Topics in High-Level Robot Control: Integrating Planning and Reactivity, and Multiple-Robot Coordination

Ho-Kong Ng

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

Master of Science

Thesis Supervisors: **Yves Lespérance and Michael Jenkin**

Graduate Programme in Computer Science
York University
Ontario, Canada

July 2001

Abstract

It is difficult to build an autonomous robot controller that can control a robot to perform complex tasks. It should not only generate plans for the robot to accomplish tasks, but also be able to handle unexpected events and run-time failures with incomplete knowledge of the environment. This thesis presents a robust approach for constructing high-level robot control programs using the agent-oriented programming language IndiGolog. Its predecessor ConGolog has been shown to be an effective programming language for constructing reactive and deliberative agent programs. By improving the planning and plan execution mechanisms in IndiGolog, planning and reactivity can be integrated effectively to optimize the actions performed by the robot while maintaining robustness in a dynamic environment. Our approach is also applied on a real shipment delivery application with multiple robots, to demonstrate how one can build a distributed control system to coordinate multiple robots to accomplish a task using IndiGolog.

Acknowledgments

There are many people to thank for their support and encouragement, without them this thesis would not have been possible.

Firstly, I would like to thank my first thesis supervisor, Prof. Yves Lespérance, for his brilliant advice and guidance throughout the course of this work, and help in bringing this thesis to completion. I am also grateful to Prof. Michael Jenkin for great advice and support on building the robot system.

I would like to thank my committee members Prof. James Elder and Prof. Mokhtar Aboelaze, for their helpful discussions and suggestions.

Many thanks to Bill Kapralos, Rajesh Radhakrishnan, Arlene Ripsman, Victor Wong, Kaiqi Yu, and all the friends I have made in York University, for their help and encouragement.

Finally, I wish to thank my parents, brother and Lumilla for their considerable encouragement, support and patience, especially when I claimed my thesis would be finished in a few months for nearly three years.

Contents

Chapter 1

Introduction

When executing complex tasks in a real environment, a mobile robot may need to perform many actions to accomplish its goals, as well as monitor relevant conditions in the environment and react to problems that occur at run-time. It is very important that the robot should plan the actions it should perform to reach its goals quickly and correctly.

This thesis presents an approach to model a mobile robot's environment and capabilities and implement high-level control programs which combine planning and reactivity, for robots working in dynamic environment. This kind of controller can reason about the actions that should be performed, generate plans to achieve goals in different situations, and adjust them according to changes in the environment. We also examine how the approach can be used to coordinate multiple robots.

1.1 The Problems that Will Be Addressed

1.1.1 Planning in Dynamic and Incompletely Known Environment

A plan can be viewed as a sequence of actions the robot can perform to accomplish its task. Planning is useful to optimize the performance of a robot when its tasks are complicated and to deal with unexpected goals or situations. But this can be difficult if some of the necessary information can only be obtained at run-time or the environment can change. Suppose that a robot has to move several boxes to different locations inside a building. It has to construct a route for moving all boxes that requires the least amount of time and power. However, the robot cannot know what will happen on its way to each location in advance. It may not be able to reach a place because of a locked door. Also, the destination of a box may only be known after the box has been picked up. In either case, the robot must be able to construct a plan at the beginning, and adjust it according to what happens at execution time. Without the ability to plan with incomplete information and adjust the plans to deal with environmental changes at run-time, the robot may not able to work efficiently or even accomplish its task at all.

1.1.2 Combining Reactive and Planned Behaviors

A robot must not only perform a specific sequence of actions to accomplish its task. It must also sense for changes in the environment and react immediately to avoid running into problems and doing harm to itself or people, and to take advantage of opportunities. The robot controller should be able to plan and determine what actions it should perform to achieve its goal, and also be able to monitor the environment for changes and to react to handle different problems that arise. For example, the controller of a security guard robot must generate routes for the robot to patrol in the building. It must also keeps sensing to determine if there is any intruder, smoke, gas leak, etc. in the surrounding area. One can build a robot controller by combining everything into a single control thread that does both route planning and event monitoring. However, developing appropriate plans and reasoning about such a complex task can be quite complicated and time consuming. The robot may spend most of its time on planning and not react fast enough to avoid problems. Planning and reactivity are best done by different control threads, but how they interact with each other then becomes an important issue.

1.1.3 Coordinating Multiple Robots to Accomplish a Task

How to coordinate multiple robots to accomplish a task is a difficult problem. For example, in serving tables in a restaurant, two robots should not try to serve the same table. If possible, the nearest robot to the table should be chosen as the server. In addition, the work must be distributed to all robots evenly so that a robot does not have to serve too many tables while another robot has nothing to do. A centralized control system can collect the necessary information from every robot and generate plans to control all of them, but then every significant piece of information each robot obtains must be sent back to the central system and the search space of planning may becomes huge. A large amount of time may be spent on communication and planning instead of performing actions to accomplish the task. The system may even fail if the communication link to one of the robots is defective. The plans for controlling multiple robots that must be generated will be large and complex. A better approach is to use a distributed system where each robot has its own planner and they cooperate by communicating with each other.

1.2 Our Approach

To control robots in a dynamic environment, our approach is to construct robot controllers as high-level agent control programs which combine planning and reactivity, while the physical motion of the robot is governed by a low-level module which can perform very fast reactions to avoid collisions and deal with other emergencies. Our control programs are written in an agent programming language called IndiGolog [10]. We have made several enhancements to this language so that planning, reactivity, and sensing can be integrated effectively in the control programs. Each program contains a planning component that generates plans to optimize the actions of the robot in order to achieve the goal, and other reactive components for responding to changes in the environment at the same time. We have also developed meta-level planning and execution facilities for programs to have more control on plan generation and execution. We also explore how multiple robots can cooperate with each other in our framework by developing a distributed framework with a bid-for-order mechanism [60] to accomplish a delivery task.

1.3 Research Objectives

The main research objectives of this thesis are as follows:

- To develop an adequate programming language for building high-level robot controllers which combine planning, sensing, and reactivity. It should support handling run-time failures and changes in the environment, and also perform replanning when it is needed.
- To test our robot programming language on a real robot in a real environment.
- To extend our approach to handle multiple-robot applications and show they can be delivered efficiently with the robots cooperating by communicating with each other.

1.4 Outline of the Thesis

In Chapter 2, we discuss related work in robot architecture and high-level robot control with planning and reactivity. In Chapter 3, we present the logical foundations of situation calculus and the components of the Indigolog programming language. In Chapter 4, we describe the enhancements we have made to Indigolog to support planning, replanning, and reactivity. Chapter 5 presents some meta-level facilities for the program to have access to and control of the plan generation and execution process. In Chapter 6, we show how IndiGolog can be used to program multiple robots that cooperate on a shipment delivery task; we develop an efficient and robust distributed system architecture for this. Chapter 7 describes the internal architecture of our implemented robot control system, and presents the tests that have been performed on this system. In Chapter 8, we give our conclusions regarding this work.

Chapter 2

A Review of Agent Architectures and Programming Languages

In a classical intelligent autonomous robot system, there is a distinction between planning and execution, in which the system first constructs a plan and then executes all the actions in the plan. The first problem with this kind of structure is that the environment of the robot is usually dynamic. What the robot assumes to be true during planning may become false as a result of the actions of other agents during execution. By arranging planning and execution in sequence, the robot may run into the risk of constructing a plan that is not valid because changes in the environment later invalidate some preconditions. Another problem with this approach is that the robot may not have complete information about its environment and cannot always decide which action should be performed without getting additional data at run-time.

This chapter provides a brief introduction to some of the architectures and reasoning frameworks that have been proposed for robots which work in dynamic environments.

2.1 Deliberative Architectures

The most obvious approach to building an intelligent robot controller is to specify all the actions that the robot can perform, and build a planner which can generate a sequence of actions to achieve the robot's goals automatically. Genesereth and Nilsson [23] introduced the term *deliberative agent*, where a robot is treated as an agent who can plan and perform actions in an environment that is represented in some symbolic model. Objects in the environment are represented by symbols and their relationships are modeled by symbolic rules. The agent takes an initial description of the environment, a list of rules which specifies the actions available to change the environment, and then it decides which actions in what order should be performed to achieve the goal by using pattern matching and symbolic manipulation. This is somewhat similar to a logic theorem prover, where a formula to be proved and some symbolic rules and axioms are given and the system has to derive the formula from the rules based on logic.

The *Shakey* robot [44] built by Nilsson was constructed using this deliberative idea. Its main component is a plan generation system called $STRIPS$ [44]. It has a symbolic representation of both the robot's world and the desired goal state, and a set of operators/actions that the robot can perform with their preconditions and effects. See Figure 2.1 for a simple example for moving a robot to a storage room. The system attempts to construct a sequence of operations that will achieve the goal by using a simple means-ends analysis, which involves matching the effects of the operations against the goal (regression planning). Although the STRIPS planning algorithm is very simple, a linearity assumption is embodied in its means-ends analysis and as a result it cannot find the optimal solution or even any solution to problems such as the *Sussman anomaly.* STRIPS assumes that its world model is complete (the closed world assumption) and represents the effects of actions by lists of atomic formulas to be added and deleted from the world model. This provides a restricted but quite efficient solution to the frame problem [50] (i.e. representing what is not affected by the actions).

The type of reasoning that is used in most of the deliberative approaches is commonly referred to the *Sense-Model-Plan-Act* framework. The robot first obtains information from its environment through sensing. Next, it constructs a model of the world based on the perceptual data. Then, it generates a plan for accomplishing its tasks; and finally, it performs the actions in the plan to accomplish the task. The advantage of this approach is that the robot can deal with arbitrary goals and automatically figure out a plan or sequence of actions for accomplishing the goal. This works reasonably well in small domains where the robot is the only one who can change the world. However, it often fails in a dynamic environment where the state of the environment can be changed by other agents during the execution of the plan. Also, it assumes that all necessary information about the environment is available. However, sensor data may not be available or accurate and it is often impossible to sense every aspect of the world to construct a complete world model. Even if it can build such a model, many symbol manipulation algorithms that can be used for theorem proving are intractable and time-consuming. The robot may not be able to generate a plan in this way before the environment changes to another state.

2.2 Reactive Architectures

In contrast to the deliberative approach, the reactive approach aims for constructing autonomous robot controllers that do not maintain complex models of the world and can act quickly without planning. Basically, a reactive architecture does not contain a symbolic model of the world and a reasoner for manipulating rules and finding plans. Instead, it consists of a set task-oriented behaviors that examine sensor data and then immediately make decisions on which action should be performed. It focuses on reacting to changes in the environment quickly rather than wasting time on doing complex reasoning.

The approach was first popularized by Brooks with his *Subsumption Architecture* [5, 6]. It can be viewed as a control system which consists of a hierarchy of task-achieving behaviors. Each layer in the hierarchy is a simple state machine which represents a behavior of the robot for accomplishing a specific task. Each layer is completely independent from the others and all of them run in parallel. As a result, the robot is capable of reacting immediately when an appropriate layer detects a new condition in the robot's environment. The layers are arranged hierarchically according to their levels of competence. Figure 2.2 shows an example of the architecture for a mobile robot controller. A higher level layer, such as building plans in the example, can take control of the robot by subsuming the execution of a lower level layer, such as moving to a location, which was controlling it. In this architecture there is no explicit representation of the robot's goals or knowledge of the world. Each layer decides which action should be performed to change the environment based on only the data returned from the sensors and its internal state. This is the ideal reactive architecture because it only maps the sensory inputs to the robot's actuators and does not require any reasoning.

A major issue in reactive architectures is behavior arbitration, that is, deciding which behavior should run at any given moment. For example, if one is not careful, it can easily get into a situation where the robot keeps switching between two behaviors and never accomplishes anything useful.

Similar to the structure of layers in the subsumption architecture, the *Agent Network Architecture* [38] proposed by Maes treats a robot as a set of competence modules. The designer of the robot controller specifies the function of each module in terms of preconditions and postconditions and a priority value to indicate the relevance of the module in a particular situation. Before the robot can execute, all the modules are linked together to form a spreading activation network according to their preconditions and postconditions. This allows the robot to react quickly and several modules can be activated at a particular situation for completing several tasks at the same time. Performing an execution step may either cause the robot to execute an action, or change the relevance level of a module.

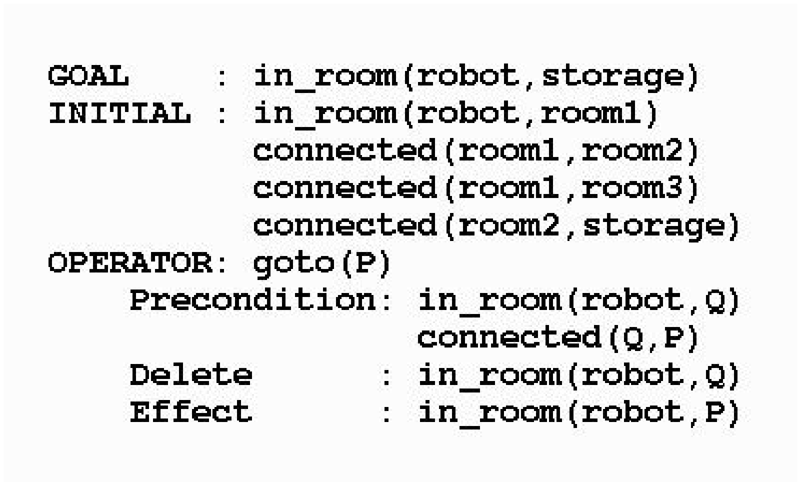
The *Concrete-Situated* architecture [1] proposed by Agre and Chapman is another example of a reactive architecture. They argue that the world can be characterized as complex, uncertain, and immediate, and that the robot can simply react to the world and does not need to maintain a model of the world or store any state in memory. They also argue that the world is actually the best model for itself and information can be gathered directly from it. This architecture consists of patterns of behavior, which are called *routines*. Routines are implemented on low-level structures such as digital circuits and do not require any explicit world model or symbolic manipulation. The robot can interact with the environment according to the routines it has without constructing any plan. This idea was applied to build the software agent PENGI [1] for playing a computer game. It neither has any internal memory nor makes any plans. It uses only the information it gets from the environment during execution as the inputs to its routines to determine the actions that should be performed.

The *situated automata* framework [53, 54] developed by Rosenschein and Kaelbling models the robot's operation in terms of a perception and an action module, and provides tools for doing off-line reasoning to synthesize reactive controllers. The perception component of the robot is specified using a tool called RULER, while the action module is implemented using GAPPS. The behaviors of the robot are generated from a set of goal reduction rules and a list of prioritized goals. The rules describe how the actions achieve the goals. GAPPS translates this information into specifications that can be further simplified into digital circuits. The process of converting the programs into digital circuits is done at compile time. Therefore, the robot does not have to manipulate any symbolic representation of the world.

An important characteristic of the robot controllers built under these reactive approaches is that they do not contain an explicit world model or planner for doing complex reasoning. Reasoning and decision making are distributed into various components in the system. Since they are hard-coded or compiled before the robot's execution, the controllers are only required to sense the environment and then instruct the robot how to act afterwards. The main problems with this approach are that it is hard to specify complex robot behaviors and to deal with unanticipated situation. Without a model of the world and explicit reasoning, they cannot be used for complex tasks in which search and planning are required. They cannot be guaranted to pick the best action for the robot and handle unexpected situations. The robot may keep reacting to the changes in the environment and may not do any action to achieve its goals.

2.3 Hybrid Architectures

For applications where the robot has to reason about the world to make a decision and react quickly to changes in the environment, an obvious alternative is to combine the deliberative approach and the reactive approach. A hybrid architecture is defined as a system that consists of a deliberative component and a reactive component. The deliberative component keeps an explicit world model and makes decisions on action to achieve the goal by reasoning and planning, while the reactive component provides quick response to changes in the environment without doing any complex reasoning.



```
GOAL     : in_room(robot,storage)
INITIAL  : in_room(robot,room1)
           connected(room1,room2)
           connected(room1,room3)
           connected(room2,storage)
OPERATOR: goto(P)
    Precondition: in_room(robot,Q)
                  connected(Q,P)
    Delete      : in_room(robot,Q)
    Effect      : in_room(robot,P)
```

Figure 2.1: A STRIPS Example: Moving a Robot to the Storage Room

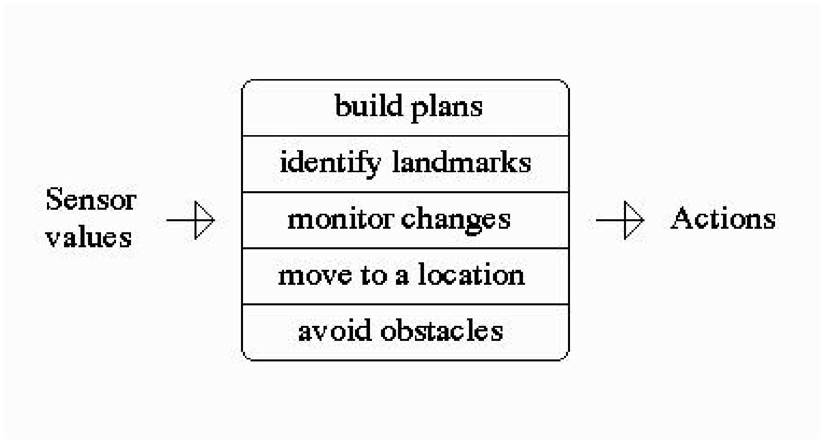


Figure 2.2: Layers of Task-Achieving Behaviors in a Subsumption Architecture

A lot of work has been done in this area to argue that a combination of deliberative and reactive approaches can compensate for the limitations of each of them, but how they should be integrated to give the most efficient control is still not known. Work in this area also looks at how continuous low-level control should interface with discrete high-level control.

One simple approach for combining the two components is to build a planner that generates plans and a separate executor which selects the most appropriate plan to follow for achieving the goal or reacting to changes in the environment. An example of this is the *Reactive Action Packages* (RAP) framework proposed by Firby [18], which is designed for reactive execution of symbolic plans. The RAP architecture consists of three components: a controller, a planner, and a RAP executor. The controller contains low-level control and sensing modules to control the effectors and sensors of the robot. Decision making is handled by the planner, where a world model is maintained for reasoning. A plan generated by the planner consists of tasks and the RAP executor tries to carry out each task in turn using different methods. Each method can enable or disable a set of control and sensing modules in the controller for the robot to achieve a particular task. In each cycle, a task is first selected to be executed by the executor. If the task is only a primitive action and it can be performed in the current situation, then the executor executes the action directly. Otherwise, the executor checks whether the task is already satisfied in the current situation. To accomplish the task, a method from the RAP library is chosen such that it can be used to complete the task. The task is then suspensed and the executor tries to complete the sub-tasks of the method. When all sub-tasks have been executed, the task is reactivated and the executor checks again if it has been satisfied in the current situation again. If this is true then the executor can proceed to the next task. Otherwise, another method is chosen to try to accomplish the task.

Another well-known framework in this category is the *ATLANTIS* architecture [22] proposed by Gat. It has a reactive control system and a traditional planning system which run in parallel. It is designed for robots that are operating in dynamic environment without complete information. Basically, it consists of three components. The controller is a reactive control system which is composed of behavior producing modules called primitive activities. Each primitive activities can enable or disable a set of effectors and sensors of the robot. The sequencer is the operating system which controls the initiation and termination of primitive activities in the controller. The deliberator performs the deliberative time-intensive planning and world modeling. The plan generated by it is used only as advice by the sequencer and the sequencer may drop a plan or switch to another plan depending on state of the environment which can be changed by unpredictable events. It also contains a time-out mechanism to prevent the controller from spending too much time on accomplishing a task. Since the deliberative component and the reactive component are executed asynchronously, the system is able to distribute its time and resources on planning and reacting to changes in the environment.

Another example that is based on similar approach is the *Intelligent Resource-Bounded Machine Architecture* (IRMA) [4] developed by Bratman et al. It focuses on modelling the beliefs, desires, and intentions of the agent and how much it should commit to particular intentions/plans. These mental attitudes are modeled by four kinds of symbolic data structures. There is a plan library that provides predefined solutions for handling different simple tasks, a reasoner which can be used to reason about the world, a plan analyzer that contains methods for comparing the available plans and find out which is the best for achieving the robot's intentions, and a filter process that determines which possible plans are consistent with the robot's existing intentions.

For the case where one must control more than one robot, Jennings also proposed the $GRATE$ [33] framework which is based on not only on beliefs, desires, and intentions, but also joint intentions. These kinds of attitudes are represented in standard if-then format. Each agent has two components: a domain level layer that tries to solve problems for the individual agents, and a cooperation and control layer to ensure that each agent's domain level activities are coordinated with those of others in the community.

The *Theo* [40] architecture proposed by Mitchell integrates planning, reacting, and knowledge compilation. Each agent in Theo has a set of stimulus response rules which allows it to respond to sensory data rapidly in a dynamic environment. It stores its knowledge in a frame representation. A frame which has a slot and a value is said to be a belief. An *impass* occurs when there is a slot without a value, and it triggers the system to attempt to infer the slot's value. Since the system works only when an impass occurs in which some answer is required, it focuses on only one behavior at a time.

Based on the prioritization of the rules it has, it can handle multiple goals simultaneously by suspending a goal and selecting to work on another one which has the highest priority. However, it is not clear whether the priorities of the rules can be changed at execution time to support interleaving of multiple plans for several goals.

Another common approach to combine deliberative and reactive components is to modify the planner to generate plans which contain branches for different possible situations that may appear during execution. The *Procedural Reasoning System* (PRS) [25] proposed by Georgeff integrates reactive reasoning and planning at this level. It is quite similar to RAP with hierarchical plan expansion at run-time, but has a more systematic approach to plan reconsideration. It has a set of facts as its beliefs, a set of goals, and also a library of alternative plans for the agent to accomplish different goals. A plan can be both partial and hierarchical. It is represented as a rule with a condition and a body. The body is a sequence of sub-goals to be pursued to bring the robot to the goal, while the condition is the situation in which the body should be triggered. In the course of chosing a plan, new beliefs may be derived. This may result in a situation where another plan is more appropriate than the current one, and the system then selects the new one for execution. As a result, the plan that is being executed is always relevant to the current situation. In addition, it can reconsider the current plan, suspend the current plan and/or adopt new plans during execution if more than one task must be handled at the same time or an unexpected change in the environment occurs. With these features, the robot controller can reason about its world as well as react to run-time problems.

Generally, there are two kinds of architectures among the hybrid approaches. The first is to build the whole system as a single control loop and implement the reasoning component as a part inside the loop. In each cycle, the task of the robot can be broken down into sub-tasks to support reactivity or deliberative planning according to the perceptual data and the state of the robot. Architectures such as RAP, PRS and Theo are built in this way. The other alternative is to have a deliberative component and a separate reactive component operating asynchronously. In this way the deliberative component can reason about the world and find the best way to achieve the robot's goal while the reactive component can control the robot to avoid running into problems at the same time. Examples of this are ATLANTIS and InteRRAP [41, 42]. However, synchronizing the components can be a problem in this approach.

In our work, we use a hybrid layered architecture similar to these. It includes a low-level reactive layer which controls the robot to perform fast reaction to avoid critical problems such as crashing into a wall or hitting an human, and also a high-level deliberative and reactive layer which runs in parallel with the low-level component. The high-level layer has a model of the robot's world in order to perform planning and reasoning and also reactive capabilities to change the robot's behavior when there is a change in the environment which affects the robot's intentions or the details of the robot's tasks. In other frameworks, the high-level reasoner and the low-level reactor are built and integrated by focusing on specific task. However, adapting these controllers to more complex tasks or other applications may require much effort. We built our high-level layer by using a high-level agent programming framework, which makes modification of the robot controller much easier. The next section describes some of the existing agent programming languages for implementing robot control programs and in Chapter 3, we describe our programming framework and how robot control can be specified in it.

2.4 Agent Programming Languages

There are many agent programming languages that have been proposed for designing agents or robots which operate in dynamic environments. They uses different approaches to represent the mental states of an agent and state of the world, and specify the agent's behavior. One example of this is the *Agent-0* language [56, 57] proposed by Shoham which uses the notions of beliefs, commitments, and capabilities. An agent is specified as a set of logical rules for entering into commitments and executing capabilities, in terms of a set of beliefs. In addition, the language supports some simple communicative actions for exchanging messages between agents. Messages that can be sent are either inform, request, unrequest, or refrain messages. Inform messages are used to communicate the beliefs of the agent, while the other three kinds of messages are provided for changing commitments. The *Agent-K* extension [8] allows many commitments to match a single message. This was not allowed in Agent-0, and that interpreter simply selected the first rule that matched a message. However, the limitation of this approach is that commitments can be made only to execute primitive actions. The agent has no mechanism to specify complex actions for achieving goals. The *Planning Communicating Agents* (PLACA) extension proposed by Thomas

[62] attempts to solve this problem by allowing the programs to include operators for planning to achieve goals. However, the relationship between the logic and interpreted programming language is still loosely defined.

Another example of an agent programming language is $METATEM$ [3], which was proposed by Barringer and Fisher. It models the world and the behavior of the agent using temporal logic. A logical model of each component is constructed incrementally in execution, and so the dynamic attributes of it can be concisely represented. Its extension, $Concurrent\ METATEM$ [20, 19], allows several agents to run concurrently and uses a broadcast message passing mechanism to allow them to communicate with each other for distributed applications.

The need for reasoning and the possibility of direct execution of agent specifications specified in logic have led to a number of attempts to build executable logic-based agent programming languages. The *AgentSpeak*(L) language proposed by Rao [49] is a formally specified language based on the Procedural Reasoning System architecture discussed in the previous section. To compare and evaluate different languages, Hindriks and his colleagues developed the *3APL* programming language [29], which has been formally specified by providing a Plotkin style structured operational semantics and a formal specification in the Z language. Several papers [29, 28, 30] showed that these languages are closely related.

A limitation in many of these agent programming languages is that the programs are expressed in terms of some sort of if-then rules. By following these rules, one may be able to realize that the agent will eventually achieve its goal, but not when or by what means. It is not easy for one to analyze and reason about the behavior of the agent in detailed way. Also, most of them do not support true planning, only the dynamic selection of prewritten plans from a library. *IndiGolog* [10] is also an agent programming language for modelling agent behavior declaratively. It is the latest version in the *Golog* family [37, 13, 12], which is based on the *situation calculus* [39], a logic formalism for reasoning about actions. In [61], *ConGolog*, an extension of Golog, was used for implementing robust robot control modules. IndiGolog supports ConGolog's features such as concurrency, priorities, interrupts, and exogenous actions handling, and also provides additional facilities on performing on-line planning, sensing, and incremental execution. We believe that these are essential for building robot control program for complex tasks. We will use this language in this thesis and present it in detail in Chapter 3.

Chapter 3

IndiGolog — Agent Programming in the Situation Calculus

In this chapter, we present the IndiGolog agent programming framework that we will use in this thesis. We first describe the logical foundation of IndiGolog, which is the situation calculus. Then, we present the agent programming languages Golog, ConGolog, and IndiGolog.

