet(cust),place))?,
         [ startGoTo(place,cust),
           not(val(robotStatus,moving))?,
           if(val(robotPlace,place),
                 isCustHere(cust,place),   /* Redefined for KB'd search */
                 if(val(robotStatus,blocked),
                       resetRobot,
                       no_op
                 )
           ) ]
   ] )
).

```

```
/* Adjusts the credit of Cust: Reduces Cust's credit by 1 and if   */
/* the remaining credit is 0, then notifies Cust that he/she will  */
/* not be served anymore.                                          */
/* A procedure expanded for knowledge-based search                 */
proc(adjustCredit(Cust),
   [ pi(credits, [ and(val(haveCredit(Cust),credits), credits > 0)?,
        [ reduceCredit(Cust),
          if(val(haveCredit(Cust),0),
              notifyStopServing(Cust),
              no_op)
        ]
   ] ) ]
).


/* Drops off all onboard invalid shipments to central office */
proc(dropOffToCentralOffice,
   while(some(shNo, and(shipmentOnBoard(shNo),
                        not(or(shipmentDeliverable(shNo),
                               shipmentReturnable(shNo))))),
       pi(shNo, [and(shipmentOnBoard(shNo),
                     not(or(shipmentDeliverable(shNo),
                            shipmentReturnable(shNo))))?,
          dropOffShipment(shNo)])
 )
).


/* Picks up all shipments from Cust if there are any */
proc(pickUpShipmentsFrom(Cust),
   while(some(shNo, and(shipmentToPickUp(shNo),val(sender(shNo),Cust))),
      pi(shNo, [ and(shipmentToPickUp(shNo),val(sender(shNo),Cust))?,
         pickUpShipment(shNo) ]))).


/* Drops off all Cust's shipments onboard to Cust */
proc(dropOffShipmentsTo(Cust),
   while(some(shNo, and(shipmentOnBoard(shNo),
                        or(val(recipient(shNo),Cust),
                           and(val(sender(shNo),Cust),
                               not(shipmentDeliverable(shNo)))))),
      pi(shNo, [ and(shipmentOnBoard(shNo),
                     or(val(recipient(shNo),Cust),
                        and(val(sender(shNo),Cust),
                            not(shipmentDeliverable(shNo)))))?,
```

```
         dropOffShipment(shNo) ]))).



/* Procedures defined for knowledge based search */

/* Procedure to handle the case of exhausted place list:               */
/* Cust's credit will first be reduced by 1.  If the remaining credit  */
/* is positive, then knownPossPlacesFor set will be reset to the value */
/* of the possiblePlacesFor.                                           */
proc(checkPlaceList(Cust),
   if(val(knownPossPlacesFor(Cust),[]),
        [ adjustCredit(Cust),
          if(clientSuspended(Cust),
               no_op,
               restorePlaceList(Cust)
          )
        ],
        no_op )
).


/* Scans the place to found out whether some people are there */
proc(isCustHere(Cust,Place),
   [ scanPlace,
     not(val(robotStatus,scanning))?,
     if(val(vSensorReadings,[undefine]),
        no_op,
        [ delFromKnownPossPlaces(Cust,Place),
          somebodyHere(Cust) ]
     )
   ]
).


/* If somebody are there and Cust is one of them, serves Cust,   */
/* otherwise, asks others where Cust can be found.               */
proc(somebodyHere(Cust),
   pi(people, [ val(vSensorReadings,people)?,
     if(member(Cust,people),
        [ pickUpShipmentsFrom(Cust),
          dropOffShipmentsTo(Cust),
          restorePlaceList(Cust) ],
        [ if(not(people=[]),
             [ seekHelpFrom(people,Cust),
               if(highPriorityShptJustPlaced,
```

```
                  no_op,
                  checkPlaceList(Cust) ) ],
             checkPlaceList(Cust) ) ]
       )
   ] )
).


/* Seeks help from the people nearby: Only those people in the    */
/* mightKnowBy relation will be asked                             */
proc(seekHelpFrom(ListOfPerson,Cust),
   [ for(person, ListOfPerson, [possSource],
         if(and(mightKnowBy(Cust,possSource),member(person,possSource)),
            [ askAbout(person,Cust),
              not(waitingForAnswer)? ],
            no_op
         ),
         not(val(robotStatus,idle))
     ),
     if(not(val(robotStatus,idle)),
        resetRobot,
        no_op)
   ]
).



/* ************************************************************************
**                              Main Program                             *
** ************************************************************************/

runSystem(Ini,Final) :-
   initialize(1),

   pdo(
   /* Acknowledges shipment order if there are any */
   (==>([shNo], shipmentOrdToAck(shNo),
           acknowledgeShipmentOrder(shNo)))

   #=
   /* Cancels pick up if there are any invalid shipment waiting for pickup */
   (==>([shNo,sender], and(and(shipmentToPickUp(shNo),
        val(sender(shNo),sender)),
                           not(and(shipmentReturnable(shNo),
                                   shipmentDeliverable(shNo)))),
           cancelPickUp(shNo,sender)))
```

```
   #>
   /* Main Loop to serve a client */
   (==>([shNo], and(readyToBeServed,
             or(or(shipmentToPickUp(shNo), shipmentOnBoard(shNo)),
                       not(val(robotPlace,centralOffice)))),
           robotMotionControl))

   #>
   /* Skips if the robot is moving */
   or(not(readyToBeServed), robotAlive) ==> no_op,

   Ini,Final).