3.1 The Situation Calculus

The *situation calculus* [39] is a first-order language (with some second-order axioms) that can be used to represent dynamic worlds. It models the world in the way that all changes to the world are caused by the execution of some actions. At any moment, the world is in some state that is represented by a *situation* term. A situation is viewed as the sequence of actions that has led to it. There is a special situation, the initial situation S_0, which represents the situation when no action has yet occurred. In addition, there is a binary function $do(a, s)$ that denotes the successor situation of the situation s after executing the action a. There are two kinds of fluents for describing what is true in a situation. A predicate fluent $f(\vec{x}, s)$ denotes that fluent f with the parameters $\vec{x}$ is true in situation s. A functional fluent $f(\vec{x}, s) = c$ indicates that the value of fluent f with the parameters $\vec{x}$ in situation s is c. For example,

$$newOrder(n, do(orderShipment(n), S_0))$$

means that there is a new order for shipment n in situation $do(orderShipment(n), S_0)$, and

$$robotPos(do(goto(storage), S_0)) = storage$$

states that the position of the robot is in the storage room in the situation $do(goto(storage), S_0)$.

The *Golog* framework that we will use in this thesis is based on Reiter's version of the situation calculus [50]. In it, there is also a special predicate $Poss(a, s)$ that means it is possible to perform action a in situation s. A dynamic world is specified by a logical theory that includes the foundational axioms for the situation calculus given in [50], action precondition axioms, successor state axioms, initial state axioms, and unique names axioms. Brief descriptions of these are given in the following sub-sections.

3.1.1 Action Precondition Axioms

Action precondition axioms describe all the important preconditions or requirements that must be satisfied for it to be possible to execute an action. Each action has its own precondition axiom. Action precondition axioms have the following format:

$$Poss(A(\vec{x}), s) \equiv \pi_A(\vec{x}, s)$$

where $A(\vec{x})$ is an action with the parameters $\vec{x}$, s is a situation and $\pi_A(\vec{x}, s)$ is a first-order formula that is uniform in s [50] (i.e. it only talks about what is true in situation s), and this specifies the necessary and sufficient conditions that must be satisfied in order to perform the action.

For example, a robot for shipment delivery can pick up a shipment if it is where the shipment is being sent from, and it can drop off a shipment if the shipment is on board. These can be represented by the following action precondition axioms:

$$Poss(pickUp(n), s) \equiv shipmentPos(n, s) = robotPos(s)$$
$$Poss(dropOff(n), s) \equiv shipmentPos(n, s) = onBoard$$

where n is a shipment number, and $pickUp(n)$ and $dropOff(n)$ are the primitive actions for the robot to pick up and drop off shipment n.

3.1.2 Successor State Axioms

Successor state axioms are used to specify how the world can change. There is one successor state axiom for each fluent that specifies how the value of the fluent changes when some action is performed. For a predicate fluent F, we have a successor state axiom of the form:

$$F(\vec{x}, do(a,s)) \equiv \gamma^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma^-(\vec{x}, a, s)$$

where $\vec{x}$ are the parameters of F, a is an action, and s is a situation. γ^+ is a first-order formula that is uniform in s and represents all the conditions for which when a is performed, F will become true. On the other hand, γ^- is a first-order formula that is uniform in s and specifies all the conditions for which when a is performed, F will become false. Essentially, this axiom says that fluent F is true in the situation $do(a,s)$, provided that either doing a in s makes F become true, or F was already true in situation s and doing a in s does not make F become false. For a functional fluent f, we have a successor state axiom of the form:

$$f(\vec{x}, do(a,s)) = c \equiv \gamma^+(\vec{x}, a, s) \vee f(\vec{x}, s) = c \wedge \neg\gamma^-(\vec{x}, a, s)$$

For example, we might say that a robot can move to a place p by performing the $goTo(p)$ action. How the position of the robot, $robotPos$, changes is specified by the following successor state axiom:

$$\begin{aligned}&robotPos(do(a,s)) = p \equiv \\ &\qquad a = goTo(p) \vee ((\forall\, p')\ a \neq goTo(p') \wedge robotPos(s) = p)\end{aligned}$$

This says that the robot is at position p in situation $do(a,s)$ if the robot has just performed the action $goTo(p)$, or the action it performed in situation s was not $goTo$ and it was at p in situation s. Here is another example:

$$\begin{aligned}&newOrder(n, do(a,s)) \equiv \\ &\qquad a = orderShipment(n) \vee (a \neq acknowledge(n) \wedge newOrder(n,s))\end{aligned}$$

This says that there is a new shipment order for shipment n in situation $do(a, s)$ if and only if someone has just made a new shipment order for shipment n in situation s, or the robot did not acknowledge shipment n and there was a new shipment order for n in situation s.

Successor state axioms provide a solution to the *frame problem* [50]. To axiomatize dynamic worlds, one must specify all effects and invariants of each action. The frame axioms specify the fluents that would not be affected by performing an action. However, since most actions have no effect on a particular fluent, there is a huge number of these axioms in a given domain. Reasoning in the presence of so many axioms may be very inefficient and resource-consuming. Successor state axioms solve this problem by quantifying over actions, all actions that are not explicity mentioned are taken to leave the fluent unchanged.

3.1.3 Initial State Axioms

Initial state axioms specify the values of the fluents in the initial situation S_0, that is, the initial state of the world before any action is executed. They can be any formulas that are uniform in S_0.

For example, we can say the robot is at the *home* position in the initial situation with the following initial state axiom:

$robotPos(S_0) = home$

3.2 Golog

Golog is an agent-oriented programming language [37] based on the situation calculus. It uses action precondition axioms and successor state axioms to model the preconditions and effects of primitive actions on the world, and also allows the user to define complex behaviors for the agent by providing a full range of programming constructs, which are all defined in the situation calculus.

A Golog program consists of one or more of the following constructs:

a	primitive action
$c?$	testing of a condition
$(\delta 1; \delta 2)$	sequence of actions
$(\delta 1 \vert \delta 2)$	nondeterministic choice between actions
$(\pi x)\ \delta$	nondeterministic choice of arguments
δ^*	nondeterministic iteration
$if\ c\ then\ \delta 1\ else\ \delta 2$	conditional statement
$while\ c\ do\ \delta$	while loop
$proc\ p(\vec{x})\ \delta\ end$	procedure definition
$p(\vec{t})$	procedure call

3.2.1 Classical Planning and Golog Programming

To understand the idea behind Golog, let us compare it with classical planning. Given a set of axioms which describes the agent's domain and a goal $Goal(s)$ with the free variable s, the task of classical planning is to find a sequence of actions σ such that it is executable, and leads to the goal being achieved when executed. It can be specified in the situation calculus as follows:

$$Axioms \models Legal(\sigma, S_0) \wedge Goal(do(\sigma, S_0))$$

where $do(\sigma, S_0)$ is an abbreviation for

$$do(a_n, do(a_{n-1}, ..., do(a_1, S_0))...)$$

and $Legal(\sigma, S_0)$ stands for

$$Poss(a_1, S_0) \wedge Poss(a_2, S_0) \wedge \ldots \wedge Poss(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)) \ldots)$$

However, doing planning to find a sequence of actions that achieves a goal can be very time-consuming when there are more than a few actions and the domain is complicated. Also, it might not be possible to find a plan if complete information about the domain is not available and the plan found may not work if the environment changes as a result of actions performed by other agents. Golog tries to handle the efficiency problem by requiring the programmer to provide an extra piece of information, a high-level program, which describes the desired behavior for the agent. Instead of searching for a sequence of actions that leads to a given goal, the Golog interpreter takes the given program and tries to find a legal sequence of actions that leads to some final state of the program, one where the program can legally terminate. This can be formalized as:

$$Axioms \models Do(\delta, S_0, do(\sigma, S_0))$$

where δ is a high-level program and $Do(\delta, s, s')$ states that δ can start to run at s and may legally terminate in situation $do(\sigma, S_0)$. The supplied program may be nondeterministic, and the Golog interpreter must search to find a successful execution. As argued in [37], this is similar to classical planning in that the interpreter "needs to use what it knows about the prerequisites and effects of the actions to find an execution".

Golog allows programs to be nondeterministic by permitting them to have several terminating situations and nondeterministic choices between actions. For example, a nondeterministic choice on executing either one of the two programs, $\delta 1$ and $\delta 2$, can be written as follows:

$\delta 1 \mid \delta 2$

This means that the interpreter is free to execute either $\delta 1$ or $\delta 2$. Golog programs can also have nondeterministic choices of arguments. For example, the procedure *serveAllClients* finds a schedule to serve all clients that have shipments to be picked up or delivered by the robot:

```
proc serveAllClients
    while $(\exists x) clientToServe(x)$
        $(\pi x)[\ ?(clientToServe(x))\ ;\ serve(x)\ ]$
end
```

$(\pi x)[\ ?(clientToServe(x)); serve(x)\ ]$ nondeterministically chooses value for a client x, that allows the program to be successfully executed. Finally, Golog programs can also include nondeterministic iteration δ^* (i.e. perform δ 0 or more times).

The Golog interpreter has to search for a branch of the nondeterministic program that leads to successfully termination. A logic programming implementation of Golog has been developed, which does search and backtracking to find successful program executions.

3.3 ConGolog

The programming language *ConGolog* [13, 12] is an extension of Golog that supports concurrency. It provides extra facilities for concurrent program execution, priorities, interrupts, and exogenous actions during the program execution. It has been shown that these are useful to specify reactive behaviors for robots in dynamic environments, that is, environmental changes may occur at any time and the robots need to adjust their decisions on actions and react to the changes quickly.

The syntax of the ConGolog language is the same as that of Golog with the following additional program constructs:

$(\delta 1 \mid\mid \delta 2)$	concurrent execution
$(\delta 1 \gg \delta 2)$	concurrency with different priorities
$\delta^{\mid\mid}$	concurrent iteration
$< c \rightarrow \delta >$	interrupt

3.3.1 Concurrency

Concurrency means that two or more processes or program threads can be executing at the same time. ConGolog models the concurrent execution of programs as interleavings of the primitive actions and tests of these programs. The execution order of the primitive actions of the programs is chosen by the interpreter. Suppose a process is being executed and it reaches a point where it is about to perform a primitive action a or a test $c?$, where c is a condition. Also, suppose that it is impossible to do the action, that is, $Poss(a, s)$ is false, or c does not hold. In Golog, the interpreter would have to backtrack and try a different nondeterministic branch of the program, and it would fail if there is none. However, in ConGolog, the execution could proceed by having the process blocked and executing another process that is running concurrently. The suspended process will be unblocked and will get a chance to be executed again when all the preconditions of a or condition c become true.

For example, consider the following program:

$(a1; c?; a2) \mid\mid (a3)$

Suppose the condition c is false at the beginning and only performing the action $a3$ can make c hold. Suppose the interpreter first picks the concurrent process $(a1; c?; a2)$ and the first action $a1$ is then executed. It cannot continue with the same process since the next step in it is the test of condition c and this does not hold in the current situation. So the first process is blocked and the interpreter picks the other process, which is $(a3)$, and runs it. Once $a3$ has been performed, c becomes true and the first process is unblocked. The interpreter can then continue the execution of the first process, by doing the test on c and then executing the last action $a2$.

For the program:

$\delta 1 \gg \delta 2$

It also means that the programs $\delta 1$ and $\delta 2$ are executed concurrently, with the restriction that the interpreter can execute the actions or tests of $\delta 2$ only when $\delta 1$ is either done or blocked, that is, $\delta 1$ is executed with higher priority than $\delta 2$.

3.3.2 Interrupts

An interrupt in ConGolog has the form $< c \rightarrow \delta >$. It contains a condition c and a body program δ. It will be triggered and execute δ when the condition c become true and no other process with higher priority is able to run. Once it has completed the execution of δ, the interrupt can be triggered again. When there are several interrupts with diferent priorities running, the one with highest priority that has triggered and is not blocked will get executed first.

3.3.3 Exogenous Actions

Exogenous actions are primitive actions that may occur without being part of the user's program; they are performed by agents in the environment. The user can specify actions as exogenous by using the predicate $Exo(a)$. Exogenous actions can be generated manually by the user, randomly, or by a real environment when the ConGolog agent is interfaced to one.

3.3.4 Limitations of ConGolog

Although ConGolog has a rich account of concurrency, it is still not a perfect language for designing and implementing complex programs for agents working in dynamic environments. In Golog/ConGolog, an off-line search model is used and the interpreter must search all the way from the beginning to a final state of the program before it can execute a single action. Unfortunately, this execution model is often not suitable for running control programs for complex applications. Some of the limitations are as follows:

- It is impractical for the interpreter to spend large amounts of time on searching for a complete sequence of actions before performing any real actions when the environment may change and exogenous actions may occurs.

- It is impossible to find a plan if part of the program depends on some data that can only be obtained by doing sensing in the real world at run-time.

- The interpreter may not be able to find a complete plan for the applications which are designed for long run-times.

One way to handle these problems is to break the robot's task into smaller sub-tasks, and have several programs running at the same time, where each program works on one sub-task. However, how the main task should be broken up and how the programs can cooperate with each other become problems. Another possible solution is to separate planning and plan execution into two different modules as in [27]. The planning module which uses Golog to keep constructing plans; while the plan execution module is responsible for selecting a plan from those generated by the planning module and control the robot. The plans that are computed by the Golog interpreter are only treated as suggestions and it is the responsibility of the execution module to decide what actions will be executed.

Golog has been interfaced to software developed for the ARK robotics project [32, 52]. The ARK software was tested on the robots Nomad-200, ARK-1, and ARK-2 for monitoring tasks in industrial environments. In [61], it was used to design high-level robot controller for a mail delivery application. An extended version in ConGolog that can handle new mail requests and navigation failures was presented in [35]. Another version of Golog, called *sequential, temporal Golog*, which has the ability to represent temporal behavior and time explicity was proposed in [51]. In [11], it was used for planning and execution monitoring and implemented on a RWI-B21 robot to perform coffee delivery in an office environment. *cc-Golog* [26], an extended version of ConGolog which supports continuous change and event-driven behavior, was also used to build robot controllers for scheduling mail delivery tasks. A team at the University of Bonn also combined Golog with a plan executor to control a successful museum guide robot [7].

3.4 IndiGolog

The programming language *IndiGolog* [10] is the next generation of ConGolog. It has all ConGolog's features, such as nondeterministism and concurrency. With the additional facilities of on-line planning, sensing, and incremental execution, we believe it can be used to develop robust high-level control programs that involve planning and reactive capabilities.

3.4.1 Incremental Execution

The main difference between IndiGolog and the earlier members in the Golog family is its incremental approach to program execution. As we have explained, it is impractical to spend large amounts of time to search for a complete plan before execution when the program is large and complex and the environment may change. In cases where exogenous actions happen frequently and the amount of time for planning is limited, the system may not be fast enough to construct plans. Even if it can come up with a plan, the occurence of an exogenous acton may force it to throw the plan away and find a new one. Also, finding a complete plan may be problematic when the program depends on perceptual data that can only be obtained at execution time. For example, a robot that has to deliver a box n from a *sender* to a recipient may have to first go to the *sender*, pick up the box, and find out who the recipient is, before it can plan the delivery actions. The control program for this robot may look like:

```
[ $goTo(sender)$ ;
  $pickUp(n)$ ;
  $(\pi\ recipient)$ [ $read(recipient)$ ; $goTo(recipient)$ ] ;
  $dropOff(n)$ ]
```

ConGolog will fail to find a sequence of actions for this program because it cannot determine who the recipient is without executing the first 3 actions.

IndiGolog addresses these problems by switching to an incremental execution model, that is, it normally executes the next action immediately instead of trying to find a complete sequence of actions in advance. For our example, it does not need to know who the recipient is before it chooses and executes the action $goTo(sender)$. Next it picks up the box from the sender by doing the second action *pickUp*. After that, a sensing action finds out who the recipient of the box is by reading it from the box. Finally, it delivers the box to the recipient. At each point, it executes an action allowed by the program whose preconditions are satisfied, without doing lookahead. Note that once an action has been chosen and performed in the real world, it is impossible to backtrack to the previous state as could be done in Golog/ConGolog.

3.4.2 On-line Planning

Although incremental execution has its advantages, planning is still necessary in some cases, for example to optimize the sequence of actions that will be performed by the robot. The search block construct $search(d)$ in IndiGolog takes a program d and searches for a sequence of program transitions that leads to some final situation for the program in the block before it is executed. For a simple example, consider the following program:

$(a1; False?) \mid (a2; True?)$

Suppose that both actions $a1$ and $a2$ are executable. An executor that does no lookahead might first pick the left branch of the program $(a1; False?)$ and execute $a1$. Once $a1$ is performed, it comes to a dead end and cannot backtrack or complete the execution. If this program had been rewritten as:

$search(\ (a1; False?) \mid (a2; True?)\ )$

then the interpreter would have first checked the left branch and found that it could not be successfully executed to the end; then it would choose to execute the right branch $(a2; True?)$ which is the only one that is executable. Once the interpreter has found a sequence of transitions for the given program, it keeps following those transitions and makes sure that the program will terminate at a legal final situation. If an exogenous action occurs, it may have to redo the search to ensure that the execution of the search block program can be completed successfully.

3.4.3 Sensing

The robot may not have complete knowledge of the environment at the beginning and other agents may perform actions that change the environment during the program's execution. Information about the environment can be obtained by the robot through its sensors. The robot may keep checking for a particular event using its sensors during its operation and notify the IndiGolog control program about the occurence of the event by having an exogenous action occur. Exogenous actions arise from a form of sensing. But IndiGolog also allows explicit calls on sensing actions in the control program for the robot to obtain information it needs. In [10], the effects of performing a binary sensing action are modeled using $sensed\ fluent\ axioms$ of the form:

$$SF(a, s) \equiv \phi_a(s)$$

Such an axiom means that the value returned by the sensing action a in situation s is 1 if and only if ϕ_a is true in the same situation. For example, performing the sensing action *detect_obstacle* tells the robot whether it is blocked by an obstacle which is in front of it:

$$SF(detect\_obstacle, s) \equiv blocked(s)$$

Note that in general, the interpreter may not be able to find a complete execution of a program if the given program requires knowledge that it does not have. Because of this, IndiGolog assumes that the interpreter must have all the necessary information when it needs to decide whether an action should be performed or whether a condition is true. In other words, the value of a sensed fluent should be known when the interpreter needs it to make a decision.

3.4.4 Histories

One of the differences between the semantics of IndiGolog and ConGolog is that the semantics of IndiGolog uses the notion of *history* [10] to deal with sensing actions. A history is a list of pairs of an action and the associated sensing value, which can be either 0 or 1. For non-sensing actions in the list, the corresponding sensing value is always 1 and can be ignored. For example, the following history:

$$[(enter, 1), (senseDoorOpen, 1), (goToDoor, 1)]$$

represents the following situation:

$$do(enter, do(senseDoorOpen, do(goToDoor, S_0)))$$

where the sensing action *senseDoorOpen* produced the sensor value 1 (i.e. the door was open). By using histories, IndiGolog can evaluate sensed fluents correctly.

3.4.5 Formal Semantics of IndiGolog

The semantics of ConGolog and IndiGolog is specified using a form of transition system [47], which is formulated within the situation calculus. It uses two special predicates: $Final$ and $Trans$. The execution of an program is treated as a sequence of program transitions. A transition is a single step of computation of a program, which can be either performing a primitive action or testing whether a condition holds in a situation. $Trans(\delta, s, \delta', s')$ means that for a program δ and situation s, one can take a single step transition from δ and s by going to the situation s' with the program δ remaining to be executed. $Final(\delta, s)$ means that the program δ can legally terminate in situation s. We often refer to a pair (δ, s) of a program δ and a situation s as a *configuration*. We can think of $Trans$ as a relation on configurations.

The $Trans$ and $Final$ predicates are specified by axioms for the various program constructs. For example, the transition axioms for primitive actions and sequences are written as follows:

$$Trans(a, s, d, s') \equiv Poss(a, s) \wedge d = nil \wedge s' = do(a[s], s)$$

$$\begin{aligned}&Trans(d_1; d_2, s, d', s') \equiv \\ &\qquad (Final(d_1, s) \wedge Trans(d_2, s, d', s')) \vee \\ &\qquad (\exists d''.Trans(d_1, s, d'', s') \wedge d' = (d''; d_2))\end{aligned}$$

The first axiom states that one can take a transition from a primitive action a in situation s if a can be legally performed in s. There is no remaining program ($d = nil$) and the situation becomes $do(a[s], s)$ after the transition is performed ($a[s]$ evaluates the fluents in a in situation s). The second axiom says that one can take a transition from the program $(d_1; d_2)$ at situation s if either d_1 is $Final$ (already terminated) and there is a transition from d_2 at s to d' and s', or there is a transition from d_1 at s to d'' and s' and d' is $(d''; d_2)$.

The semantic of the search operator is also expressed in terms of $Final$ and $Trans$:

$$Final(search(d), s) \equiv Final(d, s)$$

$$\begin{array}{l} Trans(search(d), s, search(d'), s') \equiv \\ \qquad Trans(d, s, d', s') \wedge \\ \qquad \exists d'', s''.Trans^*(d', s', d'', s'') \wedge Final(d'', s'') \end{array}$$

The first axiom says that $search(d)$ at situation s can be terminated legally if program d at situation s is $Final$. The second axiom states that one can take a transition from $search(d)$ and situation s to $search(d')$ and s' if and only if there is a legal transition from d and s to d' and s', and also there is a sequence of legal transitions that turns d' and s' into some d'' and s'', where d'' with s'' is $Final$.

A detailed presentation of ConGolog's transition semantics can be found in [12]. [10] provides the semantics of the IndiGolog extensions, in particular the treatments of search blocks already mentioned, and sensing actions. The latter are modeled in terms of histories. At each step in a program's execution, what transitions are possible is determined by the history.

3.4.6 Limitations of IndiGolog

Although IndiGolog provides many features and has many advantages, it still has limitations. First of all, incremental execution allows actions to be executed as early as possible. However, in its interpreter, once a program branch has been chosen and some actions have been executed, the program is now committed to what it has done and there is no way to undo those actions or take another branch of the original program. The program execution may come to an dead end if it picked the wrong branch of the program. Also, although it provides the search construct for doing on-line planning, it may not always provide the best mechanism for combining execution to on-line planning. For example, one might want to find multiple plans from a program at first and then executes only the one that is the best. Also, as we will see in the next chapter, the replanning mechanism does not always work when the environment has changed.

3.4.7 Example: A Simple Robot Control Program in IndiGolog

Before describing the enhancements that we have made to the IndiGolog interpreter, let us present an example IndiGolog robot control program for doing shipment delivery. The complete program can be found in Appendix A. The task of the robot is to pick up all ordered shipments and deliver them to the recipients using the shortest possible route. In this program, the position of the robot is maintained by the primitive fluent *robotPos*. The robot can move to place c by doing the primitive action $goTo(c)$. At the location of a client, the robot can pick up a shipment n by doing $pickUp(n)$, and drop off a shipment n by performing $dropOff(n)$. Each shipment has a sender and a recipient, and an unique number is assigned to it. The state of a shipment n is modelled by the primitive fluent $shipmentPos(n)$. Its value can be a client's name or the value $onBoard$. A client's name indicates that the shipment is at the location of a client, while the value $onBoard$ means the robot is carrying the shipment. There are two other primitive fluents, $shipmentSender(n)$ and $shipmentRecipient(n)$, that denote the sender's name and recipient's name.

The robot determines whether it has to go to the location of a client c

and pick up or drop off some shipments by checking the value of the complex fluent $clientToServe(c)$, which is defined as follows:

```
proc $clientToServe(c)$
       $(\exists n) shipmentPos(n) = c \wedge shipmentTo(n) \neq c)$
       $\vee$
       $(\exists n) shipmentPos(n) = onBoard \wedge shipmentTo(n) = c)$
end
```

There are two complex actions for the robot to pick up and drop off shipments at a location. The robot can execute the action $pickAll(c)$ to pick up all the shipments that have to be picked up from the client c. The action $dropAll(c)$ makes the robot drop off every shipment for client c that it has on board. These are written as follows:

```
proc $pickAll(c)$
       while $((\exists n) shipmentPos(n) = c \wedge client(c) \wedge$
                    $shipmentRecipient(n) \neq c)$
              $(\pi n)$ [ $?(shipmentPos(n) = c \wedge client(c) \wedge$
                      $shipmentRecipient(n) \neq c);$
                    $pickUp(n2)$ ]
end

proc $dropAll(c)$
    while $((\exists n) shipmentPos(n) = onBoard \wedge shipmentRecipient(n) = c)$
        $(\pi n)$ [ $?(shipmentPos(n) = onBoard \wedge shipmentRecipient(n) = c);$
              $dropOff(n)$ ]
end
```

Our control program uses planning/search to optimize the robot's route. The main robot control procedure is as follows:

```
proc $control$
    $search(\ minimizeMotion(0)\ )$
end
```

Before any action is executed, the IndiGolog interpreter first finds the shortest route to serve all shipment orders by doing $search(minimizeMotion(0))$. Once it is found, the robot can follow the steps of the plan and deliver the shipments. The procedure *minimizeMotion* that finds the shortest route to handle all shipments by searching for a solution iteratively is written as follows:

```
proc $minimizeMotion(max)$
    $handleRequests(max)$
    $|$
    $minimizeMotion(max + 1)$
end
```

The procedure $handleRequests(max)$ makes the robot serve all shipments in at most *max* distance:

```
proc $handleRequests(max)$
    $?(\neg((\exists c)\ clientToServe(c)))$
    $|$
    $(\pi c, p, m)[\ ?(clientToServe(c) \wedge robotPos = p \wedge$
            $m = max - distance(p, c) \wedge m \geq 0);$
        $goTo(c);$
        $pickAll(c);$
        $dropAll(c);$
        $handleRequests(m)\ ]$
end
```

The $minimizeMotion$ procedure performs iterative deepening search to find the minimal total distance the robot has to travel in order to deliver all shipments. It first assumes that minimal total distance to do this is 0 ($minimizeMotion(0)$). If it is impossible to serve everything in 0 units of distance, then it increments the limit on the distance from 0 to 1 (*minimizeMotion(1)*) and checks if it is possible. If it is still impossible, it keeps incrementing the limit on the distance until it finds a limit that allows it to deliver the shipments. We can observe that the solution is actually the optimal solution, which is the one that requires the robot to travel the shortest distance to deliver all shipments.

Chapter 4

Integrating Planning into Reactive Indigolog Programs

This chapter describes how IndiGolog can be used to design and implement robot control programs that do planning, sensing, and react to changes in the environment together at the same time. In order to support all of these features, we have made several enhancements to the original IndiGolog planning and execution mechanisms. The following sections describe how they allow effective robot control programs to be written in this language.

4.1 An Enhanced Replanning Mechanism for IndiGolog

4.1.1 Limitations of the Original IndiGolog Replanning Mechanism

As mentioned in the previous chapter, IndiGolog has a search block construct $search(d)$ for doing search/planning in a program. In executing $search(d)$, the original IndiGolog interpreter first searches for a sequence of transitions from the given program d to some configuration where it can legally terminate. For efficiency, it saves the sequence it has found and keeps following it until its execution is completed or an exogenous action occurs. When the latter happens (i.e. the environment has changed), it rechecks whether the transitions in the sequence still work and if not, it performs a new search if necessary. The new search starts with the program that is currently left to be executed in the search block. This means that the interpreter has committed to the transitions that have been done so far, and backtracking to the original program d and taking another branch of it is not allowed. However, there is no reason to commit to the performed transitions; all that the agent is really committed to are the actions it has already performed.

There are many cases where it is impossible to find a sequence of transitions from the program that is currently left in the search block when the environment has changed during program execution. But if one were to start searching from the initial program and situation, then one would be able to find another sequence which involves the same sequence of actions that has already been performed, and leads to a terminating configuration. One example of this involves the shipment delivery program described in previous chapter:

```
proc control
    search( minimizeMotion(0) )
end

proc minimizeMotion(Max)
    handleRequests(Max)
    |
    minimizeMotion(Max + 1)
end

proc handleRequests(Max)
    $?(\neg((\exists c)\ clientToServe(c)))$
    |
    $(\pi c, p, m)$ [ $?(clientToServe(c) \wedge robotPos = p \wedge$
                        $m = Max - distance(p, c) \wedge m \geq 0)$;
                    goTo(c);
                    pickAll(c);
                    dropAll(c);
                    handleRequests(m) ]
end
```

By executing *search*(*minimizeMotion*(0)) at the beginning, this program does an iterative deepening search to find a sequence of transitions for the robot to serve all the clients that minimizes the distance it has to travel. Suppose that the robot has 3 clients to serve initially and has to travel at least 6 units of distance to serve all of their shipments. The interpreter has to find a sequence of transitions for the search block to serve the clients. At first, it can perform either *handleRequests*(0) or *minimizeMotion*(1). Since it is impossible to complete its mission in 0 distance (i.e. $handleRequests(0)$ fails), $minimizeMotion(1)$ is chosen. Then, the interpreter has to choose between $handleRequests(1)$ and $minimizeMotion(2)$. The distance bound keeps incrementing until it reaches 6, where $handleRequests(6)$ succeeds. The original IndiGolog interpreter then commits to the the program branch $handleRequests(6)$ and starts executing the sequence of transitions it has found. Now suppose an exogeonous event occurs in which a client makes a new new shipment order, and serving it requires the robot to travel 2 more units of distance. Since the program left in the search block is what remains of the execution of $handleRequests(6)$, the interpreter cannot increase the distance bound and is thus unable to complete its execution. However if it could reconsider the whole sequence of transitions from the original program $search(minimizeDistance(0))$ and choose the branch $handleRequests(8)$ instead of $handleRequests(6)$, it would find that this branch is an alternative solution which can be used to serve all clients including the new one. We conclude that the interpreter should not commit to a particular branch of the program and should replan from the original program and situation when necessary.

4.1.2 Replanning from the Initial Program and Situation

To allow replanning from the original program and situation, both of them must be memorized and kept within the search block as transitions are performed. We define an auxiliary construct $search'(d, di, si)$, where di and si are the initial program and situation (i.e. the program and situation when the search is first performed; we call this pair the initial configuration). The revised semantics of the search block is written in terms of $search'$ as follows:

$$Trans(search(d), s, d', s') \equiv Trans(search'(d, d, s), s, d', s')$$
$$Final(search(d), s) \equiv Final(search'(d, d, s), s)$$

The transition and termination axioms for $search'$ are defined as follows:

$$\begin{array}{l} Trans(search'(d, di, si), s, d', s') \equiv \\ \quad (\exists d'', d''')\ d' = search(d'', di, si) \wedge \\ \qquad Trans^*(di \| exo, si, d''' \| exo, s) \wedge Trans(d''', s, d'', s') \wedge \\ \qquad (\exists d'''', s'''')\ Trans^*(d'', s', d'''', s'''') \wedge Final(d'''', s'''') \end{array}$$

$$\begin{array}{l} Final(search'(d, di, si), s) \equiv \\ \quad (\exists d'')\ Trans^*(di || exo, si, d'' || exo, s) \wedge Final(d'', s) \end{array}$$

where the program exo is defined as $(\pi a)[exog\_action(a)?; a]$.

The first axiom states that one can take a single transition from *search'(d,di,si)* in situation s to $search'(d'', di, si)$ and s'' if and only if there is a sequence of transitions from the initial configuration that ends with some program d''' at the current situation s. This must involve the same sequence of primitive actions have been performed by the program since si with possibly some exogenous actions occurring. Also, there is a single transition from d''' in s to some program d'' and situation s', and a sequence of transitions that starts at d'' in s' to some final configuration s'''' and d''''. The second axiom says that the program $search'(d, di, si)$ is $Final$ in situation s if there is a sequence of transitions that takes the initial program and situation to a program d'' and the current situation s allowing some exogenous actions to occur, and d'' and s is a final configuration.

With this modification, the interpreter is able to switch to another program branch in cases such as the example described in the previous section. In that example, a client makes a new shipment order while the robot is serving some existing orders. The sequence of transitions found before the new order was made, which starts from $handleRequests(6)$ is not valid any more. The distance bound on the sequence of transitions for the robot to serve all orders including the new one must be at least 8. An interpreter that uses our enhanced search mechanism will be able to construct a new sequence of transitions for serving all orders. Starting from the original program $minimizeMotion(0)$, it will keep incrementing the distance bound until $minimizeMotion(8)$ is reached, where there is a sequence of transitions which starts from $handleRequests(8)$ in the initial situation to the current situation, and can be extended to a final situation where the robot has served all orders in 8 units of distance in total.

Our enhancement allows the interpreter to consider other branches of the given program in a search block and find a sequence of transitions that involves the same sequence of actions that has been performed since the beginning. When a new search is required during the execution of the search block, it is not restricted to the program that is left inside the block.

4.1.3 Efficiency Consideration

Searching for a sequence of legal transitions can be very time consuming. It is inefficient to perform a search every time the interpreter needs to take a transition of a search block as stated in the semantics. Our interpreter addresses this by storing the sequence it finds in its initial search and simply following it until all transitions in the sequence have been executed or an exogenous action occurs; it only redoes the search in the latter case.

The efficiency of the replanning mechanism can be further improved by checking whether the exogenous action that has occurred makes the saved sequence of transitions no longer valid. Replanning should be performed only when following the current sequence no longer leads to a legal termination. For example, if a person turns on the light in the robot's work area by performing the exogenous action *turnOnLight*, this does not affect the route of the robot to serve the clients and deliver the shipments. Thus, the sequence of transitions saved in the search block is still executable if this action occurs. If the interpreter replans, it will find the same sequence that was found in the initial search. Redoing the search would cause unnecessary delay. By checking the saved sequence before redoing the search, which is relatively fast, this inefficiency can be avoided.

4.2 Planning Using a Simulated Environment

A robot may not have complete information about its environment initially and may need to get additional data at run-time to decide which action should be performed in order to complete its task. For example, it may need to get sensor data to find out whether it is facing an obstacle. It is impossible to know what data sensing will produce until its actual execution. Planning is often useful to optimize the performance of the robot, but it can be hard to do if some of the necessary information depend on exogenous events or environmental data that can be known only at run-time. One solution is to interleave planning and sensing. The robot constructs a partial plan up to the point where making the next decision requires data that has to be obtained at run-time. Then it executes the partial plan, and obtains the data from the environment or wait for some exogenous event to occur, then searches for the next partial plan, and so on and so forth. The major problem with this approach is that if the program needs frequent feedback from the environment, then the planner can only do a limited amount of lookahead and construct a short partial plan each time. As a result, planning may provide only limited benefit. This might not be an appropriate approach if global optimization of actions is needed, requiring planning from the beginning to the end of the program.

One instance of this is when the robot must rely on the occurrence of exogenous actions at some point in its plan. These could be signals from other components in the robot's architecture, such as the completion of a robot motion, or actions by other agents, such as an indication of a shipment having been placed in the robot's bin for transport. The original IndiGolog interpreter cannot construct a sequence of transitions for a search block where the program relies on the occurrence of exogenous actions. Usually, there is one normal outcome exogenous action that would typically happen, and planning could proceed if we are willing to be optimistic and allow it to simulate the expected exogenous action. So, the approach we develop here is to add simulated actions into an environment process that is executed concurrently with the robot control process.

Let us illustrate this with an example. Suppose that we want to use a more general model of navigation that allows for failure and permits the robot to do other things while navigation proceeds. The robot will perform an action *startGoTo*(*Location*) to initiate the navigation and then wait for an exogenous event indicating that it has reached the client's location before serving the client. The *handleRequest* procedure is rewritten as follows for this purpose:

```
$proc\ handleRequests(Max)$
    $?(\neg((\exists c) clientToServe(c)))$
    $|$
    $(\pi\ c, p, m)$ [ $?(clientToServe(c) \wedge robotPos = p \wedge$
                         $m = Max - distance(p, c) \wedge m \geq 0);$
                      $startGoTo(c);$
                      $?(robotState \neq moving);$
                      $if(robotState = reached)\ then$
                         [ $pickAll(c);$
                           $dropAll(c)$ ];
                      $handleRequests(m)$ ]
$end$
```

Suppose that there are some clients for the robot to serve initially. The program first picks a client c and performs the action *startGoTo*(c) to start moving to the client's location. It is then blocked at the test $?(robotState \neq moving)$ until the exogenous action *reachDest* occurs, signaling that the robot has actually reached the client's location (on the other hand, if the navigation fails and the exogenous action *getStuck* occurs, the robot will keep trying to handle all requests). Afterward, the robot can finish serving the client by picking up and dropping off its shipments, and then proceed to serve other clients. However, the exogenous action *reachDest* is not part of the program and only happens during the program execution. To allow the interpreter to construct a sequence of transitions with incomplete information in this case, we add an environment simulator program that generates a simulated version of the *reachDest* action. The environment simulator that inserts this piece of information into the search during planning is written as follows:

```
proc envSimulator
    $< \ robotState = moving \ \rightarrow \ sim(reachDest) \ >$
end
```

This procedure is executed concurrently with the robot motion control program:

```
proc control
    $search(minimizeMotion(0) \ \| \ envSimulator)$
end
```

Whenever the search reaches a situation where the robot is moving to its destination ($robotState = moving$), the simulator generates a simulated action $sim(reachDest)$ to indicate that the robot has reached the location successfully. Therefore, the search can proceed as usual.

This simulated version of an action is treated same as the real action during planning. We specify:

$Poss(sim(a), s) \equiv Poss(a, s)$

$$F(\vec{x}, do(sim(a), s)) \equiv F(\vec{x}, do(a, s)) \qquad \text{for any fluent } F$$

Here we assume the simulated exogenous action $sim(a)$ has the same preconditions and effects as the real one. However, simulated actions are never actually executed. When the interpreter executes the sequence of transitions that is saved in the search block and encounters a simulated action such as $sim(reachDest)$, this action is simply skipped and the interpreter assumes that some exogenous action must happen at this moment. It keeps waiting until some real exogenous action occurs. If the real exogenous action is the one that it expected in the search ($reachDest$ in our example), the interpreter can replace the simulated one with the real one and continue with the execution. If the exogenous action that occurs is something else, such as $getStuck$ to signal that the robot is blocked by an obstacle on its way to its destination, replanning may be performed.

4.3 Combining Reactive Behaviors and Planned Behaviors

A robot that works in a dynamic environment is not only required to perform its main task. It must also watch for changes in the environment and react to them immediately to avoid running into problems. So, the robot program usually has a deliberative element for planning and controlling the robot to accomplish its main task, and some reactive elements for reacting to environmental changes quickly. In this kind of structure, the program should contain a thread for suggesting what actions the robot should perform for its main task, which often involves search or planning, and also other higher priority threads to monitor for important environmental changes and perform the corresponding reaction. The main thread will be blocked when an exogenous event occurs and a reactive thread will be executed to solve the problem immediately. Cooperation between the these threads is very important, since the decision of what action to perform in the main thread may have to depend on what has been done by the reactive threads.

Consider the following example. We insert a high priority interrupt in our shipment delivery robot control program to detect when a client has made a new shipment order and send back an acknowledgement to the client. This additional thread looks like the following:

$< (\exists n) newOrder(n) \rightarrow acknowledge(n) >$

Here the fluent $newOrder(n)$ becomes true when a client sends in a new shipment order n and then the robot acknowledges it to the client by doing the primitive action $acknowledge(n)$. Since sending the acknowledgement does not affect the current motion of the robot, this is essentially independent from the main control of the robot. We want to write the whole control program as follows, with the reactive thread running at higher priority than the motion control thread:

```
proc control
    $< (\exists n) newOrder(n) \rightarrow acknowledge(n) >$
    $\gg$
    $search(minimizeMotion(0))$
end
```

Unfortunately, the original IndiGolog interpreter will fail to find a sequence of transitions for this program after an acknowledgement has been performed. The *search* block can find a sequence for the search block to control the robot to serve the shipments, but it does not have the required information to deal with the actions that have been performed by the reactive thread, for instance, the *acknowledge* action. When the interpreter replans, it can only use the program provided inside the search block and is not able to reason about the *acknowledge* actions which were performed by other threads. Therefore, replanning will fail at this point.

One way to solve this problem is to put all the threads inside the search block and ask the interpreter to find a sequence of transitions that contains the actions required to accomplish the main task as well as react to all kinds of exogenous events:

```
proc control
    search(  $< (\exists n) newOrder(n) \rightarrow acknowledge(n) >$
             $\gg$
             $minimizeMotion(0)$ )
end
```

However, searching for a larger program is more complicated and time consuming. If the search involves not only finding a sequence for accomplishing the main task but also handling all exogenous events reactively, the interpreter may spend much more time on planning instead of executing actions. Moreover, the interpreter has to replan *before* it can perform the reaction to the exogenous event.

We can handle programs that involve both search threads and reactive threads like the one in the example on the previous page if we make search blocks remember what actions they have performed and ignore other actions that are exogenous or performed by other threads in replanning. The semantics of the search block with this enhancement are as follows:

$$Trans(search(d), s, d', s') \equiv Trans(search'(d, d, s, \emptyset), s, d', s')$$

$$Final(search(d), s) \equiv Final(search'(d, d, s, \emptyset), s)$$

$$\begin{aligned}
&Trans(search'(d, di, si, I), s, d', s') \equiv \\
&\qquad (\exists d'', d''')\ d' = search'(d'', di, si, I') \land \\
&\qquad\qquad\qquad Trans^*(di \| d_o(I), si, d''' \| d_o(I), s) \land Trans(d''', s, d'', s') \land \\
&\qquad\qquad\qquad s' = s \to I' = I \land (\exists a).(s' = do(a, s) \to I' = I \cup s') \land \\
&\qquad\qquad\qquad (\exists d'''', s'''')\ Trans^*(d'', s', d'''', s'''') \land Final(d'''', s'''')
\end{aligned}$$

$$\begin{aligned}
&Final(search'(d, di, si, I), s) \equiv \\
&\qquad (\exists d'')\ Trans^*(di || d_o(I), si, d'' || d_o(I), s) \land Final(d'', s)
\end{aligned}$$

where $d_o(I) \equiv ((\pi a)\ if\ (do(a, now) \notin I)\ then\ a\ else\ False?)^*$

Here I is a set of situations $do(a, s)$ where the last action a of each situation in the set comes from inside the search block, and $d_o(I)$ is a program that can generate actions that are exogenous or from threads outside the search block (i.e. do not belong to the set I). The third axiom says that one can take a transition from $search'$ if and only if there is a sequence of legal transitions from the initial program di and situation si that involves the actions performed since the initial configuration, and leads to some final configuration. If performing the next transiton to $search'(d'', di, si, I')$ in situation s' involves an action a from inside the search block, then the situation after the action is performed is added into the set I.

4.4 Implementation

In this section, we present the implementation of the enhancements to the IndiGolog interpreter that have been described in the previous sections. The interpreter is written mainly in terms of the Prolog clauses `final` and `trans`, which are the implementation of the $Final$ and $Trans$ predicates in the semantics. Like that in [10], our implementation uses *histories* instead of situations to represent states of the world with sensing information. As described in Chapter 3, a history is a list of primitive actions sensor values with the associated for sensing actions. In the implementation, sensor values are included in the history as sensor reports of the form $e(fluent, value)$ following the sensing action. For example, the history

$$[\ (enter, 1), (senseDoorOpen, 1), (gotoDoor, 1)\ ]$$

corresponding to the situation

$$do(enter, do(senseDoorOpen, do(gotoDoor, S_0)))$$

where the sensing action *senseDoorOpen* produced the sensor value 1 is represented in the implementation by the list

$$[\ enter, e(doorOpen, 1), senseDoorOpen, gotoDoor\ ]$$

4.4.1 The Enhanced Search Mechanism

As mentioned earlier, for efficiency, our interpreter saves the sequence of transitions it finds in the search for a search block so that no more search is needed unless the sequence becomes invalid because of the occurrence of some exogenous action. The sequence is saved as a *path*, which is a list of program-and-history pairs: `[E0,H0,E1,H1,...,En,Hn]`, where `E`'s are programs and `H`'s are histories. Each pair `(Ek,Hk,Ek+1,Hk+1)`, with $0 \leq$ `k` $<$ `n`, of the path represents a transition from the configuration `(Ek,Hk)` to `(Ek+1,Hk+1)`. Also, `(En,Hn)` is a final configuration. In order to replan from the original configuration when the path becomes invalid, the interpreter also stores the initial program, initial history, and a list of *snapshots*, in addition to the path. The list of snapshots corresponds to the set I of actions from inside the search block in the semantics. A snapshot is a history `[An,An-1,...,A1]` and it is added to the list of snapshots only when the program inside the search block performs a primitive action. In other words, the last action `An` of each snapshot in the list comes from inside the search block. This is used to handle programs that contain both search and reactive threads. The snapshots appear in the list in reverse order; we will give an example later in this section.

Let us now go over the details of the `trans` clauses for handling the search construct. The following clauses are used by the interpreter to perform the initial search and store the execution path found when it encounters a search block in the program:

```
trans(search(E),H,E1,H1) :-
    findpath([],E,H,P), trans(followpath(P,[],E,H),H,E1,H1).

findpath([],E,H,[E,H]) :- final(E,H).
findpath([],E,H,[E,H|L]) :- trans(E,H,E1,H1), findpath([],E1,H1,L).
```

The interpreter handles a search block `search(E)` by first calling `findpath` to find a path for the program inside the block and then transforming everything into a `followpath(P,[],E,H)` contruct, where `P` is the path found by `findpath`, `E` is the initial program, `H` is the initial history, and `[]` is the list of snapshots, which is empty at the beginning. The `followpath` term is analogous to *search'* in the semantics. `findpath(_,E,H,P)` will succeed if it can find a path from the initial program `E` and initial history `H` to a terminating configuration; the path is bounded to `P`.

Once a path is found, the interpreter keeps following it as long as no exogenous action and no action from another thread occurs, that is, as long as the current history `H` remains the same as the one saved in the path:

```
trans(followpath([E,H,E1,H|L],LS,E0,H0),H,
      followpath([E1,H|L],LS,E0,H0),H) :- !.
trans(followpath([E,H,E1,[A|H]|L],LS,E0,H0),H,
      followpath([E1,[A|H]|L],[[A|H]|LS],E0,H0),[A|H]) :- !.
```

If the current history does not match the one that is saved in the path because of the occurence of an action from outside the search block, the interpreter checks to see if the path can still be followed to get to a terminating configuration. That is, if the remaining path is `[E0,H0,...,En,Hn]` and the current history is `CH`, it checks whether $Trans^*(E0, CH, En, H') \wedge Final(En, H')$ holds. Actually, if the path can still be followed to a terminating configuration, we want to fix the histories in it to include the outside action that made the current history different from the one expected. This is done by `canfixpath(P,CH,P1)`, where the fixed path is returned as `P1`. The `trans` clause for this case is as follows:

```
trans(followpath([E,H,E1,H1|L],LS,E0,H0),CH,E2,H2) :-
    canfixpath([E,H,E1,H1|L],CH,L1),
    trans(followpath(L1,LS,E0,H0),CH,E2,H2), !.

canfixpath([E,H],CH,[E,CH]) :- final(E,CH).
canfixpath([E,H,E1,H1|P],CH,[E,CH,E1,CH1|P1]) :-
    trans(E,CH,E1,CH1), canfixpath([E1,H1|P],CH1,[E1,CH1|P1]).
```

In the case where the current path is not valid anymore, the interpreter finds a new path that starts from the initial configuration and involves the actions that have already been performed by the program inside the search block. This can be seen in the last `trans` clause of the search mechanism:

```
trans(followpath([E,H,E1,H1|L],LS,E0,H0),CH,E2,H2) :-
    append(LS,H0,LS1), extactout([CH|LS1],[],AL),
    findpath(AL,E0,H0,P), trans(followpath(P,LS,E0,H0),CH,E2,H2).
```

As shown in the body of the clause, the interpreter first calls `extactout` (and `extactin`) to construct a list of the actions performed by the search block from the saved sequence of snapshots and current history. Each primitive action `A` that was performed by the program inside the search block in this list is labelled as `inside(A)`:

```
extactin([H0],AL,AL).
extactin([[A|H1],H|L],AL,R) :- extactout([H1,H|L],[inside(A)|AL],R).

extactout([H,H|L],AL,R) :- extactin([H|L],AL,R).
extactout([[A|H1],H|L],AL,R) :- extactout([H1,H|L],[A|AL],R).
```

For example, consider the following list of snapshots `LS`:

```
[startGoTo(p2),reachDest,pickUp(1),acknowledge(2,p2,p3),startGoTo(p1)],
[pickUp(1),acknowledge(2,p2,p3),startGoTo(p1)],
[startGoTo(p1)]
```

After adding the initial history `[]` to the end of the list, we can get the following list of performed actions `AL` from `extactout(LS,[],AL)`:

```
[inside(startGoTo(p1)), acknowledge(2,p2,p3), inside(pickUp(1)),
 reachDest, inside(startGoTo(p2))]
```

Once the list of performed actions is found, the interpreter uses `findpath` to construct a new path. The following three additional `findpath` clauses are used to handle the case where we need to find a path that includes some already performed actions:

```
findpath([A|AL],E,H,L) :-
    trans(E,H,E1,H), findpath([A|AL],E1,H,L).
findpath([inside(A)|AL],E,H,L) :-
    prim_action(A), trans(E,H,E1,[A|H]), findpath(AL,E1,[A|H],L).
findpath([A|AL],E,H,L) :-
    A \= inside(_), findpath(AL,E,[A|H],L).
```

The first clause holds if there is a transition from program `E` to `E1` that does not involve performing an action. Otherwise, if the next action `A` in the list is labeled as `inside(A)`, it attempts to find a transition that takes the program `E` to `E1` which involves that action. If the action `A` is not labeled (i.e. either an exogenous action, a sensing value or some action that was done by a thread outside the search block), the interpreter simply adds it into the history. By combining these with the original two `findpath` clauses, a new path `P` can be found by executing `findpath(AL,E0,H0,P)`, where `E0` is the initial program, `H0` is the initial history, and `AL` is the actions that have been performed since `H0`.

The `final` clauses of the search block construct that specify the terminating configurations are written in the similar way as those for `trans`. They are as follows:

```
final(search(E),H) :- final(E,H).

final(followpath([E,H],LS,E0,H0),H) :- !.
final(followpath([E,H|P],LS,E0,H0),CH) :- final(E,CH), !.
final(followpath(P,LS,E0,H0),CH) :-
    addtail(LS,H0,LS1), extactout([CH|LS1],[],AL), !,
    findpath(AL,E0,H0,[E1,H1]).
```

4.4.2 The Simulated Environment Mechanism

Our approach to planning in applications where the agent needs feedback from the environment involves the use of an environment simulator program to produce simulated exogenous actions so that the search mechanism can find an execution path. Simulated actions generated by the environment simulator are labeled $sim(\alpha)$, where α is a real exogenous action. As we have seen, a simulated action is treated in the same way as the corresponding real action during the planning/search phase. The implementation supports this as follows:

```
prim_action(sim(A)) :- exog_action(A)

has_val(F,V,[sim(A)|H]) :- has_val(F,V,[A|H])

poss(sim(A),P) :- poss(A,P)
```

The first clause says that `sim(A)` is a simulated primitive action if and only if `A` is an exogenous action. The second clause states that they have the same effect on the value of fluents. The third clause says that the preconditions for performing `sim(A)` are the same as those for the exogenous action `A`; so the simulator is allowed to generate a simulated action `sim(A)` if the exogenous action `A` could occur in the same history.

During execution on the other hand, we never want to perform the simulated actions; we treat them as indications that we should wait for a real exogenous action to occur (hopefully the expected one). When this happens, the simulated action can be discarded. Let us look at the execution mechanism in more detail. The interpreter takes a program and executes it transition by transition until it can legally terminate. Before performing each transition of the program, the interpreter checks to see if any new exogenous actions have occured, or whether the program can terminate. When an exogenous action happens, the interpreter inserts the exogenous action into the current history, and if the next transition involves the corresponding simulated action, the transition is also performed to advance the program (i.e. the real exogenous action is substituted for the simulated action). This is done in the following clauses:

```
indigolog(E) :- indigo(E,[]).
indigo(E,H) :- exog_occurs(A), exog_action(A), !, subsim(E,A,H).
indigo(E,H) :- final(E,H).
indigo(E,H) :- trans(E,H,E1,H1), !, checksim(E,H,E1,H1).
```

An IndiGolog program `E` can be executed by invoking `indigolog(E)`. When an exogenous action occurs (both `exog_occurs(A)` and `exog_action(A)` hold), the interpreter calls `subsim(E,A,H)` to see if the action `A` can be substituted for a simulated action in the program. If this is the case (i.e. taking the next transition involves a simulated action which corresponds to `A`), then the exogenous action is substituted into the current history and next transition is taken. If the next transition involves a simulated action that does not correspond to the exogenous action that has just occurred, then the exogenous action is inserted into the current history as usual and the interpreter does not advance the program. This is done as follows:

```
subsim(E,A,H) :- trans(E,H,E1,H1), subsim2(E,A,H,E1,H1).
subsim2(E,A,H,E1,[sim(A)|H]) :- !, indigo(E1,[A|H]).
subsim2(E,A,H,E1,H1) :- !, indigo(E,[A|H]).
```

Since simulated actions are just simulated and the interpreter expects some exogenous action to occur when it encounters such an action during execution, the interpreter does nothing when it reaches a transition involving a simulated action and waits for the occurrence of an exogenous action. This is done by calling `checksim(E,H,E1,H1)` whenever the interpreter considers taking a transition:

```
checksim(E,H,E1,[sim(_)|H]) :- !, indigo(E,H).
checksim(E,H,E1,H1) :- indixeq(H,H1,H2), !, indigo(E1,H2).

indixeq(H,H,H).
indixeq(H,[A|H],[e(F,Sr),A|H]) :- senses(A,F), !, execute(A,Sr).
indixeq(H,[A|H],[A|H]) :- execute(A,_).
```

Before the interpreter takes a transition from the program `E` in history `H` to program `E1` and history `H1`, it checks to see whether doing it involves a simulated action. If this is true (i.e. `checksim(E,H,E1,[sim(_)|H])` holds) then it waits by calling `indigo(E,H)` until an exogenous action occurs. Otherwise, the program can be advanced as usual and `indixeq` is called to perform the real action.