/* ***********************************************************************
**        An Auxiliary Program to break the final state down into        *
**         a sequence of actions starting from the initial state         *
** **********************************************************************/

output(do(A,s0)) :- format('~N~w',A).
output(do(A,S)) :-  output(S), format('~N~w',A), nl.

run2 :- runSystem(s0,Final), output(Final).

```

Appendix F

The Complete Set of New and Modified Axioms: Knowledge-Based Search

This is the complete listing of all new or modified axioms (in Situation Calculus notation) of the mail delivery example given in chapter 5 involving knowledge-based search.

F.1 Action Precondition Axioms

$$
\begin{aligned}
&Poss(startGoTo(l,c),s) \;\equiv\; legalPlace(l) \wedge \\
&\qquad (person(c) \;\vee\; clearMailOnBoard = c) \;\wedge\; robotStatus(s) = idle
\end{aligned}
$$

$$
\begin{aligned}
&Poss(pickUpShipment(n),s) \;\equiv\; (\exists\, c)\,.shipmentToPickUp(n,s) \wedge \\
&\qquad sender(n,s) = c \;\wedge\; (\exists\, v)(vSensorReadings(s) = v \;\wedge\; c \,\in\, v)
\end{aligned}
$$

$$
\begin{aligned}
&Poss(dropOffShipment(n),s) \;\equiv\; shipmentOnBoard(n,s) \wedge \\
&\qquad (\exists\, c)\,[\,(\,sender(n,s) = c \;\vee\; recipient(n,s) = c\,) \wedge \\
&\qquad\qquad (\exists\, v)(vSensorReadings(s) = v \;\wedge\; c \,\in\, v)\,] \vee \\
&\qquad robotPlace(s) = centralOffice
\end{aligned}
$$

$$Poss(askAbout(c', c), s) \equiv (\exists t) . person(c) \wedge person(c') \wedge \\ vSensorReadings(s) = t \wedge c' \in t$$

$$Poss(delFromKnownPossPlaces(c, p), s) \equiv person(c) \wedge legalPlace(p)$$

$$Poss(restorePlaceList(c), s) \equiv person(c)$$

$$Poss(scanPlace, s) \equiv robotStatus(s) = idle$$

$$Poss(msgFromRobot(i), s) \equiv robotAlive(s) \wedge \\ \{ [(i = reached \vee i = blocked) \wedge robotStatus(s) = moving] \vee \\ [(\exists v)\, i = found(v) \wedge robotStatus(s) = scanning] \vee \\ [(\exists k)\, i = ansQuery(k) \wedge robotStatus(s) = querying] \}$$

F.2 Successor State Axioms

$$robotStatus(do(a, s)) = i \equiv \\ [(\exists p)\ a = startGoTo(p) \wedge i = moving] \vee \\ [(a = resetRobot \vee a = msgFromRobot(reached) \vee \\ (\exists t)\ a = msgFromRobot(found(t)) \vee \\ (\exists k)\ (a = msgFromRobot(ansQuery(k)) \wedge k \neq \texttt{[ ]})) \wedge \\ i = idle] \vee \\ [a = msgFromRobot(blocked) \wedge i = blocked] \vee \\ [a = scanPlace \wedge i = scanning] \vee \\ [(\exists c', c)\ a = askAbout(c', c) \wedge i = querying] \vee \\ [(\forall p)\ a \neq startGoTo(p) \wedge a \neq resetRobot \wedge a \neq scanPlace \wedge \\ (\forall m)\ (a \neq msgFromRobot(m) \vee m = ansQuery(\texttt{[ ]})) \wedge \\ (\forall c', c)\ a \neq askAbout(c', c) \wedge i = robotStatus(s)]$$

$$serving(do(a, s)) = c \equiv \\ [(\exists l)\ a = startGoTo(l, c)] \vee \\ [(\forall l, c')\ a \neq startGoTo(l, c') \wedge c = serving(s)]$$

$$
\begin{array}{l}
vSensorReadings(do(a,s)) = t \;\equiv \\
\quad [\, a = msgFromRobot(found(t))\,] \vee \\
\quad [\, a = scanPlace \;\wedge\; t = \bot\,] \vee \\
\quad [\, (\forall t')\, a \neq msgFromRobot(found(t')) \;\wedge\; a \neq scanPlace \wedge \\
\quad\;\; t = vSensorReadings(s)\,]
\end{array}
$$

$$
\begin{array}{l}
knownPossPlacesFor(c, do(a,s))) = l \;\equiv \\
\quad [\, (\exists p, l')\; (\, a = delFromKnownPossPlaces(c,p) \wedge \\
\qquad\qquad knownPossPlacesFor(c,s) = l' \wedge \\
\qquad\qquad l = removeElement(p,l')\,)\,] \vee \\
\quad [\, (\exists t, p, l', l'')\; (\, a = restorePlaceList(c) \;\wedge\; vSensorReadings(s) = t \wedge \\
\qquad\qquad\quad c \in t \;\wedge\; possiblePlacesFor(c) = l' \wedge \\
\qquad\qquad\quad robotPlace(s) = p \;\wedge\; removeElement(p,l') = l'' \wedge \\
\qquad\qquad\quad l = queueAtFront(p,l'')\,)\,] \vee \\
\quad [\, (\exists t)\; (\, a = restorePlaceList(c) \wedge \\
\qquad\quad \neg\, (vSensorReadings(s) = t \;\wedge\; c \in t) \wedge \\
\qquad\quad l = possiblePlacesFor(c)\,)\,] \vee \\
\quad [\, (\exists k, l', l'')\; (\, a = msgFromRobot(ansQuery(k)) \;\wedge\; k \neq \texttt{[ ]} \wedge \\
\qquad\qquad serving(s) = c \;\wedge\; knownPossPlacesFor(c,s) = l' \wedge \\
\qquad\qquad removeList(k,l') = l'' \;\wedge\; l = queueAtFront(k,l'')\,)\,] \vee \\
\quad [\, a \neq restorePlaceList(c) \wedge \\
\quad\;\; (\forall\, p)\; a \neq delFromKnownPossPlaces(c,p) \wedge \\
\quad\;\; (\forall\, k)\; \neg\, (\, a = msgFromRobot(ansQuery(k)) \;\wedge\; k \neq \texttt{[ ]} \wedge \\
\qquad\qquad serving(s) = c) \wedge \\
\quad\;\; l = knownPossPlacesFor(c,s)\,]
\end{array}
$$

$$
\begin{array}{l}
waitingForAnswer(do(a,s)) \;\equiv \\
\quad (\exists c, c')\; a = askAbout(c,c') \vee \\
\quad [\, waitingForAnswer(s) \;\wedge\; a \neq resetRobot \wedge \\
\quad\;\; (\forall k)\; a \neq msgFromRobot(ansQuery(k))\,]
\end{array}
$$

$$
\begin{array}{l}
highPriorityShptJustPlaced(do(a,s)) \;\equiv \\
\quad [\, a = resetRobot \;\wedge\; robotStatus(s) = querying\,] \vee \\
\quad [\, highPriorityShptJustPlaced(s) \;\wedge\; (\forall l, c')\; a \neq startGoTo(l,c')\,]
\end{array}
$$

F.3 Defined Fluents

$$interruptAllowed(s) \stackrel{\text{def}}{=}$$
$$(\exists\, i)\,(i = moving \;\vee\; i = scanning \;\vee\; i = querying)$$

$$bestPlaceToGet(c) = p \stackrel{\text{def}}{=}$$
$$(\forall\, l)\,[\, knownPossPlacesFor(c) = l \;\wedge\; l \neq \texttt{[ ]} \supset\; p = front(l)\,] \wedge$$
$$[\neg(\exists\, l)\,(\, knownPossPlacesFor(c) = l \;\wedge\; l \neq \texttt{[ ]}\,) \;\supset\; p = \perp\,]$$

$$robotFoundSomething(s) \stackrel{\text{def}}{=}$$
$$(\exists\, v)\,(vSensorReadings(s) = v \;\wedge\; v \neq \emptyset \;\wedge\; v \neq \perp)$$

Note that $\perp$ indicates an "undefined" reading.

Appendix G

Sample Run: Knowledge-Based Search

This is the sample run of the example given in chapter 5 about *knowledge-based search* . The program is listed in Appendix E.

```
Note: The annotations show the value of the yves' knownPossPlacesFor
      after the action on the same line has been executed.

1: | ?- run2.
2: orderShipment(michael,hector,0)          % <lab1,lab2,lab3>
3: orderShipment(yves,jeff,1)
4: ackShipmentOrder(1,michael,hector)
5: ackShipmentOrder(2,yves,jeff)
6: startGoTo(lab1,yves)
7: orderShipment(michael,hector,0)
8: msgFromRobot(reached)
9: ackShipmentOrder(3,michael,hector)
10: scanPlace
11: msgFromRobot(found([hector,yves]))
12: delFromKnownPossPlaces(yves,lab1)       % <lab2,lab3>
13: pickUpShipment(2)
14: restorePlaceList(yves)                  % <lab1,lab2,lab3>
15: startGoTo(lab4,jeff)
16: msgFromRobot(reached)
17: scanPlace
18: msgFromRobot(found([jeff]))
19: delFromKnownPossPlaces(jeff,lab4)
```

```
20: dropOffShipment(2)
21: restorePlaceList(jeff)
22: startGoTo(lab3,michael)
23: msgFromRobot(reached)
24: scanPlace
25: msgFromRobot(found([]))
26: orderShipment(michael,hector,0)
27: ackShipmentOrder(4,michael,hector)
28: delFromKnownPossPlaces(michael,lab3)
29: startGoTo(lab4,michael)
30: msgFromRobot(reached)
31: scanPlace
32: orderShipment(yves,jeff,1)
33: ackShipmentOrder(5,yves,jeff)
34: resetRobot
35: startGoTo(lab1,yves)
36: msgFromRobot(blocked)
37: resetRobot
38: startGoTo(lab1,yves)
39: msgFromRobot(blocked)
40: resetRobot
41: startGoTo(lab1,yves)
42: msgFromRobot(blocked)
43: resetRobot
44: startGoTo(lab1,yves)
45: msgFromRobot(blocked)
46: resetRobot
47: startGoTo(lab1,yves)
48: msgFromRobot(blocked)
49: resetRobot
50: startGoTo(lab1,yves)
51: msgFromRobot(reached)
52: scanPlace
53: msgFromRobot(found([]))
54: delFromKnownPossPlaces(yves,lab1)        % <lab2,lab3>
55: startGoTo(lab2,yves)
56: msgFromRobot(blocked)
57: resetRobot
58: startGoTo(lab2,yves)
59: msgFromRobot(reached)
60: scanPlace
61: msgFromRobot(found([hector,yves]))
62: delFromKnownPossPlaces(yves,lab2)        % <lab3>
63: pickUpShipment(5)
64: restorePlaceList(yves)                   % <lab2,lab1,lab3>
65: startGoTo(lab4,jeff)
```

```
66: msgFromRobot(blocked)
67: resetRobot
68: startGoTo(lab4,jeff)
69: msgFromRobot(blocked)
70: resetRobot
71: startGoTo(lab4,jeff)
72: msgFromRobot(reached)
73: scanPlace
74: msgFromRobot(found([hector,yves]))
75: delFromKnownPossPlaces(jeff,lab4)
76: orderShipment(yves,jeff,1)
77: ackShipmentOrder(6,yves,jeff)
78: startGoTo(lab2,yves)
79: orderShipment(michael,hector,0)
80: orderShipment(yves,jeff,1)
81: msgFromRobot(reached)
82: ackShipmentOrder(8,yves,jeff)
83: ackShipmentOrder(7,michael,hector)
84: scanPlace
85: msgFromRobot(found([jeff,michael]))
86: orderShipment(michael,hector,0)
87: orderShipment(michael,hector,0)
88: orderShipment(michael,hector,0)
89: ackShipmentOrder(9,michael,hector)
90: orderShipment(yves,jeff,1)
91: ackShipmentOrder(11,michael,hector)
92: ackShipmentOrder(12,yves,jeff)
93: ackShipmentOrder(10,michael,hector)
94: delFromKnownPossPlaces(yves,lab2)         % <lab1,lab3>
95: askAbout(jeff,yves)
96: msgFromRobot(ansQuery([3,4]))             % <lab3,lab4,lab1>
97: startGoTo(lab3,yves)
98: msgFromRobot(blocked)
99: resetRobot
100: startGoTo(lab3,yves)
101: msgFromRobot(blocked)
102: resetRobot
103: orderShipment(michael,hector,0)
104: ackShipmentOrder(13,michael,hector)
105: startGoTo(lab3,yves)
106: msgFromRobot(reached)
107: scanPlace
108: msgFromRobot(found([hector,yves]))
109: delFromKnownPossPlaces(yves,lab3)        % <lab4,lab1>
110: orderShipment(yves,jeff,1)
111: ackShipmentOrder(14,yves,jeff)
```