4.5 A Complete Example Using the Enhanced Interpreter

In this section we present a complete robot control program for shipment delivey which involves both planned and reactive behaviors, and uses the enhancements we have made to the interpreter. We provide execution traces of the program for some example scenarios. Here we do not hook up the controller to a real robot. We will show how this is done in chapter 7. The task of the robot is to serve all clients while minimizing the distance it travels.

4.5.1 Domain Specification

The domain specification is as follows:

Ordinary primitive actions:

$startGoTo(Place)$	robot starts moving to $Place$
$pickUp(SNo)$	robot picks up shipment $\#SNo$
$dropOff(SNo)$	robot drops off shipment $\#SNo$
$abortGoTo$	robot stops moving to its destination
$acknowledge(SNo, Client)$	acknowledges shipment order $\#SNo$ to $Client$

Exogenous primitive actions:

$reachDest$	robot has reached its destination
$getStuck$	robot unable to reach destination
$orderShipment(SNo, Sndr, Rcpnt)$	client $Sndr$ wants to send shipment $\#SNo$ to $Rcpnt$

Primitive fluents:

$robotPos$	position of the robot
$robotState$	state of the robot
$robotDest$	destination of the robot
$shipmentPos(SNo)$	position of shipment $\#SNo$
$shipmentSender(SNo)$	sender of shipment $\#SNo$
$shipmentRecipient(SNo)$	recipient of shipment $\#SNo$

Precondition axioms:

$$
\begin{aligned}
&Poss(startGoTo(Place), s) \equiv \\
&\qquad robotState(s) = idle \lor robotState(s) = reached \\
&Poss(reachDest, s) \equiv robotState(s) = moving \\
&Poss(getStuck, s) \equiv robotState(s) = moving \\
&Poss(abortGoTo, s) \equiv robotState(s) = moving \\
&Poss(pickUp(SNo), s) \equiv shipmentPos(SNo, s) = robotPos(s) \\
&Poss(dropOff(SNo), s) \equiv shipmentPos(SNo, s) = onBoard \\
&Poss(orderShipment(SNo, Sender, Recipient), s) \equiv true \\
&Poss(acknowledge(SNo, Client), s) \equiv \\
&\qquad shipmentPos(SNo, s) = nonExistent \land \\
&\qquad shipmentSender(N, s) = Client
\end{aligned}
$$

Successor state axioms:

$$
\begin{aligned}
&robotPos(do(a, s)) = p \equiv \\
&\qquad a = startGoTo(p) \land p = unknown \lor \\
&\qquad a = reachDest \land p = robotDest \lor \\
&\qquad (\forall\, p')a \neq startGoTo(p') \land a \neq reachDest \land robotPos(s) = p \\
&robotState(do(a, s)) = x \equiv \\
&\qquad (\exists\, p)a = startGoTo(p) \land x = moving \lor \\
&\qquad a = reachDest \land x = reached \lor \\
&\qquad a = getStuck \land x = idle \lor \\
&\qquad a = abortGoTo \land x = idle \lor \\
&\qquad (\forall\, p')a \neq startGoTo(p') \land a \neq reachDest \land a \neq getStuck \land \\
&\qquad\qquad a \neq abortGoTo \land robotState(s) = x \\
&robotDest(do(a, s)) = p \equiv \\
&\qquad a = startGoTo(p) \lor \\
&\qquad (\forall\, p')a \neq startGoTo(p') \land robotDest(s) = p \\
&shipmentSender(n, do(a, s)) = c \equiv \\
&\qquad (\exists\, r)a = orderShipment(n, c, r) \lor
\end{aligned}
$$

$$
\begin{aligned}
&\quad (\forall\ r')a \neq orderShipment(n,c,r') \wedge shipmentSender(n,s) = c\\
&shipmentRecipient(n, do(a,s)) = c \equiv\\
&\quad (\exists\ x)a = orderShipment(n,x,c) \vee\\
&\quad (\forall\ x')a \neq orderShipment(n,x',c) \wedge shipmentRecipient(n,s) = c\\
&shipmentPos(n, do(a,s)) = p \equiv\\
&\quad a = acknowledge(n,p) \vee\\
&\quad a = pickUp(n) \wedge p = onBoard \vee\\
&\quad a = dropOff(n) \wedge p = robotPos(s) \vee\\
&\quad (\forall\ p')a \neq acknowledge(n,p') \wedge a \neq pickUp(n) \wedge\\
&\qquad a \neq dropOff(n) \wedge shipmentPos(n,s) = p
\end{aligned}
$$

As already discussed, the robot control program consists of the complex procedures *handleRequests* and *minimizeMotion* which use iterative deepening search to find the shortest route the robot has to travel in order to deliver all shipments. The *minimizeMotion* procedure is executed concurrently with the environment simulator. The program is as follows:

```
proc control
    $search(minimizeMotion(0) \parallel envSimulator)$
end

proc minimizeMotion(Max)
    $handleRequests(Max)$
    |
    $minimizeMotion(Max + 1)$
end

proc handleRequests(Max)
    $?(\neg((\exists c) clientToServe(c)))$
    |
    $(\pi\ c, p, m)$ [ $?(clientToServe(c) \wedge robotPos = p \wedge$
                       $m = Max - distance(p,c) \wedge m \geq 0);$
                     $startGoTo(c);$
                     $?(robotState \neq moving);$
```

```
                  $if\ (robotState = reached)\ then$
                     $[\ pickAll(c);$
                       $dropAll(c)\ ];$
                  $handleRequests(m)\ ]$
$end$

$proc\ envSimulator$
     $<\ robotState = moving\ \rightarrow\ sim(reachDest)\ >$
$end$
```

We define the fluent $clientToServe(c)$ to hold if the robot has to go to the location of a client C to pick up some shipments or has shipments on board that must be delivered to that client:

```
$proc\ clientToServe(c)$
     $(\exists n1) shipmentPos(n1) = c \wedge shipmentRecipient(n1) \neq c)$
     $\vee$
     $(\exists n2) shipmentPos(n2) = onBoard \wedge shipmentRecipient(n2) = c)$
$end$
```

4.5.2 Example 1: Planning with Simulated Environment

Our first example shows that our interpreter performs planning to minimize the distance travelled by the robot and that this can be done even when the robot must rely on feedback from the environment, by using an environment simulator program. We suppose that there are 3 clients: *yves*, *hector* and *mike*, and 2 shipments for the robot to deliver: shipment 1 from *yves* to *hector* and shipment 2 from *hector* to *mike*. The robot is at the *home* position at the beginning. The initial axioms for this are as follows:

$robotPos(s0) = home$ $robotState(s0) = idle$
$robotDest(s0) = home$
$distance(yves, hector, 1)$ $distance(yves, mike, 1)$
$distance(hector, mike, 2)$ $distance(home, yves, 1)$
$distance(home, hector, 1)$ $distance(home, mike, 1)$

$shipmentPos(1) = yves$ $shipmentSender(1) = yves$
$shipmentRecipient(1) = hector$ $shipmentPos(2) = hector$
$shipmentSender(2) = hector$ $shipmentRecipient(2) = mike$

Note that we set up the initial situation such that there are shipment orders at the beginning. Here is the trace produced by the program:

```
| ?- indigolog_itr(control).
Exogenous input:nil.
start_interrupts
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
path :  [startGoTo(yves), sim(reachDest), pickUp(1), startGoTo(hector),
sim(reachDest), pickUp(2), dropOff(1), startGoTo(mike), sim(reachDest),
dropOff(2)] in [start_interrupts]
```

At this step the interpreter found the shortest route, `home->yves->hector->mike` with a total distance of 4. Another possible route is `home->hector->mike->yves->hector`, but its total distance is 5. We can also see that the environment simulator inserted the simulated action *sim*(*reachDest*) right after each occurrence of *startGoTo* in the path successfully. Then, the path is executed:

```
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
startGoTo(yves)
Exogenous input:nil.
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
pickUp(1)
Exogenous input:nil.
Exogenous input:nil.
startGoTo(hector)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
pickUp(2)
Exogenous input:nil.
dropOff(1)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
startGoTo(mike)
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
dropOff(2)
Exogenous input:nil.
Exogenous input:nil.
stop_interrupts
Exogenous input:nil.

yes
```

Once the robot had performed `startGoTo`, it kept waiting until the real exogenous action `reachDest` occurs. The exogenous action was then substituted into the history and no replanning was needed. Afterwards, it performed the `pickUp` and `dropOff` actions and continued following the path obtained before. This shows that the program works in the case where the exogenous action that really occurs corresponds to the simulated one in the path. With our enhancements, the interpreter can find a path even though the robot relies on environmental feedback and never needs to replan because the environment behaves as expected.

4.5.3 Example 2: Replanning

Our second example demonstrates what our interpreter can perform replanning when the current path is no longer valid after the occurence of an exogenous action. We temporarily prevent the robot from moving from *hector* to *mike*, for example, by putting a box on the robot's route. The trace produced is as follows:

```
| ?- indigolog_itr(control).
Exogenous input:nil.
start_interrupts
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
path :  [startGoTo(yves), sim(reachDest), pickUp(1), startGoTo(hector),
sim(reachDest), pickUp(2), dropOff(1), startGoTo(mike), sim(reachDest),
dropOff(2)] in [start_interrupts] Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
startGoTo(yves)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
pickUp(1)
Exogenous input:nil.
Exogenous input:nil.
startGoTo(hector)
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
pickUp(2)
Exogenous input:nil.
dropOff(1)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
startGoTo(mike)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:getStuck.
Exogenous input:nil.
path :  [startGoTo(mike), sim(reachDest), dropOff(2)] in [getStuck,
startGoTo(mike), dropOff(1), ...] Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
startGoTo(mike)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
dropOff(2)
Exogenous input:nil.
Exogenous input:nil.
stop_interrupts
Exogenous input:nil.

yes
```

This is a case where the actual exogenous action that occurs at run-time is not the one expected during planning. We expect the robot to reach its destination successfully and so the environment simulator always generates the simulated action *reachDest*. However, the robot was blocked by the box on its way to `mike` and so it received the `getStuck` signal. Therefore, the interpreter replanned to make a second attempt to deliver shipment 2 to `mike` again. Afterwards, we moved the box away so that the robot could reach `mike` (exogenous action `reachDest` occured) and drop off the shipment. Note that we could easily handle failures in a more sophisticated way, for example, by suspending the client temporarily or giving up after a number of tries.

4.5.4 Example 3: Combining Planned and Reactive Behaviors

To demonstrate how a robot control program can combine planned and reactive behaviors, we added a reactive thread into the shipment delivery example. This new reactive thread simply performs the action *acknowledge* to acknowledge the sender when he/she makes a new shipment order. The main control program is now rewritten as follows:

```
proc control
    $< (\exists n) newOrder(n) \rightarrow handleNewOrder(n) >$
    $\gg$
```

```
        $search(minimizeMotion(0))$
    $end$
```

The robot acknowledges new shipment orders by calling the procedure *handleNewOrder*:

```
    $proc\ handleNewOrder(N)$
        $(\pi c)[\ ?(shipmentSender(N) = c);$
            $acknowledge(N, c)$
          $]$
    $end$
```

The fluent $newOrder(n)$ holds when a client has just made a new shipment order n and it is cancelled when the robot acknowledges the shipment:

$$\begin{aligned}&newOrder(n, do(a, s)) \equiv\\ &\quad (\exists\ c, r)\ a = orderShipment(n, c, r) \vee\\ &\quad (\forall\ c')\ a \neq acknowledge(n, c') \wedge newOrder(n, s)\end{aligned}$$

The robot is first asked to serve 2 shipments: shipment 1 from *yves* to *hector* and shipment 2 from *hector* to *mike* initially. During the execution, *yves* makes another shipment order. As the result, the robot must also serve shipment 3 from *yves* to *mike*. Here is the trace of the execution:

```
| ?- indigolog_itr(control).
Exogenous input:nil.
start_interrupts
Exogenous input:orderShipment(1,yves,hector).
Exogenous input:orderShipment(2,hector,mike).
Exogenous input:nil.
Exogenous input:nil.
acknowledge(1,yves)
Exogenous input:nil.
Exogenous input:nil.
acknowledge(2,hector)
Exogenous input:nil.
path :  [startGoTo(yves), sim(reachDest), pickUp(1), startGoTo(hector),
sim(reachDest), pickUp(2), dropOff(1), startGoTo(mike), sim(reachDest),
dropOff(2)] in [acknowledge(2,hector), acknowledge(1,yves),
orderShipment(2,hector,mike), ...]
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
startGoTo(yves)
Exogenous input:nil.
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
pickUp(1)
Exogenous input:nil.
Exogenous input:nil.
startGoTo(hector)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
pickUp(2)
Exogenous input:nil.
Exogenous input:orderShipment(3,yves,mike).
Exogenous input:nil.
Exogenous input:nil.
acknowledge(3,yves)
Exogenous input:nil.
```

```
path :  [dropOff(1), startGoTo(yves), sim(reachDest), pickUp(3),
startGoTo(mike), sim(reachDest), dropOff(2), dropOff(3)] in
[acknowledge(3,yves), orderShipment(3,yves,mike), pickUp(2), ...]
```

At this point we can see that the reactive thread was able to detect the arrival of the new shipment order `orderShipment(3,yves,mike)` and then acknowledge it to the sender. Since there was a new shipment, the main thread of the control program replanned and found a new path that can be used to serve all the shipments. This shows that the thread for detecting new orders can be executed together with the main control thread that involves planning/search. The following is the rest of the trace:

```
Exogenous input:nil.
dropOff(1)
Exogenous input:nil.
Exogenous input:nil.
startGoTo(yves)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
pickUp(3)
Exogenous input:nil.
Exogenous input:nil.
startGoTo(mike)
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:reachDest.
Exogenous input:nil.
Exogenous input:nil.
Exogenous input:nil.
dropOff(2)
Exogenous input:nil.
Exogenous input:nil.
dropOff(3)
Exogenous input:nil.
Exogenous input:nil.
stop_interrupts
Exogenous input:nil.
```

Chapter 5

Meta-Level Plan Finding and Execution

Searching for a way to accomplish a complex task can be very time consuming. Depending on the complexity of the problem domain, a robot may need to follow a plan which consists of hundreds of actions in order to finish its job. However, the amount of time that can be spent on planning is often limited and the robot may have to make a decision quickly in order to be responsive in a dynamic environment. It may be necessary to abort planning and react in the best available way.

In other cases, we may want the robot to build a plan, but not execute it at all. The plan may be used only for determining whether the robot is suitable for accomplishing a specific task. Also, the robot may want to generate several plans and select the best one among all the possible plans it has to execute. As well, during execution, the robot may need to change its behavior and drop its current plan. For example, a robot which relies on its battery may notice that it does not have enough power to finish all the remaining actions in its plan. It has to abort the execution of its current plan at some point and go back to recharge its battery, and then complete its task.

To support these, a robot controller needs to have control over when and how plans are constructed and executed; it needs to be able to:

- abort the search for a plan then the total search time has exceeded a given limit or some other restriction has been violated;
- retrieve the content of a plan once it has been found, and evaluate and compare it to other plans;
- stop the execution of a particular plan, because of the occurence of an exogenous action or because a better plan has become available.

As we have seen in the previous chapter, IndiGolog does planning by looking ahead for a sequence of transitions that starts from the given program and leads to a final situation. This sequence is stored within the search block internally and the user has no access to it. The interpreter keeps following it and does replanning automatically when an unexpected exogenous action occurs. To improve the flexibility and efficiency of planning and plan execution in the language, we develop meta-level routines for planning and plan execution that can be used in control programs. As the result, the sequences of transitions found by the interpreter are now accessible at the user level and the user can decide which sequence to execute and when to do so. An extended example which uses these facitities is presented in Chapter 6. The meta-level facilities developed in this chapter are a prototype. More work is required to fully address the problems of interruptable and flexible plan generation and execution.

5.1 A Meta-Level Plan Finding Routine

Unlike the object-level search block construct, our meta-level planning predicate, $findpath$, generates a plan for executing a given program in a given starting history and returns it without executing it. Moreover, the search will be aborted if a given condition becomes true, which can be exceeding a time limit. By a plan, we mean a sequence of transitions that starts from the given program and history and leads to some final configuration. The predicate is a generalization of the $findpath$ predicate seen in section 4.4. It can be used for finding an initial plan or for replanning when the environment has changed.

For simple path planning, the predicate is called as:

$findpath([], E0, H0, Abort\_Cond, Path)$

where

$E0$ is the IndiGolog program for which one wants to get an execution path
$H0$ is the initial history
$Abort\_Cond$ is a condition for aborting the search
$Path$, which will be bound to a sequence of transitions (a path) that is an execution of $E0$ in $H0$, or [] when the search is aborted.

For replanning, it is called as:

$findpath(Actions, E0, H0, Abort\_Cond, Path)$

where

Actions is a list of actions that have already occured since $E0$ began executing in $H0$ (labelled by *exactout* and *exactin*)
$E0$ is the initial program
$H0$ is the intital history
Abort_Cond is a condition for aborting the search
Path, which will be bound to a sequence of transitions (a path) that is an execution of $E0$ in $H0$ that includes the actions that have already occured, or [] when the search is aborted.

This predicate searches for a sequence of transitions in the same way that the object-level search mechanism does, except that it aborts the search when the condition *Abort_Cond* becomes true. In this case, it returns an empty list [] as *Path* to indicate the failure.

For example, suppose that we have two robots that must deliver some shipments. Perhaps a new job should be assigned to the robot which would need to extend its route by the shortest distance to deliver the existing shipments and the new shipment. We can do this by having the robots bid for the new shipment, based on the extra distance they would have to travel. In such a case, each robot has to plan a path to determine the (minimal) extra distance it would need to travel. The one who needs to travel the shortest distance can be granted the contract and can execute its path. The other robots can simply continue with their existing paths. The following program fragment shows how this can be done using *findpath*; it finds the total distance that a robot must travel to serve the existing shipments and the new shipment within a 30-second time bound:

```
$(\pi\ p, t, d)$
    [ $?(systemTime = t)$ ;
      $?(findpath([], minimizeMotion(0), now, timeOut(t, 30), p)$ ;
      $if\ (p \neq [])\ then$        $/*\ a\ path\ is\ found\ */$
         [ $?(d = pathLength(p, robotPos))$ ;
           $bid(d)$
         ]
    ]
```

In this, p is the path found by $findpath$ and d is the total distance the robot has to travel to follow the path; the function $pathLength$ computes this distance. The functional fluent $systemTime$ returns the current time in seconds. By setting the abort condition to $timeOut(t, 30)$, the search must be done within 30 seconds after the starting time t; otherwise $findpath$ returns an empty list [] through the argument p after this amount of time. The predicate $timeOut$ is defined as follows:

```
$proc\ timeOut(start, limit)$
      $systemTime - start > limit$
$end$
```

The implementation of $findpath$ is as follows:

```
/* the list of performed actions is empty */
findpath_cond([],E,H,AbortCond,[E,H]) :- final(E,H).
findpath_cond(AL,E,H,AbortCond,[]) :- holds(AbortCond,now), !.
findpath_cond([],E,H,AbortCond,P) :-
    trans(E,H,E1,H1), findpath_cond([],E1,H1,AbortCond,L),
    ((L = [], P = []) ; P = [E,H|L]).

/* some actions in the list of performed actions */
findpath_cond([A|AL],E,H,AbortCond,[]) :- holds(AbortCond,now), !.
findpath_cond([inside(A)|AL],E,H,AbortCond,L) :-
    prim_action(A), trans(E,H,E1,[A|H]),
    findpath_cond(AL,E1,[A|H],AbortCond,L).
findpath_cond([A|AL],E,H,AbortCond,L) :-
    A \= inside(_), findpath_cond(AL,E,[A|H],AbortCond,L).
findpath_cond([A|AL],E,H,AbortCond,L) :-
    trans(E,H,E1,H), findpath_cond([A|AL],E1,H,AbortCond,L).
```

The code is essentially the same as the one for `findpath` presented in section 4.4, except that the abort condition is checked each time a transition is added to the path. This provides some degree of interruptability. However, note that finding a transition may require a large amount of time, so there is no guarantee that the predicate will return very soon after the abort condition becomes true.

5.2 A Meta-Level Path-Following Procedure

To execute a path that has been found by $findpath$, we provide a procedure *executepath* that takes a path and keeps following it until it is done or an exogenous action occurs that makes it invalid. The procedure is called as follows:

$executepath(Path, E0, H0, I, AbortCond, SimCond, Remainder)$

where

$Path$ is a sequence of transitions (a path)
$E0$ is the initial program for which $Path$ is an execution in $H0$
$H0$ is the initial history
I is the list of snapshots
$AbortCond$ is a condition for aborting the path execution

SimCond is a condition to wait for when it encounters a simulated action
Remainder is output of this routine, which can be $remain(E0, H0, I')$,
$rpath(Path', E0, H0, I')$ or [].

It takes the *Path*, the initial program $E0$, the initial history $H0$, and the list of snapshots I (described in section 4.4) which tells what actions have been done by $E0$ since $H0$ and then follows the given path *Path* transition by transition until the path is completed or the condition *AbortCond* becomes true. Whenever it encounters a simulated action during the path execution, it blocks until the condition *SimCond* holds.

This procedure returns the status of its execution through the argument *Remainder* at the time it stops. *Remainder* can be either a *remain(E0,H0,I')* structure, a $rpath(Path', E0, H0, I')$ structure, or an empty list []. The *remain* structure is returned if the given path becomes invalid. It contains information to perform replanning. The *rpath* structure contains the remaining path $Path'$ that has not been executed and is returned when *AbortCond* becomes true and the path execution is aborted before it can complete the whole path. If the path has been completed successfully, the routine returns an empty list [].

For example, suppose the robot must pick up all the shipments from the central office and has only 5 minutes to deliver them, and it must go back to the central office to pick up the new shipment when a new order arrives. The following program fragment show how this can be done using *executepath*:

```
$while\ ((\exists c) clientToServe(c))$
    $(\pi p, h, t0, t1, r)$
      [ $?(systemTime = t0 \wedge h = now \wedge$
          $findpath([], minimizeMotion(0), h, timeOut(t0, 300), p)$ ;
        $if\ (p \neq [])\ then$        $/*\ a\ path\ is\ found\ */$
          [ $?(systemTime = t1)$ ;
            $executepath(p, minimizeMotion(0), [], h,$
                          $timeOut(t1, 300 - (t1 - t0)), true, r)$ ;
            $goto(central\_office)$
          ]
      ]
```

In this program, p is the path found by $findpath$, h is the history when $findpath$ starts trying to search for a path (it is bound to the current history now), and $t0$ and $t1$ are the time when $findpath$ and $executepath$ are called. When there are some clients to be served and a path is found ($p \neq []$), this program calls $executepath$ to execute the path p that is found by $findpath$. Since finding a path requires time, the amount of time left for the robot to follow the path and deliver the shipments is $300-(t1-t0)$ seconds. When the robot has finished the delivery, or a new shipment order arrives which makes the robot unable to follow the path, or the time limit has been reached, the execution of $executepath$ stops and then the robot goes back to the central office by performing the $goto(central\_office)$ action.

The implementation of this path execution routine follows the approach used in implementing the object-level search block construct. It checks to see if the path is still valid before it takes the next transition of the path. For efficiency, $executepath$ executes a segment of a path which contains one action of the path instead of a single transition at each iteration. To do this, the path is divided into segments and the routine executes all transitions in one segment at each step. A path segment is defined as a sub-sequence of the transitions of the given path where no action is involved in all but the last transition (i.e. following the transitions does not change the situation), and either the last transition is the last one of the given path and does not involve any action, or it involves a single action. The predicate **finalpathseg(P)** is used for identifying the first case. It holds on a path P if no action is involved in any transition of P.

```
finalpathseg([E,H]).
finalpathseg([E,H,E1,H|L]) :- finalpathseg([E1,H|L]).
```

We also define the predicate **transpathseg(P,PS,A)**, which takes a path P and returns the remaining path PS obtained by removing the first path segment from P, and the action A involved in the last transition of the removed segment. It is defined as follows:

```
transpathseg([E,H,E1,H|L],PS,A) :- transpathseg([E1,H|L],PS,A).
transpathseg([E,H,E1,[A|H]|L],[E1,[A|H]|L],A).
```

We also define the predicate **posspath(P,CH)**, which takes a path P and a history CH and finds out whether it is possible to execute P in CH (i.e. whether the path's execution in CH would terminate at some legal final configuration). It is implemented as follows:

```
posspath([E,H],CH) :- final(E,CH).
posspath([E,H,E1,H|L],CH) :- trans(E,CH,E1,CH), posspath([E1,H|L],CH).
posspath([E,H,E1,[A|H]|L],CH) :-
    trans(E,CH,E1,[A|CH]), posspath([E1,[A|H]|L],[A|CH]).
```

The procedure *executepath* takes a path and keeps following it segment by segment; it is implemented as follows:

```
$proc\ executepath(Path, E0, H0, I, AbortCond, SimCond, Remainder)$

    $/*\ check\ if\ the\ path\ has\ been\ completed\ */$
    $if\ (finalpathseg(Path))\ then$
        $if\ (posspath(Path, now))\ then$
            $?(Remainder = [])$
        $else$
            $?(Remainder = remain(E0, H0, I)$

    $else\ if\ (AbortCond)\ then$
        $?(Remainder = rpath(Path, E0, H0, I)$

    $/*\ try\ to\ take\ the\ next\ segment\ */$
    $else$
        $(\pi\ p, a)$
            $[\ ?(transpathseg(Path, p, a));$

              $if\ (a = sim(\_) \wedge posspath(Path, now))\ then$
                  $[\ ?(SimCond);\ //\ sim\ action\ handled\ by\ executor$
                    $executepath(p, E0, H0, I, AbortCond, SimCond,$
                                $Remainder)$
                  $]$

              $else$
                  $if\ (posspath(Path, now))\ then$
                      $(\pi\ I')$
                          $[\ //\ perform\ a\ and\ update\ LS$
                            $if\ (I' = I \cup do(a, now))\ then\ a;$
                            $executepath(p, E0, H0, I', AbortCond,$
                                        $SimCond, Remainder)$
                          $]$
                  $else$
                      $?(Remainder = remain(E0, H0, I)$
        $]$
$end$
```

The procedure first checks to see whether the given path is actually a final path segment. In this case, if it is possible to perform all the transitions of this final path segment in the current histoy, then *executepath* returns [] which indicates the path execution has been completed successfully; otherwise, it returns a *remain* structure to indicate the failure.

If on the other hand, the given path consists of more than one segment, then *executepath* first finds the action that must be done in order to do all transitions in the first path segment. If this action is simulated, then it can continue to follow the path when the condition to handle simulated actions is satisfied. If it is a primitive action and it is possible to follow the path, then it performs the action, updates the list of snapshots and continues following the remaining path. In the case where it is impossible to follow the path, it returns a *remain* structure.

Note that checking whether a path is executable or finding a transition may require a large amount of time. So, as with *findpath*, there is no guarantee that the predicate will return very soon after the abort condition becomes true.

5.3 Meta-Level Path Planning and Path Execution

Using the meta-level planning and path execution routines, we have defined a procedure *msearch* which does path planning and then path execution more or less as the interpreter does for a search block, and also provides more control over planning and execution. In particular, *msearch* takes not only an input program but also an abort condition to stop its execution and a condition to wait for when it encounters a simulated action during path execution.