```
112: pickUpShipment(14)
113: pickUpShipment(12)
114: pickUpShipment(8)
115: pickUpShipment(6)
116: restorePlaceList(yves)                    % <lab3,lab1,lab2>
117: startGoTo(lab1,jeff)
118: msgFromRobot(blocked)
119: resetRobot
120: startGoTo(lab1,jeff)
121: msgFromRobot(reached)
122: scanPlace
123: orderShipment(yves,jeff,1)
124: msgFromRobot(found([]))
125: ackShipmentOrder(15,yves,jeff)
126: delFromKnownPossPlaces(jeff,lab1)
127: reduceCredit(jeff)
128: notifyStopServing(jeff)
129: cancelPickUp(15,yves)
130: startGoTo(lab4,michael)
131: msgFromRobot(blocked)
132: resetRobot
133: startGoTo(lab4,michael)
134: msgFromRobot(blocked)
135: resetRobot
136: startGoTo(lab4,michael)
137: msgFromRobot(reached)
138: scanPlace
139: msgFromRobot(found([hector,yves]))
140: delFromKnownPossPlaces(michael,lab4)
141: reduceCredit(michael)
142: notifyStopServing(michael)
143: cancelPickUp(13,michael)
144: cancelPickUp(10,michael)
145: cancelPickUp(11,michael)
146: cancelPickUp(9,michael)
147: cancelPickUp(7,michael)
148: cancelPickUp(4,michael)
149: cancelPickUp(3,michael)
150: cancelPickUp(1,michael)
151: startGoTo(lab3,yves)
152: msgFromRobot(reached)
153: scanPlace
154: msgFromRobot(found([jeff,michael]))
155: delFromKnownPossPlaces(yves,lab3)           % <lab1,lab2>
156: askAbout(jeff,yves)
157: orderShipment(michael,hector,0)
```

```
158: ackWithRegret(16,michael,hector)
159: msgFromRobot(ansQuery([0]))
160: msgFromRobot(ansQuery([3,4]))               % <lab3,lab4,lab1,lab2>
161: startGoTo(lab3,yves)
162: msgFromRobot(blocked)
163: resetRobot
164: orderShipment(yves,jeff,1)
165: ackWithRegret(17,yves,jeff)
166: startGoTo(lab3,yves)
167: msgFromRobot(reached)
168: scanPlace
169: orderShipment(yves,jeff,1)
170: ackWithRegret(18,yves,jeff)
171: msgFromRobot(found([]))
172: delFromKnownPossPlaces(yves,lab3)           % <lab4,lab1,lab2>
173: orderShipment(michael,hector,0)
174: orderShipment(yves,jeff,1)
175: ackWithRegret(19,michael,hector)
176: ackWithRegret(20,yves,jeff)
177: startGoTo(lab4,yves)
178: msgFromRobot(reached)
179: scanPlace
180: msgFromRobot(found([jeff]))
181: delFromKnownPossPlaces(yves,lab4)           % <lab1,lab2>
182: orderShipment(yves,jeff,1)
183: orderShipment(yves,jeff,1)
184: ackWithRegret(22,yves,jeff)
185: ackWithRegret(21,yves,jeff)
186: askAbout(jeff,yves)
187: orderShipment(michael,hector,0)
188: ackWithRegret(23,michael,hector)
189: msgFromRobot(ansQuery([3,4]))               % <lab3,lab4,lab1,lab2>
190: startGoTo(lab3,yves)
191: msgFromRobot(reached)
192: scanPlace
193: msgFromRobot(found([]))
194: delFromKnownPossPlaces(yves,lab3)           % <lab4,lab1,lab2>
195: startGoTo(lab4,yves)
196: msgFromRobot(reached)
197: scanPlace
198: orderShipment(yves,jeff,1)
199: orderShipment(yves,jeff,1)
200: ackWithRegret(25,yves,jeff)
201: ackWithRegret(24,yves,jeff)
202: msgFromRobot(found([hector,yves]))
203: delFromKnownPossPlaces(yves,lab4)           % <lab1,lab2>
```

```
dropOffShipment(6)
dropOffShipment(8)
dropOffShipment(12)
orderShipment(yves,jeff,1)
ackWithRegret(26,yves,jeff)
dropOffShipment(14)
orderShipment(yves,jeff,1)
ackWithRegret(27,yves,jeff)
dropOffShipment(5)
restorePlaceList(yves)                    % <lab1,lab2,lab3>
startGoTo(centralOffice,clearMailOnBoard)
msgFromRobot(blocked)
resetRobot
startGoTo(centralOffice,clearMailOnBoard)
msgFromRobot(reached)
shutdownRobot

yes
```

Appendix H

Program Listing: ConGolog Program to drive Mailmond

This is the ConGolog program used in chapter 7 to drive Mailmond, a real mobile robot. The original code can be found in appendix B, which concerns about runtime failures and exogenous events handling.