Given a program, the procedure is called as follows:

$msearch(E, AbortCond, SimCond, Remainder)$

where

E is the initial program
$AbortCond$ is a condition for aborting the execution of E
$SimCond$ is a condition to wait for when it encounters a simulated action
$Remainder$ is output of this procedure, which can be $remain(E0, H0, I)$, $rpath(Path, E0, H0, I)$ or []. $E0$ and $H0$ are the initial program and initial history for which the procedure is called; I is the list of snapshots; $Path$ is the path that remains of the execution of $E0$.

Like *executepath*, *msearch* returns a *Remainder* at the end to indicate the final status of its execution. The value of this can be either a $remain(E0, H0, I')$ structure if no path can be found for the given program and history, $rpath(Path', E0, H0, I')$ if it is aborted, or an empty list [] if the given program has been executed successfully to the end.

To allow the execution of *msearch* to be resumed after it is aborted (previous call to *msearch* returned a *remain* or *rpath* structure), we define two other versions of *msearch* that take either a *remain* or *rpath* structure instead of an initial program. They are called as follows:

$msearch(rpath(Path, E0, H0, I), AbortCond, SimCond, Remainder)$
$msearch(remain(E0, H0, I), AbortCond, SimCond, Remainder)$

where

> $Path$ is a sequence of transitions left in $rpath$
> $E0$ is the initial program
> $H0$ is the initial history
> I is the list of snapshots
> $AbortCond$ is a condition for aborting the execution of E
> $SimCond$ is a condition to wait for when it encounters a simulated action
> $Remainder$ is output of this procedure, which can be $remain(E0, H0, I')$, $rpath(Path', E0, H0, I')$ or []. $E0$ and $H0$ are the initial program and initial history for which the procedure is called; I' is the list of snapshots; $Path'$ is path that remains of the execution of $E0$.

The *msearch* procedure is defined as follows:

```
proc $msearch(E, AbortCond, SimCond, Remainder)$
    $msearch(E, now, [], AbortCond, SimCond, Remainder)$
end

proc $msearch(remain(E0, H0, I), AbortCond, SimCond, Remainder)$
    $msearch(E0, H0, I, AbortCond, SimCond, Remainder)$
end

proc $msearch(E0, H0, I, AbortCond, SimCond, Remainder)$
    $(\pi\ mypath)$
        [ $?(findpath(E0, H0, I, AbortCond, mypath)$,
          $if\ (mypath = [])\ then$    $//\ search\ failed$
              $?(Remainder = remain(E0, H0, I))$
          $else$
              $if\ (AbortCond)\ then$
                  $?(Remainder = rpath(mypath, E0, H0, I))$
              $else$
                  $(\pi\ status, I')$
                   [ $executepath(mypath, E0, H0, I, AbortCond,$
                                 $SimCond, status)$
```

```
                    $if\ (status = remain(E0, H0, I'))\ then$
                        $//\ execution\ failed$
                        $msearch(E0, H0, I', AbortCond, SimCond,$
                                $Remainder)\quad //\ replan$
                    $else$
                        $?(Remainder = status)$
                  ]
        ]
    $end$
```

Given the initial configuration $E0$ and $H0$, the procedure first calls $findpath$ to find a path for the given program. If the search fails and an empty list is returned by the planner ($mypath = []$), it returns a *remain* structure. If a path is found but the condition for aborting is satisfied, then it returns a *rpath* structure with the remaining path in it. Otherwise, it calls *executepath* which takes the path and keeps following it segment by segment, until a *remain* structure is returned when replanning is needed, or an empty list [] is returned in the case where the whole path has been successfully executed. *msearch* keeps trying to execute the program, replanning as necessary, as long as the execution is not completed and the abort condition remains false.

For the case where *rpath* is one of the inputs, the procedure does not search for a path at the beginning but first attempts to execute the path stored in *rpath* by calling *executepath*. It is defined as follows:

```
    $proc\ msearch(rpath(Path, E0, H0, I), AbortCond, SimCond, Remainder)$
        $(\pi\ status, I')$
            [ $executepath(Path, E0, H0, I, AbortCond, SimCond, status)$ ;
              $if\ (status = remain(E0, H0, I'))\ then$
                  $//\ execution\ failed,\ replan$
```

```
                $msearch(E0, H0, I', AbortCond, SimCond, Remainder)$
            $else$
                $?(Remainder = status)$
        ]
$end$
```

Chapter 6

A Distributed Mail Delivery Control System

Many robotics applications involve using multiple robots working on a task. One can build a centralized control system which collects information from all the robots and then generates a complete plan to control all of them. However, this centralized strategy can require a tremendous amount of computational resources and unneccessary communication. It may not be able to generate a plan and send commands back to the robots within the desired amount of time, when the robots have to act quickly to avoid problems and accomplish tasks in a dynamic environment. Also, planning may fail because of slow or unreliable communcation channels to the robots.

Another approach is to give control to the robots and have them construct plans for themselves as individuals. The main advantage of this distributed strategy is that each robot does not need to consider a lot of information and can rebuild its plan much more quickly when the environment changes.

However, the robots must coordinate their actions to ensure that each task will be handled by some robot, that two robots will not try to do a task that can be handled by only one robot, and that the robots can perform their actions without any conflict.

Since there is a large number of non-deterministic choices in the robot control program that is described in Chapter 4, the amount of time needed for path-planning grows exponentially as the total number of shipments and robots increase. In order to improve the performance of the system and test the meta-level facilities described in Chapter 5, we modified the original control system into a distributed system to control multiple robots in which each robot is governed by an independent control program. The robots communicate with each other through external links. The rest of this chapter first explores issues in building multi-robot control systems, and then shows how the meta-level facilities can be used to construct an effective distributed control system.

6.1 Overview of the System

The main issue that needs to be considered in building a distributed robot control system is how the actions of the robots can be coordinated so that they work together effectively. The simplest approach is having an identical control system on each robot with no communication between them. This means they have identical capabilities and decision procedures, but can only obtain limited information about each other through sensing. This strategy was used in [58, 2] but it was not very successful. The main problem is that the robots may try to move to the same position or compete for doing the same task, and block each other as a result. The second approach is having a master decision maker which collects all the information from the robots and decides what action each robot must perform to achieve the goal. An example of this is the "centralized strategy" described in [58]. However, collecting information and sending commands requires a large amount of communication. Another approach is one where each robot generates a portion of the plan and then they try to form a centralized plan by sharing data or doing communication. This has been used in applications such as mission planning for unmanned vehicles [14] and logistics planning [64]. A more distributed method is having multiple robots generate plans for themselves only, and then communicate to ensure that their plans have no conflicts and the task can be accomplished. Reachability analysis [24], plan combination search based on global constraints [16], and distributed hierarchical planning such as described in [15], fall under this catergory.

The method that we use in our distributed system is based on the *contract net mechanism* [60, 9] where an agent decomposes the task into sub-tasks, and then an auction is run between the robots to decide which robot will take responsibility for which sub-task. In a contract net, there are two kinds of agents:

- A manager agent who decomposes the task into sub-tasks and runs the contract granting process with the other agents submitting bids to determine who will be responsible for handling each sub-task; it also monitors the execution of each sub-task.

- Contractor agents who bid for sub-tasks and execute the sub-tasks assigned to them by the manager.

The coordination protocol used by the manager and contractors can be viewed as a kind of auction. When the manager receives a new task, it decomposes the task into sub-tasks and the following four steps will be performed in order to determine who is going to take responsibility of each sub-task:

1. The manager announces a sub-task.
2. Each contractor evaluates the given sub-task with respect to its own abilities and the resource requirements to accomplish it.
3. Each contractor makes a bid on the sub-task if it can handle it.
4. The manager assigns the sub-task to one of the contractors based on the bids it has received.

The manager and contractors work independently. A task can still be accomplished the system even if the communication link between one of the contractors and the manager is not working properly.

In our distributed shipment delivery robot control system, the role of the manager is taken by the shipment manager running on a separate machine. Each robot is a contractor and it has its own control program. There are four kinds of messages (communication actions) that can be sent across the contract net. They are:

- $callForBid(Manager, Robot, SNo, Sender, Recipient)$
  - The shipment manager sends this message to *Robot* when a new shipment order arrives. *Robot* is asked to bid for a shipment to be taken from *Sender* to *Recipient*, which is assigned the number SNo.
- $bid(Robot, Manager, SNo, Value)$
  - The robot *Robot* makes a bid on the shipment SNo by sending this message to the shipment manager. $Value$ is an estimation of the amount of work that the robot needs to do in order to serve the shipment.
- $award(Manager, Robot, SNo, Sender, Recipient)$
  - The shipment manager sends this message to *Robot* if it chooses *Robot* to serve the shipment SNo.
- $report(Robot, Manager, SNo, Message)$
  - *Robot* reports to the shipment manager that the shipment SNo has been delivered to the recipient.

When the shipment manager receives a new shipment order, it announces the new shipment by sending the message $callForBid$ to each robot. Then, each robot estimates the amount of work required to serve the shipment and sends back bids to the manager. Since computing the exact amount of work needed can be time-consuming, each robot is allowed to make several bids for a new shipment. The bid can be a conservative one which can be made quickly, or a more competitive one, which requires more computation to generate. The manager collects the bids from the robots and picks the one which has the smallest bidding value to serve the shipment. It sends a message awarding the task to the robot who has been chosen to serve the shipment, and then waits until the robot reports that the shipment has been delivered. Detailed descriptions of the shipment manager and the individual robot control program are presented in Sections 6.2 and 6.3.

Researchers have explored the use of contract net for a variety of problems, such as the coordination of distributed sensor networks [9], load balancing on operating system [55], and robot shipment delivery with case-based reasoning [46]. Note that our system does not use a feature of the original contract net mechanism, which is that an agent can be both a manager and a contractor simultaneously. It could be a manager for one task and a contractor for another task. Since delivering a shipment from one place to another place is the only task in our application, there is no benefit to having several shipment managers or combining a manager with each individual robot control program. This could be useful if there were many robots and each robot could communicate with a small group of robots only.

6.2 The Shipment Manager

The shipment manager is responsible for receiving new shipment orders from the clients and determining which robot is the best to serve each shipment by analyzing the bids returned by the robots. The status of a shipment is maintained in the primitive fluent $shipmentState(SNo)$, and its value can be one of the following:

- *nonExistent* - the shipment does not exist yet; this is the initial value.
- *justIn* - the order for the shipment has been received by the shipment manager, but the manager has not announced it to the robots yet.
- *askedAllRobots* - the shipment has been announced to all robots, but the shipment manager has not decided who should serve it.
- *assigned* - the shipment has been assigned to a robot, but it has not yet been delivered.
- *rejected* - the shipment has been rejected by the shipment manager since no robot is available to serve it.
- *delivered* - the shipment has been delivered by a robot.

There are three new primitive actions and two new exogenous actions that can affect the status of a shipment:

Primitive actions:

- $callForBid(Manager, Robot, SNo, Sender, Recipient)$ - the shipment manager asks *Robot* to make a bid on shipment SNo, where *Sender* and *Recipient* are the sender and recipient of the shipment.
- $award(Manager, Robot, SNo, Sender, Recipient)$ - the shipment manager asks *Robot* to serve shipment SNo, where *Sender* and *Recipient* are the sender and recipient of the shipment.

- $rejectShipment(Manager, Sender, SNo)$ - the shipment manager rejects shipment order SNo from the client $Sender$.

Exogenous actions:

- $shipmentRequest(Sender, Recipient)$ - the client $Sender$ makes a shipment order for which the recipient is $Recipient$.

- $report(Robot, Manager, SNo, Message)$ - $Robot$ reports the status of shipment SNo.

To decide who should be awarded a contract, the shipment manager uses a primitive fluent $hasBid(Robot, SNo)$ which records the latest bid value made by a particular robot on a shipment. There is also a primitive fluent called $totalJobsServing(Robot)$ that stores the total number of shipments to be served by a robot. The manager receives a bid made by a robot through the occurrence of the exogenous action $bid(Robot, SNo, Value)$.

A timing scheme is used to ensure that the shipment manager selects a robot to serve a shipment within a reasonable amount of time. The fluent *counter* acts a system clock and performing the primitive action *tick* increases the counter's value by 1. This action is done periodically during the program execution. The primitive fluent $callAtTime(Robot, SNo)$ records the time when shipment SNo is announced to a robot.

A complete list of the action precondition axioms and successor state axioms for the shipment manager is provided in Appendix B. As an example, the precondition axiom of action *award* is as follows:

$$
\begin{aligned}
&Poss(award(Manager, Robot, SNo, Sender, Recipient), s) \;\equiv \\
&\qquad shipmentState(SNo, s) = askedAllRobots \;\wedge \\
&\qquad hasBid(Robot, SNo, s) > -1 \;\wedge \\
&\qquad shipmentSender(SNo, s) = Sender \;\wedge \\
&\qquad shipmentRecipient(SNo, s) = Recipient
\end{aligned}
$$

This says that $Manager$ can ask $Robot$ to handle shipment SNo at situation s if this shipment has been announced to all robots, $Robot$ has already made a bid for it, and $Sender$ and $Recipient$ are the sender and recipent of the shipment.

As another example, the successor state axiom of the fluent $shipmentState$ is as follows:

$$
\begin{aligned}
&shipmentState(SNo, do(a, s)) = x \equiv \\
&\qquad ((\exists c1, c2)a = shipmentRequest(c1, c2) \wedge counter = SNo \wedge \\
&\qquad\;\; x = justIn) \vee \\
&\qquad ((\exists r, c1, c2)a = callForBid(manager, r, SNo, c1, c2) \wedge \\
&\qquad\;\; shipmentState(SNo, s) = justIn \wedge \\
&\qquad\;\; \neg((\exists r')r' \neq r \wedge callAtTime(r', SNo) \neq currentTime) \wedge \\
&\qquad\;\; x = askedAllRobots) \vee \\
&\qquad ((\exists r, c1, c2)a = award(manager, r, SNo, c1, c2) \wedge x = assigned) \vee \\
&\qquad ((\exists c1)a = rejectShipment(manager, SNo, c1) \wedge x = rejected) \vee \\
&\qquad ((\exists r)a = report(r, manager, SNo, completed) \wedge x = delivered) \vee \\
&\qquad ((\forall r, c1, c2)a \neq shipmentRequest(c1, c2) \wedge \\
&\qquad\;\; a \neq callForBid(manager, r, SNo, c1, c2) \wedge \\
&\qquad\;\; a \neq award(manager, r, SNo, c1, c2) \wedge \\
&\qquad\;\; a \neq rejectShipment(manager, SNo, c1) \wedge \\
&\qquad\;\; a \neq report(r, manager, SNo, completed) \wedge \\
&\qquad\;\; x = shipmentState(SNo, s))
\end{aligned}
$$

The first case in this axiom says that when someone makes a shipment request, an unique number (the value of $counter$) is assigned to the shipment as its shipment number and its state becomes $justIn$. The second case states that the state of the shipment becomes *askedAllRobots* if it was just in and the manager has just announced it to the last robot that did not know of the existence of the shipment yet. If the manager assigns the shipment to a robot (by doing *award*), then the third case says that the state of the shipment becomes *assigned*. The fourth and the fifth cases change the state of the shipment to *rejected* or *delivered* when it is rejected by the manager or delivered to the recipient. The last case says that a shipment's state is unaffected by other actions.

We also have the following successor state axiom for fluent $totalJobsServing$:

$$\begin{array}{l}
totalJobsServing(Robot, do(a,s)) = n \equiv \\
\quad ((\exists o, c1, c2)a = award(manager, Robot, o, c1, c2) \wedge \\
\quad\;\; n = totalJobsServing(Robot, s) + 1)) \vee \\
\quad ((\exists o)a = report(Robot, manager, o, completed) \wedge \\
\quad\;\; n = totalJobsServing(Robot, s) - 1) \vee \\
\quad ((\forall o, c1, c2)a \neq award(manager, Robot, o, c1, c2) \wedge \\
\quad\;\; a \neq report(Robot, manager, o, completed) \wedge \\
\quad\;\; n = totalJobsServing(Robot, s)
\end{array}$$

This axiom say that the total number of shipments that are being served by *Robot* increases when the manager assigns a new shipment to it, and decreases when the robot reports that it has delivered a shipment.

For simplicity, we introduce two additional fluents which are defined in terms of the primitive fluents. The fluent $allHaveBidded(N)$ holds if all robots have already bidded on shipment N:

```
proc $allHaveBidded(N)$
    $\neg((\exists r)hasBid(r,N) = -1)$    $//\ initial\ value\ of\ hasBid\ is\ -1$
end
```

The fluent $timeOut(N)$ holds if shipment N was announced to the robots at least 3 units of time ago:

```
proc $timeOut(N)$
    $(\exists r)callAtTime(r,N) \neq -1 \wedge$
        $currentTime - callAtTime(r,N) > 3$
end
```

Basically, the shipment manager has two jobs to do. It has to broadcast messages to the robots when a client makes a shipment order (the state of some shipment is $justIn$), and it needs to determine which robot should serve a shipment after the bids are collected ($allHaveBidded$ holds) or a predefined amount of time has passed ($timeOut$ holds). Each of these is implemented as a separate thread in the shipment manager control program. There is also a third thread in the control program with the lowest priority which keeps incrementing the system time by performing the $tick$ action periodically. Combining all the threads together, the program of the shipment manager is implemented as follows:

```
proc $manager$
    $< (\exists n1)shipmentState(n1) = justIn \rightarrow$
        $handleNewRequest(n1) >$
    $\gg$
    $< (\exists n2)(shipmentState(n2) = askedAllRobots \wedge$
        $allHaveBidded(n2)) \vee$
```

```
            $(shipmentState(n2) = askedAllRobots \ \wedge \ timeOut(n2)) \rightarrow$
          $decideContractor(n2) >$
      $\gg$
      $< True \rightarrow tick >$
end
```

The first thread with the highest priority checks to see if there is a new shipment order from the clients. Whenever there is a new shipment ($shipmentState(n1) = justIn$), it calls the complex procedure $handleNewRequest(n1)$ to handle it. The procedure sends the message $callForBid$ to all robots:

```
proc handleNewRequest(N)
      if ((∃r)connectedToManager(r) = yes) then
            while(shipmentState(N) ≠ askedAllRobots)
                  (π r) [ // no callForBid sent yet
                              ?(callAtTime(r, N) ≠ −1);
                              callForBid(r, N, shipmentSender(N),
                                          shipmentRecipient(N))
                        ]
      else
            /∗ reject if no robot is available ∗/
            rejectShipment(N, shipmentSender(N))
end
```

The second thread in the manager control procedure calls the procedure $decideContractor$ to assign the new shipment N to the robot which can deliver it by doing the least amount of additional work:

```
$proc\ decideContractor(N)$

    $if\ ((\exists r, v)(v = hasBid(r, N) \wedge v > -1)\ then$

        $/ * if\ someone\ has\ bidded,\ find\ contractor * /$
        $(\pi\ r)$
            $[\ ?(bestRobot(r, N))\ ;$
              $award(manager, r, N, shipmentSender(N),$
                    $shipmentRecipient(N))$
            $]$

    $else$
        $/ *\ if\ no\ one\ has\ bidded,\ reject\ shipment\ * /$
        $rejectShipment(manager, shipmentSender(N), N)$
$end$

$proc\ bestRobot(R, N)$
    $(\exists\ v1, j1)$
        $(v1 = hasBid(R, N) \wedge v1 \neq -1 \wedge$
         $j1 = totalJobsServing(R) \wedge$
         $\neg((\exists r2, v2, j2) v2 = hasBid(r2, N) \wedge v2 \neq -1 \wedge$
             $j2 = totalJobsServing(r2) \wedge$
             $(v2 < v1 \vee (v2 = v1 \wedge j2 < j1)))$
$end$
```

The best robot to serve the new shipment N is the one that has made the smallest bid on the shipment among all bids made by the other robots. In case of a tie, the one that is serving the fewest number of shipments gets the job. If no robot has made a bid on the shipment, the shipment manager rejects it.

6.3 The Individual Robot Controller

The structure of the individual robot control program is similar to the one used for the single robot shipment delivery application, except that it also contains a section for making bids on new shipments. When the shipment manager announces a new shipment, the robot's bid generator has to evaluate its ability to handle the shipment and make a bid if possible. The whole program is divided into three main parts: a quick bidder that can make a rather high or conservative bid on a new shipment quickly, a slow bidder that requires a long period of time to find a lower bidding value that reflects the true cost of serving the new shipment, and also a motion control thread that drives the robot to deliver the shipments using the shortest route according to the service commitments that the robot has already made.

When a robot is serving some clients and a new shipment order arrives, the robot may need to adjust its route and travel a longer distance to deliver all shipments including the new one if it is awarded to it. The bidding value for a shipment is set to be the extra distance that the robot must travel in order to deliver it. The first bid generator *slowBid* finds the minimal extra distance that the robot must travel in order to serve the new shipment. Finding the minimal value requires finding the shortest route to serve the new and the existing shipments. However, this can be very time consuming. This bid generator may not be able to send back a value to the shipment manager within the requried amount of time. Because of this, the robot control program has another bid generator called *quickBid*, which makes a rough guess of the bidding value for a new shipment without finding the shortest route. It generates a bidding value quickly but this value may be unnecessarily high and not reflect the real cost of serving the new shipment.

6.3.1 The Quick Bidder

The procedure *quickBid* estimates the distance that the robot must travel to deliver a new shipment by assuming that the robot will serve this shipment when it has finished all its current jobs. We introduce a fluent $finalRobotPos$, whose value is the final position of the robot when it finishes delivering all the shipments it is already committed to serve. Once the robot motion control thread has found a path for serving all the allocated shipments, it scans the path and records the final position of the robot by performing the action *setFinalPosition* so that it can be used by *quickBid*. The successor state axiom for $finalRobotPos$ is:

$$\begin{aligned}
&finalRobotPos(do(A,s)) = Place \equiv \\
&\qquad A = setFinalPosition(Place) \vee \\
&\qquad (\forall p)\ A \neq setFinalPosition(p) \wedge Place = finalRobotPos(s)
\end{aligned}$$

The quick bidder makes a bid for a new shipment N with the bidding value being the sum of the distance from the final position of the robot to the sender of the new shipment and the distance from the sender to the recipient of the shipment:

```
proc $quickBid(R, N)$
    $(\pi\ p1, c2, c3, d3)$
        [ $?(finalRobotPos = p1 \wedge$
            $shipmentSender(N) = c2 \wedge shipmentRecipient(N) = c3 \wedge$
            $d3 = distance(p1, c2) + distance(c2, c3));$
```

```
                $bid(R, N, d3)$
            ]
    $end$
```

6.3.2 The Slow Bidder

The slow bidder *slowBid* calculates a more accurate bidding value for a new shipment by comparing the shortest path for the robot to deliver all existing shipments with the shortest path that can be used to deliver all existing shipments plus the new one. The difference between the distances of these two paths is the minimal extra distance that the robot has to travel to serve the new shipment, and this will be the bid value. This bid generator is written as the following procedure:

```
proc $slowBid(R, N)$
    $(\pi\ ch, pos, t, d1, d2)$
        [  $?(ch = now \wedge pos = robotPos \wedge current\_time(t));$

          $/ * total\ no.\ of\ steps\ to\ serve\ the\ existing\ shipments * /$
          $(\pi p1)\ ?(findpath([],$
                              [ $abortGoTo(R);$
                                $minimizeMotion(R, 0) || envSimulator(R)$ ],
                              $ch,$
                              $bid\_is\_invalid \vee timeOut(t)),$
                              $p1) \wedge$
                  $pathLength(p1, pos, d1))$

          $/ * total\ no.\ of\ steps\ to\ serve\ the\ existing\ +\ new\ shipments * /$
          $(\pi p2)\ ?(findpath([],$
                              [ $pi(c1, c2)$
                                  [ $?(shipmentSender(N) = c1 \wedge$
                                      $shipmentRecipient(N) = c2);$
                                  $sim(award(R, N, c1, c2))$ ];
                                $ackShipment(R, N)$;
                                $abortGoTo(R)$;
                                $minimizeMotion(R, d1) || envSimulator(R)$ ],
                              $ch$,
                              $bid\_is\_invalid \vee timeOut(t)$,
                              $p2) \wedge$
                $pathLength(p2, pos, d2))$

          $/ * calculate\ the\ bid\ value * /$
          $if(bid\_is\_invalid \vee timeOut(t) \vee d1 = -1 \vee d2 = -1)$
              $stopBidding(R, N)$
          $else$
              $bid(R, N, d2 - d1)$
        ]
$end$
```

The first call on $findpath$ tries to find the shortest distance that must be travelled to serve all existing shipments. This is done by first aborting the current path and then generating a new path. Note that this does not really abort the robot's motion because the new path is never executed. The search on the path is stopped if the condition $(bid\_is\_invalid \vee timeOut(t))$ becomes true. The condition $bid\_is\_invalid$ becomes true when an exogenous action such as $getStuck$ occurs during the search. In this situation the bidder must restart the search in order to consider the new exogenous action. The condition $timeOut(T)$ becomes true when the current time reaches T plus 60 seconds; thus, the search is aborted when the planner cannot find an answer within 60 seconds.

The second call on $findpath$ first hypothesizes that the new shipment is assigned to the robot and then tries to find the shortest distance that it has to travel to deliver all existing shipments plus the new one. Here too, the time limit is set to 60 seconds, and no exogenous action can occur during the search. If both paths are found, then bidder makes a bid on the new shipment using the difference of distances between these two paths; otherwise, it does not make a bid.

6.3.3 Robot Motion Control

The main control program consists of four threads and is written as follows:

```
$proc\ control(R)$
    $< (\exists n1) shipmentState(n1) = justAwarded \rightarrow$
        $handleNewOrder(R, n1) >$
    $\gg$
    $< (\exists n2) shipmentState(n2) = requested \rightarrow$
        $quickBid(R, n2) >$
    $\gg$
    $< (\exists c) clientToReach(c) \rightarrow motionControl(R) >$
    $\gg$
    $< (\exists n3) shipmentState(n3) = replied(1) \wedge$
        $(robotState = moving \vee \neg((\exists c3) clientToReach(c3))) \rightarrow$
        $slowBid(R, n3) >$
    $\gg$
    $< True \rightarrow no\_op >$    $//keep\ running$
$end$
```

The first thread stops the robot motion and sends back an acknowledgement when a shipment is awarded to the robot. When the shipment manager announces a new shipment, the second thread calls the quick bidder to make a bid on the shipment immediately. The third thread is the robot motion control which controls the robot to deliver all shipments using the shortest path. When the shipment manager has annnounced a new shipment and the robot is moving and has nothing else to do, then the fourth thread with lower priority calls the slow bidder to compute the minimal extra travelling distance as the best bidding value for the new shipment. The last thread prevents the controller from terminating when there is absolutely nothing to do.

The thread which controls the robot's motion also uses the $minimizeMotion$ procedure described in Chapter 4 to find the shortest path to deliver all shipments that are assigned to the robot. The following procedure is the implementation of the robot motion control:

```
$proc\ motionControl(R)$
    $(\pi\ path)$
        $[\ msearch(minimizeMotion(R,0)||envSimulator(R),$
                $pathFound(mypath),$
                $unblockSimCond,$
                $path),$

          $/*\ save\ the\ final\ position\ of\ the\ robot\ */$
          $(\pi\ fpos, p, d, h, i)$
             $[\ ?(rpath(p,d,h,i) = path\wedge$
                  $findFinalPosition(p, robotPos, fpos),$
               $setFinalPosition(fpos)$
             $]$

          $/*\ execute\ the\ path\ */$
          $followRoute(path)$
        $]$
$end$
```

This procedure first calls $msearch$ to find a $path$ for delivering the shipments assigned to the robot R. Since the slow bidder needs to know the final position of the robot when it completes following the path, the procedure uses the condition $pathFound(mypath)$ to stop the execution of $msearch$ when it has found a path but has not tried to execute it. The term $mypath$ is a reserved keyword for the program to refer to the path found by the $msearch$ in the abort condition. The procedure then looks for the final position $fpos$ from the path by performing $findFinalPosition(p, robotPos, fpos)$. Once the final position is saved in the fluent $finalRobotPos$, the path is then passed to procedure $followRoute$ for execution. The procedure $followRoute$ is almost the same as $motionControl$ and it is implemented as follows:

```
proc $followRoute(Path)$
    $(\pi\ rm)$
        [ $msearch(Path, pathFound(mypath), unblockSimCond, rm)$,
          $if(rm = [],$
              $no\_op$,  $/*\ succeeded\ */$
          $else,$
              [ $/*\ save\ the\ final\ position\ of\ the\ robot\ */$
                $(\pi\ fpos, p, d, h, i)$
                    [ $?(rpath(p, d, h, i) = rm \wedge$
                          $findFinalPosition(p, robotPos, fpos))$,
                      $setFinalPosition(fpos)$
                    ]

                $/*\ execute\ the\ new\ path\ */$
                $followRoute(rm)$
              ]
        ]
end
```

This procedure first attempts to execute the given path by calling $msearch$ with it. During the path execution, the path may stop being executable (*posspath*(*path*, *now*) in *msearch* becomes false) if an exogenous action such as *getStuck* or *award* occurs. In this case *msearch* replans and finds a new path. Since changing the path may also change the final position of the robot, *msearch* is stopped and the final position is saved before the new path is executed. After this, the procedure passes the new path to another call of *followRoute* for execution.

6.4 Experiments

6.4.1 A Sample Test

Here is the trace of a sample test run on the system with two robots *rb*1 and *rb*2:

```
| ?- tracer.
rb1 :  connectToManager(rb1)
rb2 :  connectToManager(rb2)
man :  shipmentRequest(yves,hector)
man :  shipmentRequest(mike,hector)
man :  callForBid(rb1,1,yves,hector)
man :  callForBid(rb2,1,yves,hector)
man :  callForBid(rb1,2,mike,hector)
man :  callForBid(rb2,2,mike,hector)
rb1 :  bid(rb1,1,2)
rb2 :  bid(rb2,1,2)
rb1 :  bid(rb1,2,2)
rb2 :  bid(rb2,2,2)
rb1 :  bid(rb1,1,2)
rb1 :  bid(rb1,2,2)
rb2 :  bid(rb2,1,2)
rb2 :  bid(rb2,2,2)
```

There were 2 shipment orders at the beginning (1:yves-hector, 2:mike-hector). From the output we can see that the manager first sent the information to the robots and then both of them were able to bid on the shipments. Note that in this test, we assume that the distance between any two clients/places is 1.

```
man :  award(rb1,1,yves,hector)
rb1 :  ackShipment(rb1,1)
man :  award(rb2,2,mike,hector)
rb1 :  setFinalPosition(hector)
rb2 :  ackShipment(rb2,2)
rb1 :  startGoTo(rb1,yves)
rb2 :  setFinalPosition(hector)
rb2 :  startGoTo(rb2,mike)
```

At this stage, shipment 1 was assigned to rb1 while rb2 got shipment 2. Each robot controller was able to set the final position before the robot started moving to its first destination. This illustrates how the controller can perform actions after a path is found and before the path is executed.

```
man :  shipmentRequest(mike,kong)
man :  callForBid(rb1,3,mike,kong)
man :  callForBid(rb2,3,mike,kong)
rb1 :  bid(rb1,3,2)
rb1 :  bid(rb1,3,2)
rb2 :  bid(rb2,3,2)
man :  award(rb1,3,mike,kong)
rb1 :  abortGoTo(rb1)
rb1 :  ackShipment(rb1,3)
rb2 :  stopBidding(rb2,3)
```

Then, another shipment order (3:mike-kong) arrived. The quick bidder of rb2 was able to bid on it while the slow bidder did not have enough time to find a possible path. As the result, shipment 3 was assigned to rb1 even through it would have been better to give the order to rb2. It also shows that rb1 first aborted its current path and then generated a new one to handle the new shipment after it received the message from the manager.

```
rb1 :  setFinalPosition(kong)
rb2 :  reachDest(rb2)
rb2 :  pickUp(rb2,2)
rb2 :  startGoTo(rb2,hector)
rb1 :  startGoTo(rb1,yves)
man :  shipmentRequest(hector,kong)
man :  callForBid(rb1,4,hector,kong)
man :  callForBid(rb2,4,hector,kong)
rb2 :  bid(rb2,4,1)
rb1 :  bid(rb1,4,2)
rb2 :  bid(rb2,4,1)
rb1 :  bid(rb1,4,0)
man :  award(rb1,4,hector,kong)
```

Then, there is a fourth shipment order (4:hector-kong). The distance estimated by the quick bidder of rb1 for handling shipment 4 (hector-kong) was 2. However, rb1 did not need to travel any more distance to serve this new shipment because it had to reach hector and then kong in order to deliver shipments 1 and 3. So its slow bidder bided on the shipment with the value 0. Therefore, shipment 4 was also assigned to rb1. The following is the rest of the trace:

```
rb1 :  abortGoTo(rb1)
rb1 :  ackShipment(rb1,4)
rb2 :  getStuck(rb2)
rb2 :  setFinalPosition(hector)
rb2 :  startGoTo(rb2,hector)
```

```
rb1 :  setFinalPosition(kong)
rb1 :  startGoTo(rb1,yves)
rb1 :  reachDest(rb1)
rb2 :  reachDest(rb2)
rb1 :  pickUp(rb1,1)
rb2 :  dropOff(rb2,2)
rb1 :  startGoTo(rb1,hector)
rb2 :  report(rb2,2,completed)
rb1 :  reachDest(rb1)
rb1 :  pickUp(rb1,4)
rb1 :  dropOff(rb1,1)
rb1 :  report(rb1,1,completed)
rb1 :  startGoTo(rb1,mike)
rb1 :  reachDest(rb1)
rb1 :  pickUp(rb1,3)
rb1 :  startGoTo(rb1,kong)
rb1 :  reachDest(rb1)
rb1 :  dropOff(rb1,3)
rb1 :  report(rb1,3,completed)
rb1 :  dropOff(rb1,4)
rb1 :  report(rb1,4,completed)
rb1 :  disconnectFromManager(rb1)
rb2 :  disconnectFromManager(rb2)
```

When the robot rb1 got the shipment order for shipment 4, it first acknowledged the shipment and then recomputed a new path which could be used to handle all existing shipments and the new one. Then, it first went to yves to pick up shipment 1 and then moved to hector to pick up shipment 4 and drop off shipment 1. Afterward, it went to mike to pick up shipment 3. Finally it moved to kong to drop off both shipments 3 and 4. On the other side, rb2 was stuck on its way to hector to drop off shipment 2. So it replanned and tried to move to hector again. It reached hector successfully on its second attempt and then dropped off shipment 2 at the end.

6.4.2 Comparative Experiments

For comparsion, we have done experiments on three different systems: a single robot control system, a centralized control system with two and three robots, and the distributed control system described in this chapter with two and three robots. We measure the performance of the systems by finding the time used to construct a complete path for handling a given set of shipment orders. The single robot control system is the one presented in Chapter 4. Given a set of shipment orders, the system searches for the shortest route to serve them. We measure the total time used by the interpreter to construct an execution path only (the time spent on running the routine $findpath$).

Our centralized control system for use with different numbers of robots can be found in Appendix D. It is a modified version of the single robot controller that tries to find a path which minimizes the distance travelled by each robot to deliver the shipments. Its main control procedure with 3 robots is as follows:

```
proc control
    search( minimizeMotion(0) )
end

proc minimizeMotion(Max)
    ( handleRequests(robot1, Max) ||
      handleRequests(robot2, Max) ||
      handleRequests(robot3, Max) )
    |
    minimizeMotion(Max + 1)
end
```

Thus, it first tries to find a path where every robot travels 0 units of distance to serve all orders, then 1 unit, then 2 units, and so on.

To test the distributed system with different numbers of robots, each shipment order is sent to the shipment manager only when it has awarded the previous shipment order to some robot. We define the total time taken by the system to contruct a complete path for handling all shipments as the time from when the shipment manager first asks the robots to bid on the first shipment to when the robot who got the last shipment finishes to construct a path to handle all the shipments which have been awarded to it. Notice that the robots do not execute any action in the paths they found during the experiment.

Figure 6.1 shows the amount of time used by the different types of robot control systems to generate the minimal path on various numbers of shipments. 10 different tests were performed on each system with a specific total number of shipments and the average amount of time among them was then calculated. The single robot control program and the centralized systems were run on a Sun Ultra-10 machine with a 300MHz CPU. For the distributed system, the shipment manager and each individual robot control program was run on a separate machine with the same specification. In each test, the sender and the recipient of each shipment order were randomly generated, and the distance between every two places was set to 1.

From the results, we notice that the amount of time used by the single robot control program grows exponentially as the total number of shipments increases. This is because the total number of possible interleavings of the program's actions that can be obtained during the search increases when there are more shipments to handle. Whenever the search has to increase the distance bound in the *minimizeMotion* procedure by 1 to try to handle more shipments, the amount of time that must be spent on searching within the new distance bound is much larger than the time for the search on the previous smaller distance bound. This also applies to the cases with the centralized control system since total number of shipments and also the total number of robots affect the numbers of possible interleavings of the program actions. However, the distributed control system tends to distribute the shipments to the robots, and so each robot usually has fewer shipments to handle and it does not have to consider the actions that can be done by other robots. Therefore, as shown in the figure, the distributed system was able to handle 5 shipments in less than a minute on average, while the single robot control program and centralized system take hours to generate the paths.

Total number of shipments	single robot	centralized (2 robots)	distributed (2 robots)	centralized (3 robots)	distributed (3 robots)
1	<0.1	0.3	1.8	5.2	1.9
2	0.4	6.1	4.2	68.4	4.2
3	2.0	3233.8	9.5	26229.1	6.7
4	19.4	-	14.8	-	10.7
5	423.7	-	38.1	-	23.1

Figure 6.1: Average Amount of Time Required to Generate a Path (in seconds)

Chapter 7

A Shipment Delivery Application with a Real Robot

Our shipment delivery robot has to perform many tasks. It has to construct the shortest route to serve the clients, follow the paths generated, and avoid unexpected obstacles during the execution. The architecture of our robot system is composed of three modules: a low-level reactive control module, a navigator module for detailed route planning, and a high-level robot control module which is written in IndiGolog. Each of these is used to deal with different levels of robot operation and they are all run in parallel. This chapter describes these modules and the testing that we have done on the real robot.

7.1 A 3-Layer Robot Architecture

7.1.1 The Low-Level Reactive Robot Control Module

The low-level reactive control module makes the robot move towards the destination specified by the navigation module that is described in the next section, while detecting landmarks for pose estimation and avoiding obstacles along the way. The robot we used is a Nomad *Super Scout II* [45]. It utilizes a two-wheel synchronous drive mechanical system in which both of its drive wheels remain parallel to each other at all times. It has 16 bumpers to detect contact with objects, 16 sonar sensors to determine the distance to objects, and also a video camera to obtain visual information from its environment. The perceptual data obtained by these sensors is used by the robot to acquire knowledge about the environment and avoid hitting objects in the surroundings. In addition, external devices such as keyboard and joystick, and output devices such as a speaker and monitor, can be connected to it to provide interaction with the outside world.

The motion of the robot and the sensors are controlled by the low-level reactive controller. Given a destination, this module is responsible for moving the robot to the given location. When it encounters an obstacle which blocks it from reaching its destination, its obstacle avoidance mechanism ensures that it will not hit the obstacle. This module is run on the robot's on-board computer to minimize the cost of communication between the module and the robot's hardware. This is essential since avoiding obstacle is time-critical and must be performed without delay, otherwise the robot may be damaged or it may harm the people nearby.

7.1.2 The Robot Navigation Module

In the middle layer of our robot architecture, we have the navigation module that is responsible for constructing a path between two points based on an internal map of the environment and making sure that the robot follows the path correctly. This module is based on the robot navigation program "Navigator3.2" developed for the *ARK* project [43] (see Figure 7.1). Its internal world model is constructed as an occupancy grid in which each 0.1 x 0.1 meter cell of the grid is either empty or occupied. Planning a path from one location to another location is done by calculating the potential field for each empty cell in the grid, and the path between the two given locations with the minimal sum of the potential fields of its cells is considered the most effective route. Once a path has been found, the navigator breaks it into segments and then advises the low-level robot controller to follow it segment by segment. It also keeps monitoring the execution of the low-level robot controller and sends information on the current status of the robot to the high-level control.

The path planning and navigation module has an interface for communication with the low-level and high-level control modules. The high-level module sends commands such as "go to a location" and "pick up a shipment" to this module. In the case where navigation to a new destination is requested, it first constructs a path between the robot's current position and the destination according to the information stored in the world model, and then sends the segements of the path to the low-level robot control module for execution. When an unexpected situation such as the robot being un-

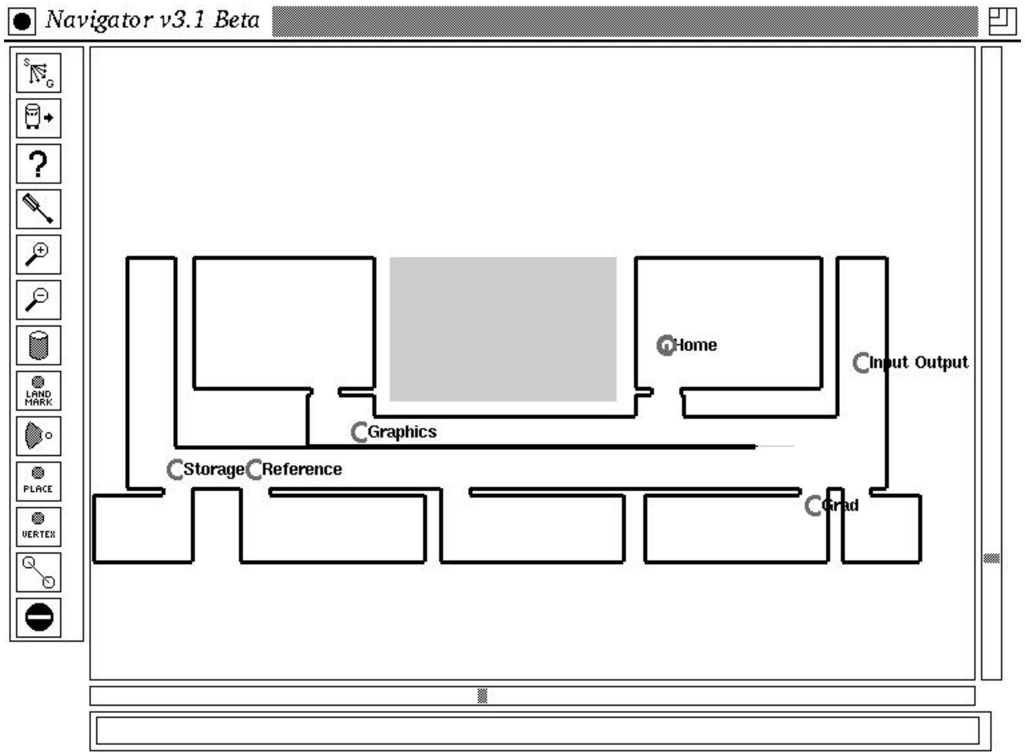


Figure 7.1: Graphical User Interface of the "Navigator", with a Map of the Robot's Work Area in the Building

able to reach the destination arises, this module receives messages from the low-level module describing the situation and sends the information to the high-level control module for an appropriate reaction. In this way, the high-level control module needs only deal with high-level tasks without paying much attention to lower-level details such as how to find a path from a place to another and how to drive the robot along the path.

7.1.3 The High-level IndiGolog Robot Control Module

The high-level robot control module consists of a knowledge base, a high-level control program and an interpreter for executing the program. It is responsible for deliberation, maintaining the knowledge base about the robot and the environment, and monitoring the high-level plan execution. As described in Chapter 3, the knowledge base and high-level robot control program are implemented in the IndiGolog language. Basically, the knowledge base contains a set of facts, such as the distance between two places and what the existing shipments are, that are known by the robot. Facts are represented as fluents and non-fluents. In addition, it contains a specification of the preconditions for each available primitive action, and also a set of rules that specify how the facts will be changed after performing a primitive action. This knowledge base is updated automatically whenever an action is performed. The high-level control program specifies the behavior of the robot. As we mentioned in Chapter 4, it has a main thread that instructs the robot how to handle the shipments. The IndiGolog interpreter takes the program and constructs a plan for the robot to serve the shipments using the shortest route. It re-plans when an unexpected event occurs during the execution which makes it impossible to follow its plan. Moreover, it keeps monitoring for exogenous actions and inserts them in the history when they occur.

The IndiGolog interpreter we used for running the robot control program is written in *Quintus Prolog* [48]. We also developed a simple interface that provides facilities for communication between the users and the interpreter. It passes the shipment orders that are made by the users from their graphical user interfaces to the interpreter and sends back the status of the shipments to the appropriate user.

7.1.4 The Individual Client Interface

In our system, the user can make shipment orders and view the status of the shipments through a simple graphical user interface, which is shown in Figure 7.2. This interface is written in Java [31], and can be run on different platforms by multiple users at the same time. Each interface displays information about the shipments that have been ordered by its user. The user can make a shipment order by clicking the button that corresponds to the recipient of the shipment.

7.2 Test Runs with the Real Robot

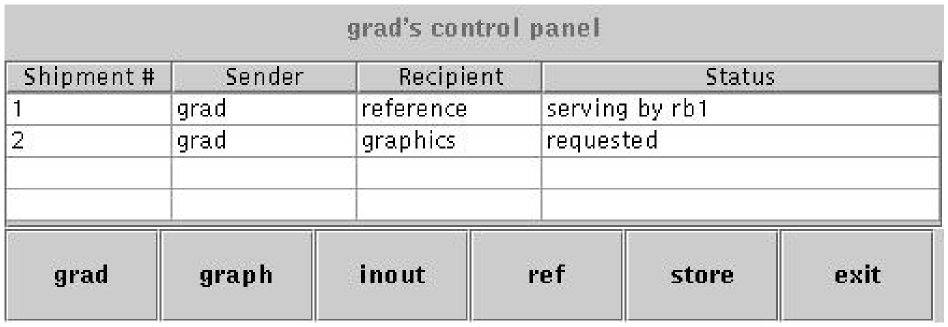
The test runs presented in this section are aimed at testing whether the robot control system is able to exhibit the expected behaviors. There are 5 clients/locations in our robot's working environment; they are grad, graphics, inout (input/output), reference, and storage. Their true layout in the robot's work area is shown in Figure 7.1. For planning routes in IndiGolog, we assume that the distances between each of these places is as listed in Figure 7.3.

7.2.1 Serving Shipments Using the Shortest Route

The purpose of this test run was to show that the robot can choose the shortest route to deliver the shipments. In this experiment, there were two shipment orders: shipment 1 from graphics to reference, and shipment 2 from inout to storage. The robot was at the home position at the beginning. The robot first constructed the shortest route for the delivery, which is home - graphics - inout - reference - storage, and then followed it. So the robot moved from home to graphics and picked up shipment 1 at there. Next, instead of going to the recipient of shipment 1, it moved to inout and picked up shipment 2. Then, it carried both shipments and moved to reference to drop off shipment 1. Finally, it moved from reference to storage and dropped off shipment 2. This is illustrated in Figures 7.4 to 7.7.

7.2.2 Replanning When a New Shipment Order Arrives

This experiment demonstrates that the robot can replan and serve the shipments using a different route when a new shipment arrives during the execution of a path. Initially, the robot was at home and there was only one shipment order: shipment 1 from inout to graphics. A shortest route to serve this shipment is home - inout - graphics. So the robot first moved to inout and picked up shipment 1, and then started to move to graphics. On



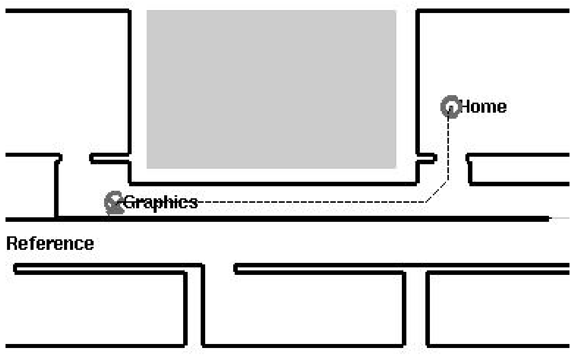
grad's control panel

Shipment #	Sender	Recipient	Status
1	grad	reference	serving by rb1
2	grad	graphics	requested

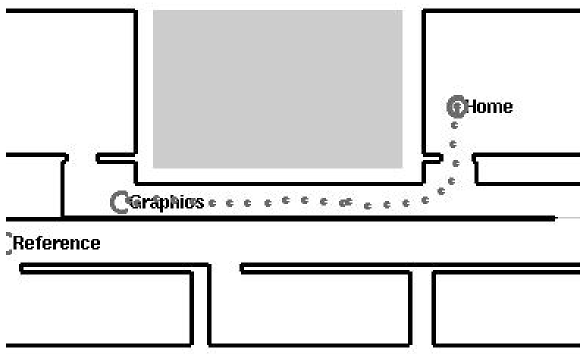
Figure 7.2: Graphical User Interface of the Client "grad"

	home	grad	graphics	inout	reference	storage
home		1	1	1	2	2
grad	1		2	1	2	2
graphics	1	2		2	3	3
inout	1	1	2		2	2
reference	2	2	3	2		1
storage	2	2	3	2	1	

Figure 7.3: Distances between Different Locations

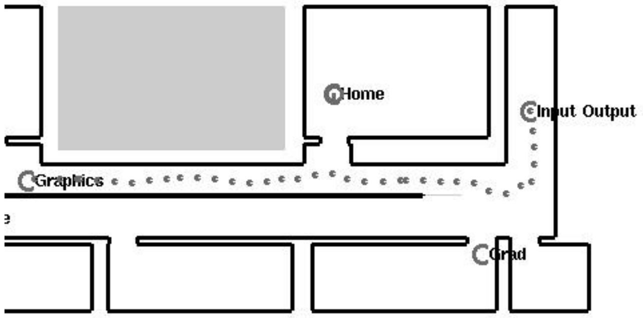


a. Route from home to graphics lab constructed by the Navigator.

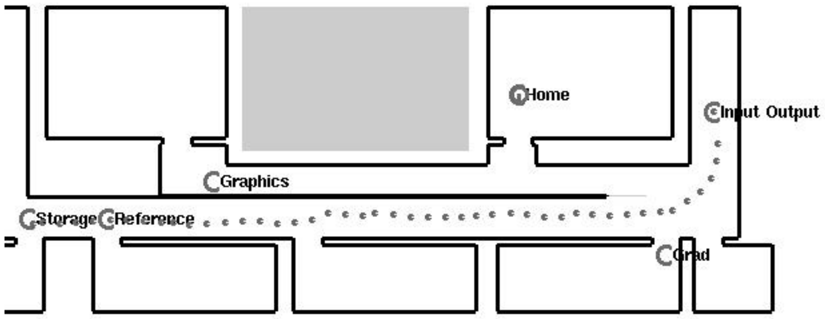


b. Actual route taken by the robot from home to graphics lab.

Figure 7.4: Test Run 1 (a,b)



c. Route taken by the robot from graphics lab to input/output lab.



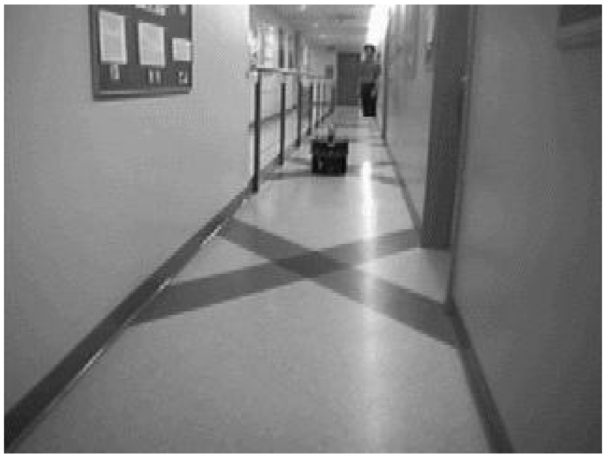
d. Route taken by the robot from input/output lab to storage room through reference room.

Figure 7.5: Test Run 1 (c,d)

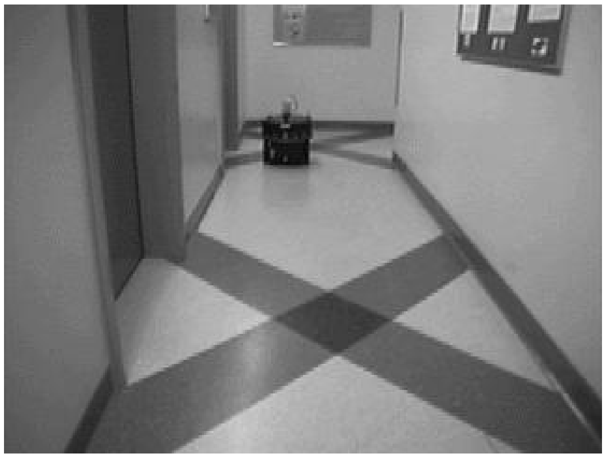
e. Robot at home position.

f. Robot starting to move to graphics lab to pick up shipment 1.

Figure 7.6: Test Run 1 (e,f)

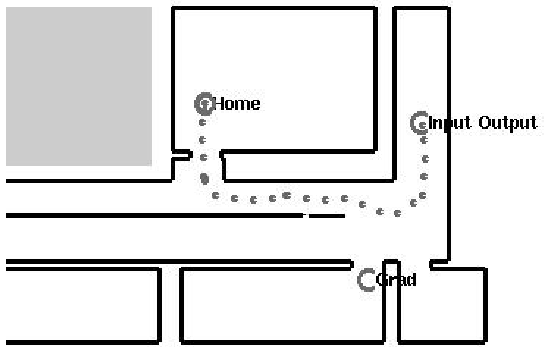


g. Robot moving to reference room to drop off shipment 1.