```
/*************************************************************************
**                                                                      **
**                  Robot Mail Delivery Example                         **
**                       ConGolog version                               **
**              (For GlobalCommit and Roll Forwards)                    **
**                                                                      **
**                Implemented  by  Kenneth B L Tam                      **
**                        July 18, 1997                                 **
**                                                                      **
**************************************************************************
**************************************************************************
**                                                                      **
**    This is the controlling program for a robot mail delivery         **
**    system in an office setting.  The program is designed to be       **
**    highly responsive to exogenous events and runtime failures        **
**                                                                      **
**************************************************************************
**************************************************************************
**                                                                      **
**                            Updates                                   **
```

```
**                                                                      **
**    Aug 28, 97: The controller can start up without any shipment      **
**                order being placed.                                   **
**                                                                      **
**    Feb 23, 98: Minor Bugs fixed.                                     **
**                                                                      **
**************************************************************************
**************************************************************************
**                                                                      **
**    Primitive Actions:                                                **
**                                                                      **
**      For Normal Operations and Runtime Failures Handling             **
**         startGoTo(Place)                                             **
**         ackShipmentOrder(ShNo,Sender,Recipient)                      **
**         pickUpShipment(ShNo)                                         **
**         dropOffShipment(ShNo)                                        **
**         shutdownRobot                                                **
**         cancelPickUp(ShNo,Sender)                                    **
**         reduceCredit(Person)                                         **
**         resetRobot                                                   **
**         notifyStopServing(Person)                                    **
**         ackWithRegret(ShNo,Sender,Recipient)                         **
**                                                                      **
**    Exogenous Actions:                                                **
**         orderShipment(Sender,Recipient,Priority)                     **
**         msgFromRobot(robotStatus)                                    **
**                                                                      **
**    ************************************************************      **
**                                                                      **
**    Predicate Fluents:                                                **
**                                                                      **
**      For Normal Operations and Runtime Failures Handling             **
**         shipmentOrdToAck(ShNo)                                       **
**         shipmentToPickUp(ShNo)                                       **
**         shipmentOnBoard(ShNo)                                        **
**         shipmentDelivered(ShNo)                                      **
**         clientSuspended(Person)                                      **
**         readyToBeServed                                              **
**         robotAlive                                                   **
**                                                                      **
**    ************************************************************      **
**                                                                      **
**    Functional Fluents:                                               **
**                                                                      **
**      For Normal Operations and Runtime Failures Handling             **
**         sender(ShNo)                                                 **
```

```
**        recipient(ShNo)                                          **
**        priority(ShNo)                                           **
**        shipmentCtr                                              **
**        robotStatus                                              **
**        robotPlace                                               **
**        haveCredit(Person)                                       **
**        approaching                                              **
**                                                                 **
***********************************************************************/



/* ********************************************************************
**                  Primitive Actions Declarations                    *
** ********************************************************************/

/* Primitive Actions for normal operations and handling runtime failures */
primAct(startGoTo(_)).
primAct(ackShipmentOrder(_,_,_)).
primAct(pickUpShipment(_)).
primAct(dropOffShipment(_)).
primAct(shutdownRobot).
primAct(cancelPickUp(_,_)).
primAct(reduceCredit(_)).      /* Reduce one credit from a client          */
primAct(resetRobot).           /* Halt and Reset robot                      */
primAct(notifyStopServing(_)). /* Stop serving a person                     */
primAct(ackWithRegret(_,_,_)). /* Acknowledge an invalid shipment order     */
                               /*   the shipment won't be served            */


/* Exogenous Actions Declarations */
primAct(orderShipment(_,_,_)).
primAct(msgFromRobot(_)).      /* simulating feedbacks from robot */



/* ********************************************************************
**                    Non-Fluent Declarations                         *
** ********************************************************************/

/* Non-Fluent Declarations */
non_fluent(person(_)).



/* ********************************************************************
```

```
**                    Action Precondition Axioms                        *
** ***********************************************************************/

/* Primitive Actions for normal operations and handling runtime failures */

/* For Primitive Actions */
poss(startGoTo(Place),S):-
   legalPlace(Place),
   holds(val(robotStatus,idle), S).

poss(ackShipmentOrder(ShNo,Sndr,Rcpt),S):-
   holds(shipmentOrdToAck(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(recipient(ShNo),Rcpt),S).

poss(pickUpShipment(ShNo),S):-
   holds(shipmentToPickUp(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(robotPlace,mailBox(Sndr)),S).

poss(dropOffShipment(ShNo),S):-
   holds(shipmentOnBoard(ShNo),S),
   ( (holds(val(recipient(ShNo),Person),S);
        holds(val(sender(ShNo),Person),S)),
      holds(val(robotPlace,mailBox(Person)),S) );
   holds(val(robotPlace, centralOffice),S).

poss(shutdownRobot,_).

poss(cancelPickUp(ShNo,Sender),S) :-
   holds(shipmentToPickUp(ShNo),S), person(Sender).

poss(reduceCredit(Person),S) :-
   holds(val(haveCredit(Person),Credits),S),
   Credits > 0.

poss(resetRobot,S) :-
   not(holds(val(robotStatus,idle),S)).

poss(notifyStopServing(P),_) :-
   person(P).

poss(ackWithRegret(ShNo,Sndr,Rcpt),S):-
   holds(shipmentOrdToAck(ShNo),S),
   holds(val(sender(ShNo),Sndr),S),
   holds(val(recipient(ShNo),Rcpt),S).
```

```
/* For Exogenous Actions */

/* Priority=1 means high priority, Priority=0 means low priority */
poss(orderShipment(Sender,Recipient,Priority),S) :-
   holds(robotAlive,S),
   person(Sender),
   person(Recipient),
   Priority < 2.

poss(msgFromRobot(_),S) :-
   holds(robotAlive,S),
   holds(val(robotStatus, moving),S).