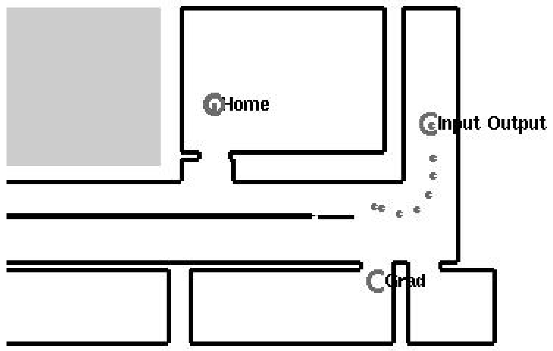


h. Robot stopping in front of the storage room.

Figure 7.7: Test Run 1 (g,h)

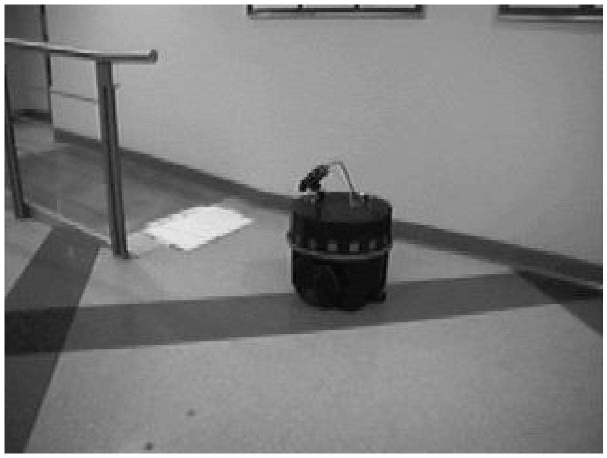


a. Route taken by the robot from home to input/output lab to pick up shipment 1.

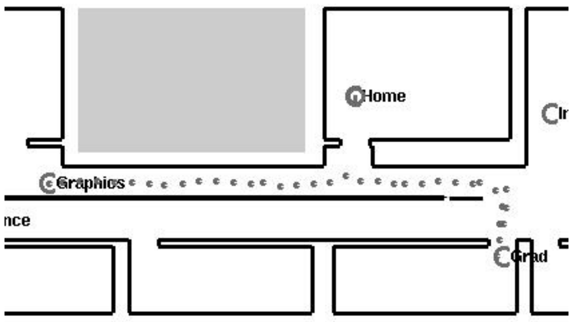


b. Route taken by the robot from input/output lab to graphics lab; the robot stops because of the arrival of a new shipment order.

Figure 7.8: Test Run 2 (a,b)



c. Robot has stopped and is planning for the new shortest route.



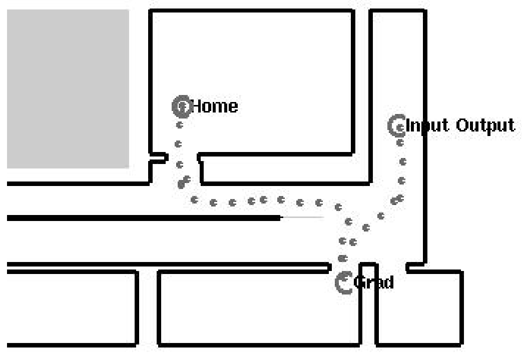
d. Route taken by the robot to grad lab to pick up shipment 2, and then to graphics lab to drop off the shipments.

Figure 7.9: Test Run 2 (c,d)

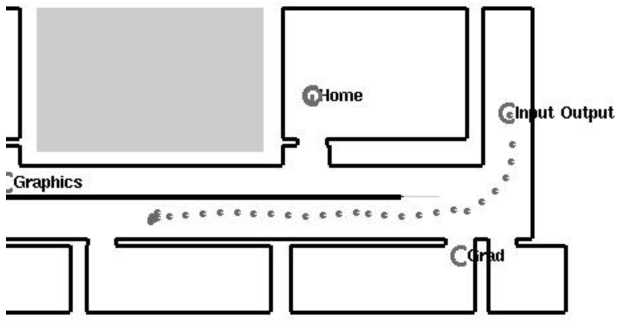
its way to graphics, grad made a new shipment order, shipment 2 from grad to graphics. In this case, although the robot could first drop off shipment 1 at graphics and then come back to grad to pick up shipment 2, this was not the best way; the shortest way was to pick up shipment 2 at grad first and then delivering both shipment to graphics. When the new order was made, the robot stopped, replanned, figured out the new shortest route, and then followed it. It moved to grad to pick up shipment 2, and carried both shipments to graphics and dropped them off. This is illustrated in Figures 7.8 and 7.9.

7.2.3 Replanning When Blocked by an Obstacle

The objective of this experiment is to show that the robot can still complete its task in the presence of a failure exogenous event. In this experiment, there were two shipment orders: shipment 1 from grad to inout, and shipment 2 from inout to storage. The robot was at the home position initially. We temporaily prevented the robot from moving from inout to storage by putting a box on the way to storage. The shortest route for serving the shipments in this experiment is home - grad - inout - storage. The robot constructed this path first and moved to grad to pick up shipment 1. Next, it moved to inout, picked up shipment 2 and dropped off shipment 1. Then, on its way to storage, it was blocked by a box for 30 seconds. Because of this, the IndiGolog controller received the "getStuck" signal and so it generated a new plan to make another attempt to go to storage. Then the box was moved away and the robot was able to move to storage and drop off shipment 2. This is illustrated in Figures 7.10 and 7.11.

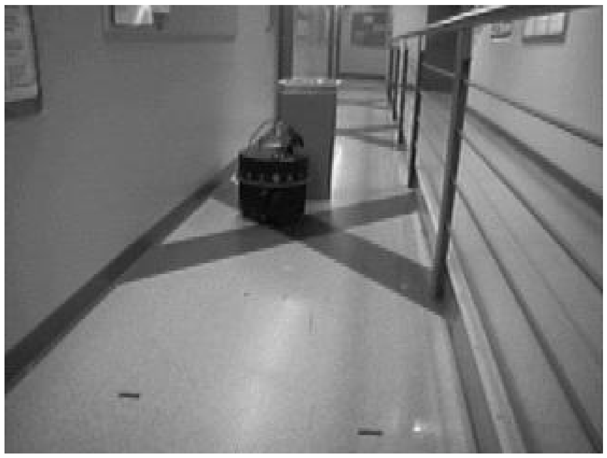


a. Route taken by the robot from home to grad to pick up shipment 1.

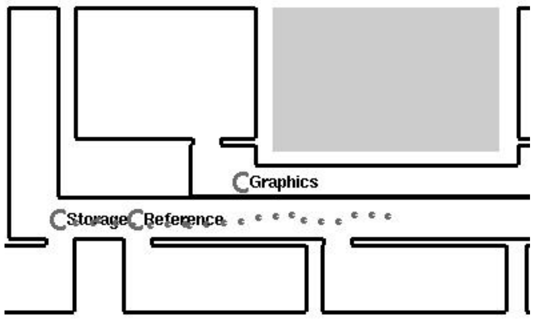


b. Route taken by the robot from grad lab to input/output room to pick up shipment 2 and drop off the shipment 1.

Figure 7.10: Test Run 3 (a,b)



c. Robot is blocked by a box on its way to the storage room.



d. Route taken by the robot to the storage room to drop off shipment 2.

Figure 7.11: Test Run 4 (c,d)

Chapter 8

Conclusion

In this thesis, we have presented an approach to implementing robot controllers for autonomous robots that are working in dynamic environments. The approach is mainly based on the IndiGolog framework. It combines planning and reactivity, so that the controller can reason about the actions that should be performed, generate plans, and react to changes in the environment quickly. We have also demonstrated that the approach can also be applied to cases where each robot has its own controller and they cooperate to perform a common task.

8.1 Contributions

We have made several enhancements to IndiGolog for building high-level robot control programs in this thesis. The main ones are as follows:

- In replanning, our IndiGolog interpreter starts searching for a new plan from the initial program and situation intead of the program and situation that are left in the search block. So, new plans can now be found in the cases such as our route optimizing delivery robot control program, where the original interpreter would fail.

- The interpreter now supports planning for programs which rely on feedback from the environment, which is simulated by a program thread.
- The interpreter now keeps the information about which actions have been performed by the thread that involves search/planning; this allows the control program to have a search/planning thread which finds an optimal way to accomplish the robot's task, and separate reactive threads for handling critical problems quickly.
- We have developed meta-level planning and execution routines to provide access to the plan found by the interpreter and give the programmer control over the execution of the plan. Planning and execution of a plan can now be interrupted when necessary and programs are now able to decide when they should search for a plan and when a plan should be executed.

We have shown that our approach can be applied to controlling multiple cooperating robots. For this, we used a simple bidding mechanism so that the given task can be distributed effectively to multiple robots. Instead of having a centralized control system that instructs all robots, each robot can have its own controller and they can cooperate by communicating with each other. Each robot can work on its own task without considering the actions that must be performed by other robots, and this reduces the amount of time and resources needed for planning. Also, the task can be accomplished even in situations where one of the robots is malfunctioning or the communication to a robot is lost.

The individual robot controller was interfaced to a real robot and architecture, and tested on real scenarios.

8.2 Future Work

There are many areas where our work can be extended. In terms of planning, our interpreter generates a sequence of program transitions for the given program and situation. In the case where replanning is required, the interpreter has to throw away the original plan and looks for a new plan from the initial program. In some cases, modifying the existing plan may be a better approach for handling the new unexpected event. Also, the system's performance might be better if it supported conditional planning, that is, generating plans with conditional branches. It could construct plans that handle many possible exogenous events in various branches of the plan, so that no replanning would be needed when one of the expected exogenous events happenned.

Our meta-level path planning routines provide more control over planning and execution, but we would also like to see if the basic object-level search meachanism can be extended to provide hooks or settable parameters so that the program can have full control on the path planning and path execution.

Although our meta-level path planning and path execution routines provide parameters for the program to specify when the execution should abort, the execution may not stop immediate when the abort condition becomes true. This is because our meta-level routines check the abort condition only between calls to $Final$ and $Trans$. Finding a transition or checking whether a configuration is final may require a large amount of time. One possible solution to this problem is to incorporate checking of the abort condition in the transition and termination checking mechanism and in the formula evaluation mechanism.

Each robot control program we have developed contains only a single thread which involves search/planning to control the motion of the robot. It might be useful to have more than one such thread in the program, if the robot has to accomplish several goals and each of its tasks is handled by a single thread. However, this may require a better mechanism for detecting when plans are compatible and avoiding unnecessary replanning. In addition, it would be interesting to try to have the robot build different plans for a task using the meta-level planning routine and then execute the best one to achieve its goal.

The measure we used to characterize the quality of a route returned by the planner was the total distance that the robot had to travel in following the route. In reality, the quality of a route relates to many other factors such as the amount of time required to complete the route and the difficulty of reaching a location. The control program should include these factors to provide a more realistic model of the real environment.

Our work on the multiple robots distributed control system is quite preliminary. The robots do not consider the actions that will be performed by the other robots at all. Two robots may try to get into the same room at the same time and they may block each other as a result. We would like the robots to communicate and adjust their plans so that this kind of problem can be avoided.

Bibliography

[1] P. Agre and D. Chapman. PENGI: An implementation of a theory of activity. In Proceedings of the Sixth National Conference on Artifical Itellgence, pages 268-272, Seattle, WA, 1987.

[2] T. Balch and R. C. Arkin. Motor schema-based formation control for multiagent robot teams. In Proceeding of the First International Conference on Multi-Agent Systems, pages 10-16, Menlo Park, California, June 1995.

[3] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A Framework for programming in temporal logic. In REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (LNCS Volume 430), pages 94-129, Springer-Verlag: Berlin, Germany, June 1989.

[4] M.E. Bratman, D.J. Israel and M.E. Pollack. Plans and resource-bounded practical reasoning. Computational Intelligence, 4:349-355, 1988.

[5] Rodney A. Brooks. Intelligence without reason. In Proceedings of the Twelfth International Joint Conference on Artifical Intelligence, pages 569-595, Sydney, Australia, 1991.

[6] Rodney A. Brooks. Intelligence without representation. Artificial Intelligence, vol. 47, pages 139-159, 1991.

[7] W. Burgard, A.B. Gremers, D. Fox, D. Haehnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In Proceedings of the 15th National Conference on Artificial Intelligence, AAAI Press, pages 31-38, July 1998.

[8] Winton H.E. Davies and Peter Edwards. Agent-K : An integration of AOP and KQML, King's College Technical Report AUCS/TR9406, 1994.

[9] R. Davis and R. Smith. Negotiation as a metaphor for distributed problem solving. Artificial Intelligence 20:63-109, 1983.

[10] Giuseppe De Giacomo and Hector J. Levesque. An incremental interpreter for high-level programs with sensing. In Logical Foundations for Cognitive Agents, eds., Hector J. Levesque and Fiora Pirri, pages 86-102. Springer-Verlag, Berlin, Germany, 1999.

[11] Giuseppe De Giacomo, Raymond Reiter and M. Soutchanski. Execution monitoring of high-level robot programs. Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98), pages 453-464, 1998.

[12] Giuseppe De Giacomo, Yves Lesperance and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence, vol. 121, pages 109-169, 2000.

[13] Giuseppe De Giacomo, Yves Lesperance and Hector J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence, pages 1221-1226, Nagoya, Japan, August, 1997.

[14] E.H. Durfee, P.G. Kenny, and K.C. Kluge. Integrated premission planning and execution for unmanned ground vehicles. In Proceedings of the First International Conference on Autonomous Agents, pages 348-354, February 1997.

[15] E.H. Durfee and T.A. Montgomery. Coordination as distributed search in a hierarchical behavior spae. IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Distributed Artificial Inelligence, SMC-21(6):1363-1378, November 1991.

[16] E. Ephrati and J.S. Rosenschein. Divide and conquer in multi-agent planning. In Proceedings of the Twelfth National Conference on Artifical Intelligence, pages 375-380, July 1994.

[17] Oren Etzioni and Daniel Weld. A softbot-based interface to the internet. Comm. of ACM, vol. 37, pages 48-53, July 1994.

[18] R.J. Firby. An investigation into reactive planning in complex domains. In Proceedings of the Sixth National Conference on Artificial Intelligence, pages 202-206, 1987.

[19] Michael Fisher. A survey of concurrent METATEM - the language and its applications. In Proceedings of First International Conference on Temporal Logic (ICTL). Bonn, Germany. Published by Springer-Verlag as Lecture Notes in Computer Science vol. 827, pages 480-505, July 1994.

[20] Michael Fisher. Concurrent METATEM - a language for modeling reactive systems. In Proceedings of Parallel Architectures and Languages Europe (PARLE). Published by Springer-Verlag as Lecture Notes in Computer Science vol. 694. June 1993.

[21] L. Gasser, C. Braganza and N. Herman. Implementing distibuted artifical intelligence systems using MACE. In Proceedings of the Third IEEE Conference on Artifical intelligence Applications, pages 315-320, 1988.

[22] Erann Gat. 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, San Jose, CA, pages 809-815, 1992.

[23] M.R. Genesereth and N. Nilsson. Logical foundations of artificial intelligence. Morgan Kaufmann Publisers: San Mateo, CA, 1987.

[24] Michael Georgeff. Communication and interaction in multi-agent planning. In Proceedings of the Third National Conference on Artificial Intelligence, pages 125-129, July 1983.

[25] M.P. Georgeff and F.F. Ingrand. Decision-making in an embedded reasoning system. In Proceedings of the Eleventh International Joint Conference on Artifical Intelligence, Detroit, Michigan, pages 972-978, 1989.

[26] H. Grosskreutz and G. Lakemeyer, cc-Golog: Towards More Realistic Logic-Based Robot Controllers, 8th Intl. Workshop on Non-Monotonic Reasoning, special session on representing actions and planning, pages 476-482, 2000.

[27] Dirk Hahnel, Wolfram Burgard and Gerhard Lakemeyer. Golex - bridging the gap between logic (GOLOG) and a real robot. In Proceedings of the German Annual Conference on Articial Intelligence (KI), vol. 1504 of LNAI, pages 165-176, Bremen, Germany, September 1998.

[28] K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J-J. Ch. Meyer. A Formal Embedding of AgentSpeak(L) in 3APL. In G. Antoniou and J. Slaney, editors, Advanced Topics in Artificial Intelligence (LNAI 1502), pages 155–166. Springer-Verlag, 1998.

[29] K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J-J. Ch. Meyer. Formal semantics for an abstract agent programming language. In intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages, volume 1365, pages 215-229. Springer-Verlag, 1998.

[30] Koen Hindriks, Yves Lesperance, and Hector J. Levesque. An Embedding of ConGolog in 3APL. Technical Report UU-CS-2000-13, Department of Computer Science, University Utrecht, 2000.

[31] Java^{TM} 2 SDK v1.2.2. Available at http://java.sun.com.

[32] M. Jenkin, E. Milios, P. Jasiobedzki, N. Bains, and K. Tran. Global navigation for ARK. In Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems, page 2165-2171, 1993.

[33] N.R. Jennings. Specification and implementation of a belief desire joint-intention architecture for collaborative problem solving. Journal of Intelligent and Cooperative Information Systems, 2(3):289-318, 1993.

[34] Yves Lesperance, Hector J. Levesque, F. Lin, D. Marcu, Raymond Reiter, and R. Scherl. A Logical Approach to High-Level Robot Programming – A Progress Report. In Benjamin Kuipers, editor,Control of the Physical World by Intelligent Systems, Papers from the 1994 AAAI Fall Symposium, pages 79-85, New Orleans, LA, November, 1994.

[35] Yves Lesperance, Kenneth Tam and Michael Jenkin. Reactivity in a Logic-Based Robot Programming Framework. Cognitive Robotics - Papers from the 1998 AAAI Fall Symposium, Technical Report FS-98-02, AAAI Press, pages 98-105, Orlando, FL, October, 1998.

[36] H.J. Levesque and M. Pagnucco. Legolog: Inexpensive Experiments in Cognitive Robotics In Proceedings of the Second International Cognitive Robotics Workshop, Berlin, Germany, August 21-22, 2000.

[37] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming, 31(59-84), 1997.

[38] P. Maes. The agent network architecture (ANA). SIGART Bulletin, 2(4):115-120, 1991.

[39] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artifical intelligence. In Machine Intelligence, volume 4, 463-502, Edinburgh University Press, Edinburgh, United Kingdom, 1979.

[40] T.M. Mitchell. Becoming increasingly reactive. In Proceedings of the Eighth National Conference on Artifical Intelligence, Boston, Massachusetts, pages 1051-1058, 1990.

[41] J.P. Muller. A conceptual model for agent interacion. In Proceedings of the Second International Working Conference on Cooperating Knowledge Based Systems, pages 213-234, DAKE Centre, University of Keele, United Kingdom, 1994.

[42] J.P. Muller, M. Pischel and M. Thiel. Modelling reactive behaviour in vertically layered agent architectures. Intelligent Agents: Theories, Architectures, and Languages, volume 890, pages 261-276, Springer-Verlag, Heidelberg, Germany, 1995.

[43] S.B. Nickerson, P. Jasiobedzk, D. Wilkes, Michael Jenkin, Evangelos Milios, John Tsotos, A. Jepson and O. N. Bains. The ARK project:

autonomous mobile robots for known industrial environments. Robotics and Autonomous Systems 25, pages 83-104, 1998.

[44] Nils J. Nilsson. Shakey the robot. Technical Note 323, SRI Artifical Intelligence Center, April, 1984.

[45] Nomadic Technologies, inc. http://www.robots.com.

[46] T. Ohko, K. Hiraki and Y. Anzai. Lemming: a learning system for multi-robot environments. The 1993 IEEE/RSJ International Confernce on Inteligent Robots and Systems, pages 1141-1146, 1993.

[47] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[48] Quintus Prolog Release 3. SICS. Available at http://www.sics.se/isl/quintus.

[49] Anand S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In MAAMAW'96, LNAI 1038, pages 42-55, The Netherlands, 1996.

[50] Raymond Reiter. Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems, to appear, MIT Press, 2001; also available at http://www.cs.toronto.edu/cogrobo.

[51] Raymond Reiter. Sequential, temporal GOLOG. Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98). 547-556.

[52] M. Robinson and M. Jenkin. Reactive low level control of the ARK. In Proceedings, Vision Interface '94, page 41-47, 1994.

[53] S.J. Rosenschein. Formal theories of knowledge in AI and robotics. New Generation Computing, pages 345-357, 1985.

[54] S.J. Rosenschein and L.P. Kaelbling. The synthesis of digital machines with provale epistemic properties. In Proceedinds of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge, pages 83-98. Morgan Kaufmann Publishers: San Mateo, CA, 1986.

[55] M. Rozier, et al. Chorus distributed operating systems. Computing Systems, 1(4), 1988, page 305-367.

[56] Yoav Shoham. Agent-oriented programming. Artifical Intelligence, 60(1): 51-92, 1993.

[57] Yoav Shoham. Agent oriented programming: an overview of the framework and a summary of recent research, Knowledge Representation and Reasoning Under Uncertainty, pp. 123-9, 1994.

[58] L.M. Stephens. The effect of agent control strategy on the performance of a DAI pursuit problem. In Proceedings of the 10th International Workshop on Distributed Artificial Intelligence, Bandera, Texas, October, 1990

[59] Herbert Simon. The Sciences of the Artificial, 2nd edition. The MIT Press, MA, 1981.

[60] R. Smith. The contract net protocol: high-level communication and control in a distributed problem solver. IEEE Transactions on Computers, volume C-29, no. 12, pages 1104-1113, December 1980.

[61] Kenneth Tam, J. Lloyd, Yves Lesperance, Hector J. Levesque, Fangzhen Lin, D. Marcu, Raymond Reiter and Michael Jenkin. Controlling autonomous robots with GOLOG. In Proceedings of the Tenth Australian Joint Conference on Artifical Intelligence, pages 1-12, Perth, Australia, November, 1997.

[62] S. Rebecca Thomas. The PLACA agent programming language, intelligent agents. ECAI-94 Workshop on Agent Theories, Architectures, and Languages Proceedings pp. 355-70, 1995.

[63] Gerhard Weiβ. Distributed reinforcement learning. Robotics and Autonomous Systems, 15:135-142.

[64] D.E. Wilkins and K.L. Myers. A common knowledge representation for plan generation and reactiv execution. Journal of Logic and Computation, 5(6):731-761, 1995.

Appendix A

Simple Robot Control Program for Shipment Delivery

The example presented in here is a robot control program for doing shipment delivery. It is adapted from the work done by Lesperance, Tam and Jenkin [61]. The task of the robot is to pick up all ordered shipments and deliver them to the clients.

Ordinary primitive actions:

$goTo(Place)$	the robot moves to $Place$
$pickUp(SNo)$	robot picks up shipment SNo
$dropOff(SNo)$	robot drops off shipment SNo

Exogenous primitive actions:

$reachDest$	robot has reached its destination
$getStuck$	robot is unable to reach destination

Primitive fluents:

$robotPos$	position of the robot
$shipmentPos(SNo)$	position of shipment SNo
$shipmentSender(SNo)$	sender of shipment SNo
$shipmentRecipient(SNo)$	recipient of shipment SNo

Precondition axioms:

$$Poss(goTo(Place), s) \equiv True$$
$$Poss(pickUp(n), s) \equiv shipmentPos(n, s) = robotPos(s)$$
$$Poss(dropOff(n), s) \equiv shipmentPos(n, s) = onBoard$$

Successor state axioms:

$$\begin{aligned}
&robotPos(do(a, s)) = p \equiv \\
&\qquad a = goTo(p) \vee (\forall\, p')a \neq goTo(p') \wedge robotPos(s) = p \\
&shipmentPos(n, do(a, s)) = p \equiv \\
&\qquad a = pickup(n) \wedge p = onBoard \vee \\
&\qquad a = dropOff(n) \wedge p = robotPos(s) \vee \\
&\qquad a \neq pickUp(n) \wedge a \neq dropOff(n) \wedge shipmentPos(n, s) = p \\
&shipmentSender(n, do(a, s)) = c \equiv shipmentSender(n, s) = c \\
&shipmentRecipient(n, do(a, s)) = c \equiv shipmentRecipient(n, s) = c
\end{aligned}$$

Defined fluents:

```
$proc\ clientToServe(c)$
    $(\exists n1) shipmentPos(n1) = c \wedge shipmentTo(n1) \neq c)$
    $\vee$
    $(\exists n2) shipmentPos(n2) = onBoard \wedge shipmentTo(n2) = c)$
$end$
```

Defined procedures:

```
$proc\ pickAll(c)$
    $while\ ((\exists n1) shipmentPos(n1) = c \wedge client(c) \wedge$
                $shipmentRecipient(n1) \neq c)$
        $(\pi n2)$ [ $?(shipmentPos(n2) = c \wedge client(c) \wedge$
                    $shipmentRecipient(n2) \neq c)$ ;
                  $pickUp(n2)$
                ]
$end$
```

```
$proc\ dropAll(c)$
    $while\ ((\exists n1) shipmentPos(n1) = onBoard \wedge$
            $shipmentRecipient(n1) = c)$
        $(\pi n2)$ [ $?(shipmentPos(n2) = onBoard \wedge$
                $shipmentRecipient(n2) = c)$ ;
              $dropOff(n2)$
            ]
$end$
```

```
$proc\ handleRequests(Max)$
    $?(\neg((\exists c)\ clientToServe(c)))$
    |
    $(\pi c, p, m)$[ $?(clientToServe(c) \wedge robotPos = p \wedge$
                $m = Max - distance(p, c) \wedge m \geq 0)$ ;
             $goTo(c)$ ;
             $pickAll(c)$ ;
             $dropAll(c)$ ;
             $handleRequests(m)$ ]
$end$
```

```
$proc\ minimizeMotion(Max)$
    $handleRequests(Max)$
    |
    $minimizeMotion(Max + 1)$
$end$
```

```
$proc\ control$
    $search(\ minimizeMotion(0)\ )$
$end$
```

Appendix B

Complete Program of the Shipment Manager for Distributed Shipment Delivery

Ordinary primitive actions:

$callForBid(Manager, Robot, SNo, Sender, Recipient)$
$award(Manager, Robot, SNo, Sender, Recipient)$
$rejectShipment(Manager, Sender, SNo)$
$tick$

Exogenous primitive actions:

$shipmentRequest(Sender, Recipient)$
$bid(Robot, SNo, Value)$
$report(Robot, Manager, SNo, Message)$

Primitive fluents:

$shipmentState(SNo)$
$shipmentSender(SNo)$
$shipmentRecipient(SNo)$
$totalJobsServing(Robot)$
$callAtTime(Robot, SNo)$
$hasBid(Robot, SNo)$
$counter$

Precondition axioms:

$$
\begin{aligned}
&Poss(callForBid(manager, r, n, c1, c2), s) \equiv \\
&\qquad shipmentSender(n, s) = c1 \wedge \\
&\qquad shipmentRecipient(n, s) = c2 \\
&Poss(award(manager, r, n, c1, c2), s) \equiv \\
&\qquad shipmentState(n, s) = askedAllRobots \wedge \\
&\qquad hasBid(r, n, s) \neq -1 \wedge \\
&\qquad shipmentSender(n, s) = c1 \wedge \\
&\qquad shipmentRecipient(n, s) = c2 \\
&Poss(rejectShipment(manager, c, n), s) \equiv \\
&\qquad shipmentSender(n, s) = c \wedge \\
&\qquad shipmentState(n, s) \neq nonExistent \\
&Poss(tick, s) \equiv True
\end{aligned}
$$