/* *********************************************************************
**                     Successor State Axioms                          *
** *********************************************************************/

/* For normal operations and handling runtime failures */
holds(shipmentOrdToAck(ShNo),do(A,S)):-
   (A = orderShipment(_,_,_), holds(val(shipmentCtr,ShNo),S));
   (holds(shipmentOrdToAck(ShNo),S),
    A \= ackShipmentOrder(ShNo,_,_), A \= ackWithRegret(ShNo,_,_)).

holds(shipmentToPickUp(ShNo),do(A,S)):-
   A = ackShipmentOrder(ShNo,_,_);
   (holds(shipmentToPickUp(ShNo),S),
   (A \= pickUpShipment(ShNo), A \= cancelPickUp(ShNo,_))).

holds(shipmentOnBoard(ShNo),do(A,S)):-
   A = pickUpShipment(ShNo);
   ( holds(shipmentOnBoard(ShNo),S),
     A \= dropOffShipment(ShNo) ).

holds(shipmentDelivered(ShNo),do(A,S)):-
   A = dropOffShipment(ShNo);
   A = ackWithRegret(ShNo,_,_);
   A = cancelPickUp(ShNo,_);
   holds(shipmentDelivered(ShNo),S).

/* Hold iff a client is no longer being served                      */
/* To make the action "cancelPickUp" executes concurrently with      */
/* robotMotionControl.  We need this as a sucessor state axiom as    */
```

```
/* well as haveCredit().                                          */
holds(clientSuspended(Cust),do(A,S)) :-
   A = notifyStopServing(Cust);
   holds(clientSuspended(Cust),S).

holds(readyToBeServed,do(A,S)) :-
   A = orderShipment(_,_,_);
   holds(readyToBeServed,S).

/* Robot is alive if it has not been shut down */
holds(robotAlive,do(A,S)) :-
   holds(robotAlive,S), A \= shutdownRobot.

holds(val(sender(ShNo),Sender),do(A,S)):-
   (A = orderShipment(Sender,_,_), holds(val(shipmentCtr,ShNo),S));
   holds(val(sender(ShNo),Sender),S).

holds(val(recipient(ShNo),Rec),do(A,S)):-
   (A = orderShipment(_,Rec,_), holds(val(shipmentCtr,ShNo),S));
   holds(val(recipient(ShNo),Rec),S).

/* Returns the priority of a shipment */
holds(val(priority(ShNo),Priority),do(A,S)):-
   (A = orderShipment(_,_,Priority), holds(val(shipmentCtr,ShNo),S));
   holds(val(priority(ShNo),Priority),S).

holds(val(shipmentCtr,N),do(A,S)):-
   (A = orderShipment(_,_,_), holds(val(shipmentCtr,M),S), N is M + 1);
   (holds(val(shipmentCtr,N),S), A \= orderShipment(_,_,_)).

/* Returns the current status of robot */
holds(val(robotStatus,Status), do(A,S)):-
   (A = startGoTo(_), Status = moving);
   ( (A = resetRobot; A = msgFromRobot(reached)), Status = idle );
   (A = msgFromRobot(blocked), Status = blocked);
   (holds(val(robotStatus,Status), S),
    A \= startGoTo(_), A \= resetRobot, A \= msgFromRobot(_)).

/* Returns the current position of the robot when not moving */
/* and returns unknown when the robot is moving              */
holds(val(robotPlace,Place),do(A,S)):-
   (A = msgFromRobot(reached), holds(val(approaching,Place1),S),
    Place = Place1);
   (A = startGoTo(_), Place = unknown);
   (holds(val(robotPlace,Place),S),
    A \= msgFromRobot(reached), A \= startGoTo(_)).
```

```
/* Returns the number of credits a person left.        */
/* Credits of a person will decrease by 1 if the robot */
/* is blocked on the way to the person's mailbox.      */
holds(val(haveCredit(Person),Credits),do(A,S)):-
   (A = reduceCredit(Person),
    holds(val(haveCredit(Person),CreditsLeft),S),
    Credits is CreditsLeft - 1);
   (holds(val(haveCredit(Person),Credits),S),
    A \= reduceCredit(Person)).

/* Returns the place to which the robot is heading               */
/* Value changes when the agent starts moving towards another's   */
holds(val(approaching,Place), do(A,S)) :-
   (A = startGoTo(Place1), Place = Place1);
   (A \= startGoTo(_), holds(val(approaching,Place),S)).



/* **************************************************************************
**                            Defined Fluents                              *
** *************************************************************************/

/* For normal operations and handling runtime failures */

/* Holds if Cust requires the services provided by the robot */
holds(needServing(Cust),S) :-
   ( holds(shipmentToPickUp(ShNo),S),
     holds(val(sender(ShNo),Cust),S),
     holds(shipmentDeliverable(ShNo),S),
     holds(shipmentReturnable(ShNo),S) );
   ( holds(shipmentOnBoard(ShNo),S),
     ( holds(val(recipient(ShNo),Cust),S);
       ( holds(val(sender(ShNo),Cust),S),
         not(holds(shipmentDeliverable(ShNo),S))) ),
     not(holds(clientSuspended(Cust),S)) ).

/* Returns 1 if Cust has some higher priority shipments, else returns 0 */
holds(val(haveShptOfPriority(Cust),P),S) :-
   ( holds(val(priority(ShNo),1),S),
     ((holds(shipmentToPickUp(ShNo),S), holds(val(sender(ShNo),Cust),S));
      (holds(shipmentOnBoard(ShNo),S), holds(val(recipient(ShNo),Cust),S))),
     !, P=1 );
   P=0.

/* Holds if Cust1 is the one with the largest amount of credits left */
```

```
/* among all those clients, including Cust1, who have some shipments */
/* of priority P                                                     */
holds(haveHighestCredit(Cust1,P),S) :-
   ( holds(val(haveCredit(Cust1),C1),S), holds(needServing(Cust2),S),
     Cust1 \= Cust2, holds(val(haveShptOfPriority(Cust2),P),S),
     holds(val(haveCredit(Cust2),C2),S), C2 > C1,
     !, not(holds(true,S)) );
   holds(true,S).

/* Holds if P1 is the highest priority amongst all available shipments */
holds(isTheHighestPriority(P1),S) :-
   ( holds(needServing(Cust),S),
     holds(val(haveShptOfPriority(Cust),P2),S),
     P2 > P1, !, not(holds(true,S)) );
   holds(true,S).

/* The best customer to serve is the one with highest priority and largest */
/* amount of credit left amongst those who are of the same priority level  */
holds(bestCustToServe(Cust),S) :-
   holds(needServing(Cust),S),
   holds(val(haveShptOfPriority(Cust),Priority),S),
   holds(isTheHighestPriority(Priority),S),
   holds(haveHighestCredit(Cust,Priority),S).

/* Holds if the robot is serving some higher priority shipment */
holds(servingPriorityShpt,S) :-
   holds(val(approaching,mailBox(Cust)),S),
   ( (holds(shipmentToPickUp(ShNo),S), holds(val(sender(ShNo),Cust),S));
     (holds(shipmentOnBoard(ShNo),S), holds(val(recipient(ShNo),Cust),S)) ),
   holds(val(priority(ShNo),1),S).

/* A shipment is returnable if its sender has not been suspended */
holds(shipmentReturnable(ShNo),S) :-
   holds(val(sender(ShNo),Cust),S), not(holds(clientSuspended(Cust),S)).

/* A shipment is deliverable if its recipient has not been suspended */
holds(shipmentDeliverable(ShNo),S) :-
   holds(val(recipient(ShNo),Cust),S), not(holds(clientSuspended(Cust),S)).



/* **********************************************************************
**                           Non-fluents values                         *
** **********************************************************************/

legalPlace([X,Y,T]):- number(X), number(Y), number(T), !.
```

```
legalPlace(P):- placeValue(P,_).

placeValue([X,Y,T],[X,Y,T]):- number(X), number(Y), number(T), !.
placeValue(mailBox(yves),[9.996250, 33.025938, 0]).
placeValue(mailBox(michael),[10.13000, 19.829689, 0]).
placeValue(mailBox(jeff),[10.630000, 18.780000, 0]).
placeValue(mailBox(hector),[8.406250, 32.825940, 0]).
placeValue(robotHome,[9.010000, 30.973438, 0]).
placeValue(centralOffice,[9.490000, 30.973438, 0]).

person(yves).
person(michael).
person(jeff).
person(hector).



/* ***********************************************************************
**                          Initial States                              *
** ***********************************************************************/

holds(val(robotPlace,Loc),s0) :- Loc = robotHome.
holds(val(shipmentCtr,N),s0) :- N = 1.
holds(val(robotStatus, Status),s0) :- Status = idle.
holds(robotAlive,s0).

/* Change the following to obtain the maximum number of blocking allowed */
holds(val(haveCredit(Person),C),s0):- person(Person),C=2.

holds(val(approaching,Place), s0) :- Place = nowhere.



/* ***********************************************************************
**                          Exogenous Actions                           *
**                       Use for simulation only                        *
** ***********************************************************************/

/* This version relies on the "feedback" from robot         */
/* So, the predicate msgFromRobot() cannot be commented out */
exoDensity(100).

exoAct(orderShipment(yves,jeff,1),1,2).
exoAct(orderShipment(jeff,michael,0),3,4).
exoAct(orderShipment(michael,hector,0),5,6).
exoAct(msgFromRobot(blocked),7,47).
```

```
exoAct(msgFromRobot(reached),48,88).
/* ***********************************************************************
**                            Tracing Controls                           *
** ***********************************************************************/
/*
tracingProg.
tracingTest.
tracingExec.
*/
/* ***********************************************************************
**                    For Global Commit and Roll Forwards                *
** ***********************************************************************/
/* For global commit and roll forward */
fl_auto(shipmentOrdToAck(_)).
fl_auto(shipmentToPickUp(_)).
fl_auto(shipmentOnBoard(_)).
fl_auto(shipmentDelivered(_)).
fl_auto(clientSuspended(_)).
fl_auto(readyToBeServed).
fl_auto(robotAlive).
fl_auto(val(sender(_),_)).
fl_auto(val(recipient(_),_)).
fl_auto(val(priority(_),_)).
fl_auto(val(shipmentCtr,_)).
fl_auto(val(robotStatus,_)).
fl_auto(val(robotPlace,_)).
fl_auto(val(haveCredit(_),_)).
fl_auto(val(approaching,_)).
action_effects(startGoTo,[ robotStatus, robotPlace, approaching ]).
action_effects(ackShipmentOrder,[ shipmentOrdToAck, shipmentToPickUp ]).
action_effects(pickUpShipment,[ shipmentToPickUp, shipmentOnBoard ]).
action_effects(dropOffShipment,[ shipmentOnBoard, shipmentDelivered ]).
action_effects(shutdownRobot, [robotAlive]).
action_effects(cancelPickUp,[ shipmentToPickUp, shipmentDelivered ]).
action_effects(reduceCredit,[ haveCredit ]).
action_effects(resetRobot,[ robotStatus ]).
action_effects(notifyStopServing,[ clientSuspended ]).
```

```
action_effects(ackWithRegret,[ shipmentOrdToAck, shipmentDelivered ]).

action_effects(orderShipment,[ shipmentOrdToAck, sender, recipient, priority,
                               shipmentCtr, readyToBeServed ]).
action_effects(msgFromRobot,[ robotStatus, robotPlace ]).


/* Comment all attached_procedure()'s if not running on Rosie */

attached_procedure(startGoTo, goTo).
attached_procedure(resetRobot, robotabort).

attached_procedure(pickUpShipment, pickUp).
attached_procedure(dropOffShipment, dropOff).

attached_procedure(ackShipmentOrder, ackShpt).
attached_procedure(ackWithRegret, nAckShpt).
attached_procedure(cancelPickUp, cancelShpt).
attached_procedure(notifyStopServing, stopServeUser).

attached_procedure(reduceCredit(Client),no_op).


goTo(Place) :- placeValue(Place,[X,Y,O]),
               ((Place=centralOffice, !,
                   sendMessage(robot,startGoTo(X,Y,O,Place)));
                (Place=mailBox(Name),
                   sendMessage(robot,startGoTo(X,Y,O,Name)))).
robotabort :- sendMessage(robot, abortGoTo).

pickUp(ShNo) :- sendMessage(robot, pickUp(ShNo)).
dropOff(ShNo) :- sendMessage(robot, dropOff(ShNo)).

ackShpt(ShNo,Sender,_) :- sendMessage(Sender,acknowledge(ShNo)).
nAckShpt(ShNo,Sender,_) :- sendMessage(Sender,ackWithRegret(ShNo)).
cancelShpt(ShNo,Sender) :- sendMessage(Sender, shipmentCancelled(ShNo)).
stopServeUser(X) :- sendMessage(X,notifyStopServing(X)).



/* **********************************************************************
**                        Procedure Definitions                         *
** *********************************************************************/

/* Acknowledges a shipment if there are any */
proc(acknowledgeShipmentOrder(ShNo),
```

```
   pi([sndr,rcpt], [ and(val(sender(ShNo),sndr),val(recipient(ShNo),rcpt))?,
        if(and(shipmentReturnable(ShNo),shipmentDeliverable(ShNo)),
            [ ackShipmentOrder(ShNo,sndr,rcpt),
              globalCommit,
              if(and(val(priority(ShNo),1),
                     and(val(robotStatus,moving),not(servingPriorityShpt))),
                    [ resetRobot, globalCommit ],
                    no_op
              ) ],
            [ ackWithRegret(ShNo,sndr,rcpt), globalCommit ]
        )
   ] )
).

/* Top level control of the robot: Serves a client if there are any,   */
/* otherwise, goes to central office and drops off all remaining       */
/* shipment onboard.  And shuts down the robot.                        */
proc(robotMotionControl,
   if(some(cust, needServing(cust)),
         serveCustomer,
         [ startGoTo(centralOffice),
           globalCommit,
           not(val(robotStatus,moving))?,
           if(val(robotPlace,centralOffice),
                 [ dropOffToCentralOffice,
                   if(some(cust2, needServing(cust2)),
                      no_op,
                      [ shutdownRobot,globalCommit ] ) ],
                 if(val(robotStatus,blocked),
                       [ resetRobot, globalCommit ],
                       no_op
                 )
           ) ]
   )
).

/* Defines the procedure of what to do when a customer who requires */
/* serving is identified: Provides service if arrived at the        */
/* destination, otherwise, calls procedure 'handleGoToFailure'      */
proc(serveCustomer,
   pi(cust, [ bestCustToServe(cust)?,
         [ startGoTo(mailBox(cust)),
           globalCommit,
           not(val(robotStatus,moving))?,
           if(val(robotPlace,mailBox(cust)),
                 [ pickUpShipmentsFrom(cust),
```

```
        dropOffShipmentsTo(cust) ],
                 if(val(robotStatus,blocked),
                       handleGoToFailure(cust),
                       no_op
                 )
           ) ]
   ] )
).

/* Defines the procedure of what to do when blocked */
proc(handleGoToFailure(Cust),
   [ reduceCredit(Cust),
     if(val(haveCredit(Cust),0),
          notifyStopServing(Cust),
          no_op),
     [ resetRobot, globalCommit ] ]
).

/* Drops off all onboard invalid shipments to central office */
proc(dropOffToCentralOffice,
   while(some(shNo, and(shipmentOnBoard(shNo),
                        not(or(shipmentDeliverable(shNo),
                               shipmentReturnable(shNo))))),
       pi(shNo, [and(shipmentOnBoard(shNo),
                     not(or(shipmentDeliverable(shNo),
                            shipmentReturnable(shNo))))?,
          [ dropOffShipment(shNo), globalCommit ] ]))).


/* Picks up all shipments from Cust */
proc(pickUpShipmentsFrom(Cust),
   while(some(shNo, and(shipmentToPickUp(shNo),val(sender(shNo),Cust))),
      pi(shNo, [ and(shipmentToPickUp(shNo),val(sender(shNo),Cust))?,
         [ pickUpShipment(shNo), globalCommit ] ]))).


/* Drops off all shipments to Cust */
proc(dropOffShipmentsTo(Cust),
   while(some(shNo, and(shipmentOnBoard(shNo),
                        or(val(recipient(shNo),Cust),
                           and(val(sender(shNo),Cust),
                               not(shipmentDeliverable(shNo)))))),
      pi(shNo, [ and(shipmentOnBoard(shNo),
                     or(val(recipient(shNo),Cust),
                        and(val(sender(shNo),Cust),
                            not(shipmentDeliverable(shNo)))))?,
```

```
          [ dropOffShipment(shNo), globalCommit ] ])).



/* *********************************************************************
**                            Main Program                            *
** *********************************************************************/

runSystem(Ini,Final) :-
   pdo(

   /* Acknowledges shipment order if there are any */
   (==>([shNo], shipmentOrdToAck(shNo),
           acknowledgeShipmentOrder(shNo)))

   #=
   /* Cancels pick up if there are any invalid shipment waiting for pickup */
   (==>([shNo,sender], and(and(shipmentToPickUp(shNo),
                               val(sender(shNo),sender)),
                           not(and(shipmentReturnable(shNo),
                               shipmentDeliverable(shNo)))),
           [ cancelPickUp(shNo,sender), globalCommit ] ))

   #>
   /* Main Loop to serve a client */
   (==>([shNo], and(readyToBeServed,
                    or(or(shipmentToPickUp(shNo), shipmentOnBoard(shNo)),
                       not(val(robotPlace,centralOffice)))),
           robotMotionControl))

   #>
   /* Skips if the robot is moving */
   or(not(readyToBeServed), robotAlive) ==> no_op,

   Ini,Final).


run0 :- setup_tcpip,
        runSystem(s0,_),
        retire.

```