Successor state axioms:

$$
\begin{aligned}
&shipmentState(n, do(a, s)) = x \equiv \\
&\qquad ((\exists c1, c2)a = shipmentRequest(c1, c2) \wedge counter = n \wedge \\
&\qquad\;\; x = justIn) \vee \\
&\qquad ((\exists r, c1, c2)a = callForBid(manager, r, n, c1, c2) \wedge \\
&\qquad\;\; shipmentState(n, s) = justIn \wedge \\
&\qquad\;\; \neg((\exists r')r' \neq r \wedge callAtTime(r', n) \neq currentTime) \wedge \\
&\qquad\;\; x = askedAllRobots) \vee \\
&\qquad ((\exists r, c1, c2)a = award(manager, r, n, c1, c2) \wedge x = assigned) \vee \\
&\qquad ((\exists c1)a = rejectShipment(manager, n, c1) \wedge x = rejected) \vee \\
&\qquad ((\exists r)a = report(r, manager, n, completed) \wedge x = delivered) \vee \\
&\qquad ((\forall r, c1, c2)a \neq shipmentRequest(c1, c2) \wedge \\
&\qquad\;\; a \neq callForBid(manager, r, n, c1, c2) \wedge \\
&\qquad\;\; a \neq award(manager, r, n, c1, c2) \wedge
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad a \neq rejectShipment(manager, n, c1)\wedge \\
\qquad\qquad a \neq report(r, manager, n, completed) \wedge x = shipmentState(n, s)) \\
\quad shipmentSender(n, do(a, s)) = c \equiv shipmentSender(n, s) = c \\
\quad shipmentRecipient(n, do(a, s)) = c \equiv shipmentRecipient(n, s) = c \\
\quad totalJobsServing(r, do(a, s)) = t \equiv \\
\qquad\quad ((\exists n, c1, c2)a = award(r, n, c1, c2)\wedge \\
\qquad\qquad t = totalJobsServing(r, s) + 1)\vee \\
\qquad\quad ((\exists n)a = report(r, n, completed)\wedge \\
\qquad\qquad t = totalJobsServing(r, s) - 1)\vee \\
\qquad\quad ((\forall n, c1, c2)a \neq award(r, n, c1, c2)\wedge \\
\qquad\qquad a = report(r, n, completed) \wedge totalJobsServing(r, s) = t) \\
\quad callAtTime(r, n, do(a, s)) = t \equiv \\
\qquad\quad ((\exists c1, c2)a = callForBid(r, n, c1, c2) \wedge t = currentTime)\vee \\
\qquad\quad ((\forall c1, c2)a \neq callForBid(r, n, c1, c2)\wedge \\
\qquad\qquad callAtTime(r, n, s) = t) \\
\quad hasBid(r, n, do(a, s)) = v \equiv \\
\qquad\quad a = bid(r, n, v)\vee \\
\qquad\quad ((\forall v')a \neq bid(r, n, v') \wedge hasBid(r, n, s) = v) \\
\quad counter(do(a, s)) = v \equiv \\
\qquad\quad ((\exists c1, c2)a = shipmentRequest(c1, c2) \wedge v = counter(s) + 1)\vee \\
\qquad\quad ((\forall c1, c2)a \neq shipmentRequest(c1, c2) \wedge counter(s) = v) \\
\quad totalJobsServing(r, do(a, s)) = t \equiv \\
\qquad\quad ((\exists n, c1, c2)a = award(manager, r, n, c1, c2)\wedge \\
\qquad\qquad t = totalJobsServing(r, s) + 1)) \vee \\
\qquad\quad ((\exists n)a = report(r, manager, n, completed)\wedge \\
\qquad\qquad t = totalJobsServing(r, s) - 1) \vee \\
\qquad\quad ((\forall n, c1, c2)a \neq award(manager, r, n, c1, c2)\wedge \\
\qquad\qquad a \neq report(r, manager, n, completed)\wedge \\
\qquad\qquad t = totalJobsServing(Robot, s)
\end{array}
$$

Defined fluents:

```
proc allHaveBidded(N)
    $\neg((\exists r)hasBid(r, N) = -1)$        // no bid = −1
end
```

```
$proc\ timeOut(N)$
    $(\exists r) callAtTime(r, N) \neq -1 \wedge$
        $currentTime - callAtTime(r, N) > 3$
$end$
```

Defined procedures:

```
$proc\ handleNewRequest(N)$
    $if\ ((\exists r) connectedToManager(r) = yes)\ then$
        $while(shipmentState(N) \neq askedAllRobots)$
            $(\pi\ r)\ [\ ?(callAtTime(r, N) \neq -1);$
                        $callForBid(r, N, shipmentSender(N),$
                                $shipmentRecipient(N))$
                    $]$
    $else$
        $/ * reject\ if\ no\ robot\ is\ available * /$
        $rejectShipment(N, shipmentSender(N))$
$end$

$proc\ decideContractor(N)$
    $if\ ((\exists r, v)(v = hasBid(r, N) \wedge v > -1)\ then$

        $/ * if\ someone\ has\ bidded,\ find\ contractor * /$
        $(\pi\ r)$
            $[\ ?(bestRobot(r, N))\ ;$
              $award(manager, r, N, shipmentSender(N),$
                    $shipmentRecipient(N))$
            $]$

    $else$
        $/ * if\ no\ one\ has\ bidded,\ reject\ shipment * /$
        $rejectShipment(manager, shipmentSender(N), N)$
$end$

$proc\ bestRobot(R, N)$
    $(\exists v1, j1)$
```

```
        $(v1 = hasBid(R, N) \wedge v1 \neq -1 \wedge$
         $j1 = totalJobsServing(R) \wedge$
         $\neg((\exists r2, v2, j2) v2 = hasBid(r2, N) \wedge v2 \neq -1 \wedge$
             $j2 = totalJobsServing(r2) \wedge$
             $(v2 < v1 \vee (v2 = v1 \wedge j2 < j1)))$
$end$

$proc\ manager$
    $< (\exists n1) shipmentState(n1) = justIn \rightarrow$
        $handleNewRequest(n1) >$
    $\gg$
    $< (\exists n2)(shipmentState(n2) = askedAllRobots \wedge$
          $allHaveBidded(n2)) \vee$
          $(shipmentState(n2) = askedAllRobots \wedge timeOut(n2)) \rightarrow$
        $decideContractor(n2) >$
    $\gg$
    $< True \rightarrow tick >$        $//\ keep\ running\ forever$
$end$
```

Appendix C

Complete Control Program of Individual Robot for Distributed Shipment Delivery

Ordinary primitive actions:

$startGoTo(Robot, Place)$
$pickUp(Robot, SNo)$
$dropOff(Robot, SNo)$
$abortGoTo(Robot)$
$acknowledge(Robot, SNo)$
$bid(Robot, SNo, Value)$
$stopBidding(Robot, SNo)$
$report(Robot, SNo, Message)$
$setFinalPosition(Robot, Place)$

Exogenous primitive actions:

$reachDest(Robot)$
$getStuck(Robot)$
$callForBid(Robot, SNo, Sender, Recipient)$
$award(Robot, SNo, Sender, Recipient)$

Primitive fluents:

$robotPos(Robot)$
$robotState(Robot)$
$robotDest(Robot)$
$shipmentState(SNo)$
$shipmentSender(SNo)$
$shipmentRecipient(SNo)$
$finalRobotPos(Robot)$

Precondition axioms:

$$\begin{aligned}
&Poss(startGoTo(r,p),s) \equiv \\
&\quad robotState(r,s) = idle \vee robotState(r,s) = reached \\
&Poss(reachDest(r),s) \equiv robotState(r,s) = moving \\
&Poss(getStuck(r),s) \equiv robotState(r,s) = moving \\
&Poss(abortGoTo(r),s) \equiv robotState(r,s) = moving \\
&Poss(pickUp(r,n),s) \equiv shipmentState(n,s) = robotPos(r,s) \\
&Poss(dropOff(r,SNo),s) \equiv shipmentState(n,s) = onBoard \\
&Poss(bid(r,n,v),s) \equiv \\
&\quad shipmentState(n,s) = requested \vee \\
&\quad (\exists t) shipmentState(n) = replied(t) \\
&Poss(stopBidding(r,n),s) \equiv \\
&\quad shipmentState(n) = requested \vee (\exists t) shipmentState(n) = replied(t) \\
&Poss(report(r,n,m),s) \equiv True \\
&Poss(setFinalPosition(r,p),s) \equiv True \\
&Poss(callForBid(r,n,c1,c2),s) \equiv True \\
&Poss(award(r,n,c1,c2),s) \equiv \\
&\quad shipmentState(n,s)) = requested \vee \\
&\quad (\exists t) shipmentState(n,s) = replied(t) \\
&Poss(acknowledge(r,n),s) \equiv shipmentState(n,s) = justAwarded
\end{aligned}$$

Successor state axioms:

$$\begin{aligned}
&robotPos(r,do(a,s)) = p \equiv \\
&\quad a = startGoTo(r,p) \wedge p = unknown \vee \\
&\quad a = reachDest(r) \wedge p = robotDest(r) \vee \\
&\quad (\forall\, p')a \neq startGoTo(r,p') \wedge a \neq reachDest(r) \wedge robotPos(r,s) = p \\
&robotState(r,do(a,s)) = x \equiv \\
&\quad (\exists\, p)a = startGoTo(r,p) \wedge x = moving \vee
\end{aligned}$$

$$
\begin{array}{l}
\quad a = reachDest(r) \wedge x = reached \vee \\
\quad a = getStuck(r) \wedge x = idle \vee \\
\quad a = abortGoTo(r) \wedge x = idle \vee \\
\quad (\forall\, p')a \neq startGoTo(r,p') \wedge a \neq reachDest(r) \wedge a \neq getStuck(r) \wedge \\
\quad\ \ a \neq abortGoTo(r) \wedge robotState(r,s) = x \\
robotDest(r, do(a,s)) = p \equiv \\
\quad a = startGoTo(r,p) \vee \\
\quad (\forall\, p')a \neq startGoTo(r,p') \wedge robotDest(r,s) = p \\
shipmentSender(n, do(a,s)) = c \equiv \\
\quad (\exists\, c')a = callForBid(robot,n,c,c') \vee \\
\quad (\forall\, c')a \neq callForBid(robot,n,c,c') \wedge shipmentSender(n,s) = c \\
shipmentRecipient(n, do(a,s)) = c \equiv \\
\quad (\exists\, c')a = callForBid(robot,n,c',c) \vee \\
\quad (\forall\, c')a \neq callForBid(robot,n,c',c) \wedge shipmentRecipient(n,s) = c \\
shipmentState(n, do(a,s)) = p \equiv \\
\quad ((\exists c1,c2)a = callForBid(robot,n,c1,c2) \wedge p = requested) \vee \\
\quad ((\exists v)a = bid(robot,n,v) \wedge \\
\quad\ \ (shipmentState(n,s) = requested \wedge p = replied(1)) \vee \\
\quad\ \ ((\exists t)shipmentState(n,s) = replied(t) \wedge p = replied(t+1))) \vee \\
\quad (a = stopBidding(robot,n) \wedge p = replied(2)) \vee \\
\quad (a = acknowledge(robot,n) \wedge p = justAwarded) \vee \\
\quad (a = pickUp(robot,n) \wedge p = onBoard) \vee \\
\quad (a = dropOff(robot,n) \wedge p = delivered) \vee \\
\quad ((\forall c1,c2,v)a \neq callForBid(robot,n,c1,c2) \wedge a \neq bid(robot,n,v) \wedge \\
\quad\ \ a \neq stopBidding(robot,n) \wedge a \neq acknowledge(robot,n) \wedge \\
\quad\ \ a \neq pickUp(robot,n) \wedge a \neq dropOff(robot,n) \wedge \\
\quad\ \ shipmentState(n,s) = p) \\
finalRobotPos(r, do(a,s)) = p \equiv \\
\quad a = setFinalPosition(r,p) \vee \\
\quad (\forall p')a \neq setFinalPosition(r,p') \wedge finalRobotPos(r,s) = p
\end{array}
$$

Defined fluents:

```
proc $clientToServe(c)$
    $(\exists n)shipmentState(n) = c \wedge client(c) \wedge shipmentRecipient(n) \neq c)$
    $\vee$
    $(\exists n)shipmentState(n) = onBoard \wedge shipmentRecipient(n) = c)$
end
```

Defined procedures:

```
$proc\ handleRequests(R, Max)$
    $?(\neg((\exists c) clientToServe(c)))$
    $|$
    $(\pi\ c, p, m)$ [ $?(clientToServe(c) \wedge robotPos(R) = p \wedge$
                          $m = Max - distance(p, c) \wedge m \geq 0);$
                     $startGoTo(R, c);$
                     $?(robotState(R) \neq moving);$
                     $if\ (robotState(R) = reached)\ then$
                         [ $pickAll(R, c);$
                           $dropAll(R, c)$ ];
                     $handleRequests(R, m)$ ]
$end$

$proc\ envSimulator(R)$
    $<\ robotState(R) = moving \rightarrow sim(reachDest(R))\ >$
$end$

$proc\ minimizeMotion(R, Max)$
    $handleRequests(R, Max)$
    $|$
    $minimizeMotion(R, Max + 1)$
$end$

$proc\ handleNewOrder(R, N)$
    [ $if(robotState = moving,$
          $abortGoTo(R),$
      $else,$
          $no\_op$
      );
      $acknowledge(R, N)$
    ]
$end$

$proc\ quickBid(R, N)$
    $(\pi\ p1, c2, c3, d3)$
```

```
        [  $?(finalRobotPos = p1 \wedge$
              $shipmentSender(N) = c2 \wedge shipmentRecipient(N) = c3 \wedge$
              $d3 = distance(p1, c2) + distance(c2, c3))$;
           $bid(R, N, d3)$
        ]
$end$

$proc\ slowBid(R, N)$
    $(\pi\ ch, pos, t, d1, d2)$
        [  $?(ch = now \wedge pos = robotPos \wedge current\_time(t))$;

           $/ * total\ number\ of\ steps\ to\ serve\ the\ existing\ shipments * /$
           $pi(p1)\ ?(findpath([],$
                        [ $abortGoTo(R)$;
                          $minimizeMotion(R, 0) || envSimulator(R)$ ],
                        $ch$,
                        $bid\_is\_invalid \vee timeOut(t))$,
                        $p1) \wedge$
                 $pathLength(p1, pos, d1))$

           $/ * total\ number\ of\ steps\ to\ serve\ the\ existing + new\ shipments * /$
           $pi(p2)\ ?(findpath([],$
                        [ $pi(c1, c2)$
                            [ $?(shipmentSender(N) = c1 \wedge$
                                  $shipmentRecipient(N) = c2)$;
                              $sim(award(R, N, c1, c2))$ ];
                          $ackShipment(R, N)$;
                          $abortGoTo(R)$;
                          $minimizeMotion(R, d1) || envSimulator(R)$ ],
                        $ch$,
                        $bid\_is\_invalid \vee timeOut(t)$,
                        $p2) \wedge$
                 $pathLength(p2, pos, d2))$

           $/ * calculate\ the\ bid\ value * /$
           $if(bid\_is\_invalid \vee timeOut(t) \vee d1 = -1 \vee d2 = -1)$
              $stopBidding(R, N)$
           $else$
```

```
            $bid(R, N, d2 - d1)$
        ]
$end$

$proc\ followRoute(path)$
    $(\pi\ rm1)$
        [ $msearch(path, pathFound(mypath), unblockSimCond, rm1)$,
          $if(rm1 = [],$
            $no\_op$,     $//\ succeeded$
          $else,$
            [ $/ * save\ the\ final\ position\ of\ the\ robot * /$
              $(\pi\ fpos, p, d, h, i)$
                [ $?(rpath(p, d, h, i) = rm1 \wedge$
                      $findFinalPosition(p1, robotPos, fpos))$,
                  $setFinalPosition(fpos)$
                ]

              $/ * execute\ the\ new\ path * /$
              $followRoute(rm1)$
            ]
        ]
$end$

$proc\ motionControl(R)$
    $(\pi\ path)$
        [ $msearch(minimizeMotion(R, 0)||envSimulator(R)$,
                  $pathFound(mypath)$,
                  $unblockSimCond$,
                  $path)$,

          $/ * save\ the\ final\ position\ of\ the\ robot * /$
          $(\pi\ fpos, p, d, h, i)$
            [ $?(rpath(p, d, h, i) = path \wedge$
                  $findFinalPosition(p1, robotPos, fpos)$,
              $setFinalPosition(fpos)$
            ]

          $/ * execute\ the\ path * /$
```

```
            $followRoute(path)$
        $]$
$end$

$proc\ control(R)$
    $< (\exists n1) shipmentState(n1) = justAwarded \rightarrow$
        $handleNewOrder(R, n1) >$
    $\gg$
    $< (\exists n2) shipmentState(n2) = requested \rightarrow$
        $quickBid(R, n2) >$
    $\gg$
    $< (\exists c) clientToReach(c) \rightarrow motionControl(R) >$
    $\gg$
    $< (\exists n3) shipmentState(n3) = replied(1) \wedge$
         $(robotState = moving \vee \neg((\exists c3) clientToReach(c3))) \rightarrow$
        $slowBid(R, n3) >$
    $\gg$
    $< True \rightarrow no\_op >$      // keep running
$end$
```

Appendix D

Centralized Control Program for Multi-Robot Shipment Delivery

This program is a modified version of the program in Appendix A and is used to control multiple robots to do shipment delivery. Here, we show a version for 3 robots.

Ordinary primitive actions:

$startGoTo(robot, place)$	*robot* starts moving to *place*
$pickUp(robot, SNo)$	*robot* picks up shipment *SNo*
$dropOff(robot, SNo)$	*robot* drops off shipment *SNo*
$abortGoTo(robot)$	*robot* stops moving to its destination

Exogenous primitive actions:

$reachDest(robot)$	robot has reached its destination
$getStuck(robot)$	robot unable to reach destination
$orderShipment(SNo, Sndr, Rcpnt)$	client *Sndr* wants to send shipment *SNo* to *Rcpnt*

Primitive fluents:

$robotPos(robot)$	position of $robot$
$robotState(robot)$	state of $robot$
$robotDest(robot)$	destination of $robot$
$shipmentPos(SNo)$	position of shipment SNo
$shipmentSender(SNo)$	sender of shipment SNo
$shipmentRecipient(SNo)$	recipient of shipment SNo

Precondition axioms:

$$\begin{aligned}
&Poss(startGoTo(r, place), s) \equiv \\
&\quad robotState(r, s) = idle \lor robotState(r, s) = reached \\
&Poss(reachDest(r), s) \equiv robotState(r, s) = moving \\
&Poss(getStuck(r), s) \equiv robotState(r, s) = moving \\
&Poss(abortGoTo(r), s) \equiv robotState(r, s) = moving \\
&Poss(pickUp(r, n), s) \equiv shipmentPos(n, s) = robotPos(s) \\
&Poss(dropOff(r, n), s) \equiv shipmentPos(n, s) = r \\
&Poss(orderShipment(n, Sender, Recipient), s) \equiv true \\
&Poss(acknowledge(n, c), s) \equiv \\
&\quad shipmentPos(n, s) = nonExistent \land shipmentSender(n, s) = c
\end{aligned}$$

Successor state axioms:

$$\begin{aligned}
&robotPos(r, do(a, s)) = p \equiv \\
&\quad a = startGoTo(r, p) \land p = unknown \lor \\
&\quad a = reachDest(r) \land p = robotDest(r) \lor \\
&\quad (\forall\, p')a \neq startGoTo(r, p') \land a \neq reachDest(r) \land robotPos(r, s) = p \\
&robotState(r, do(a, s)) = x \equiv \\
&\quad (\exists\, p)a = startGoTo(r, p) \land x = moving \lor \\
&\quad a = reachDest(r) \land x = reached \lor \\
&\quad a = getStuck(r) \land x = idle \lor \\
&\quad a = abortGoTo(r) \land x = idle \lor \\
&\quad (\forall\, p')a \neq startGoTo(r, p') \land a \neq reachDest(r) \land a \neq getStuck(r) \land \\
&\quad\quad a \neq abortGoTo(r) \land robotState(r, s) = x \\
&robotDest(r, do(a, s)) = p \equiv \\
&\quad a = startGoTo(r, p) \lor \\
&\quad (\forall\, p')a \neq startGoTo(r, p') \land robotDest(r, s) = p
\end{aligned}$$

```
$shipmentSender(n, do(a,s)) = c \equiv$
    $(\exists\ c')a = orderShipment(n,c,c') \vee$
    $(\forall\ c')a \neq orderShipment(n,c,c') \wedge shipmentSender(n,s) = c$
$shipmentRecipient(n, do(a,s)) = c \equiv$
    $(\exists\ c')a = orderShipment(n,c',c) \vee$
    $(\forall\ c')a \neq orderShipment(n,c',c) \wedge shipmentRecipient(n,s) = c$
$shipmentPos(n, do(a,s)) = p \equiv$
    $a = acknowledge(n,p) \vee$
    $(\exists\ r)a = pickUp(r,n) \wedge p = r \vee$
    $(\exists\ r)a = dropOff(r,n) \wedge p = robotPos(r,s) \vee$
    $(\forall\ p',r)a \neq acknowledge(n,p') \wedge a \neq pickUp(r,n) \wedge$
      $a \neq dropOff(r,n) \wedge shipmentPos(n,s) = p$
```

Defined fluents:

```
$proc\ clientToServe(r,c)$
    $(\exists n)shipmentPos(n) = c \wedge client(c) \wedge shipmentRecipient(n) \neq c)$
    $\vee$
    $(\exists n')shipmentPos(n') = r \wedge shipmentRecipient(n') = c)$
$end$
```

Defined procedures:

```
$proc\ pickOne(r,c)$
    $(\pi n)$ [ $?(shipmentPos(n) = c \wedge client(c) \wedge$
                 $shipmentRecipient(n) \neq c)$ ;
               $pickUp(r,n)$
             ]
$end$

$proc\ dropAll(r,c)$
    $while\ ((\exists n)shipmentPos(n) = r \wedge shipmentRecipient(n) = c)$
        $(\pi n')$ [ $?(shipmentPos(n') = r \wedge$
                      $shipmentRecipient(n') = c)$ ;
                    $dropOff(r,n')$
                  ]
$end$
```

```
$proc\ handleRequests(r, Max)$
    $?(\neg((\exists c) clientToServe(r, c)))$
    $|$
    $(\pi\ c, p, m)$
        $[\ ?(clientToServe(r, c) \wedge robotPos(r) = p\ \wedge$
            $m = Max - distance(p, c) \wedge m \geq 0);$
         $if\ (p \neq c)\ then$
             $[\ startGoTo(r, c)\ ;\ ?(robotState \neq moving)\ ];$
         $if\ (robotState = reached)\ then$
             $[\ if\ \neg((\exists n) shipmentPos(n) = r \wedge shipmentRecipient(n) = c)\ then$
                 $pickOne(r, c);$
               $dropAll(r, c)$
             $];$
         $handleRequests(r, m)$
        $]$
$end$


$proc\ minimizeMotion(Max)$
    $(\ handleRequests(robot1, Max)\ ||\ handleRequests(robot2, Max)\ ||$
      $handleRequests(robot3, Max)\ )$
    $|$
    $minimizeMotion(Max + 1)$
$end$


$proc\ envSimulator$
    $<\ (\exists r) robotState(r) = moving\ \rightarrow\ sim(reachDest(r))\ >$
$end$


$proc\ control$
    $search(minimizeMotion(0)\ \|\ envSimulator)$
$end$
```

Appendix E

Extended IndiGolog Interpreter

```
/*****************************************************************************
** IndiGolog interpreter with path manipulation                            **
**                                                                          **
** first written by Hector Levesque                                         **
** adapted to Quintus Prolog by Yves Lesperance, April 1999                 **
** modified by Yves Lesperance and Ho Ng                                    **
**                                                                          **
*****************************************************************************/

:- ensure_loaded(library(not)).

:- multifile tracingProg/0, tracingExec/0, tracingTest/0, tracingPath/0,
          tracingLeft/0, prim_action/1, prim_fluent/1, causes_val/4,
          poss/2, proc/2.

:- dynamic tracingProg/0, tracingExec/0, tracingTest/0, tracingPath/0,
        tracingLeft/0.


/*****************************************************************************
** Main Loop : indigolog(E)                                                **
**                                                                          **
** The top level call is indigolog(E), where E is a program.           **
** The history H is a list of actions (prim or exog), initially [].    **
** Sensing reports are inserted as actions of the form e(fluent,value)  **
*************************************************************************/

indigolog(E) :- indigo(E,[]).

indigo(E,H) :- exog_occurs(A), exog_action(A), !, subsim(E,A,H).
indigo(E,H) :- trans(E,H,E1,H1), !, checksim(E,H,E1,H1).
indigo(E,H) :- final(E,H), nl, length(H,N), write(N), write(' actions.'), nl.

checksim(E,H,E1,[sim(_)|H]) :- !, indigo(E,H).
checksim(E,H,E1,H1) :- indixeq(H,H1,H2), !, indigo(E1,H2).

indixeq(H,H,H).    /* for test transitions */
indixeq(H,[A|H],[e(F,Sr),A|H]) :- senses(A,F), !, execute(A,Sr).
indixeq(H,[A|H],[A|H]) :- execute(A,_).
/* Hector's original version
indixeq(H,[Act|H],[Act|H]) :- not senses(Act,_), execute(Act,_).
indixeq(H,[Act|H],[e(F,Sr),Act|H]) :- senses(Act,F), execute(Act,Sr).
*/

subsim(E,A,H) :- trans(E,H,E1,H1), subsim2(E,A,H,E1,H1).
subsim2(E,A,H,E1,[sim(A)|H]) :- !, indigo(E1,[A|H]).
subsim2(E,A,H,E1,H1) :- !, indigo(E,[A|H]).


/*************************************************************************
** indigolog_itr(E)                                                     **
**                                                                      **
** This top level call is similar to indigolog(E), except it calls      **
** start_interrupts at the beginning and stop_interrupts at the end     **
** automatically. Interrupts must be put into conc or pconc structure   **
** to indicate their priorities over other interrupts                   **
** ( e.g. pconc(interrupt(c1,a1),interrupt(c2,a2)) ). User can still put  **
** interrupts into a list and use prioritized_interrupts to convert it   **
** into a pconc structure. However, it is not recommended.              **
**                                                                      **
** Assumption : no interrupt is inside any search block                 **
*************************************************************************/
```

```
indigolog_itr(E) :- indigo(pconc(start_interrupts,
                             pconc(E,
                                   stop_interrupts)),[]).


/**************************************************************************
** exog_occurs and execute are the predicates that make contact with      **
** the outside world.  Here are two basic versions using read and write   **
**************************************************************************/

ask_exog_occurs(Act) :- write('Exogenous input:'), read(Act).
ask_execute(Act,Sr) :-  write(Act), senses(Act,_) -> (write(':'),read(Sr)); nl.


/**************************************************************************
** Trans and Final                                                        **
**************************************************************************/

/*** ConGolog.final ***/

final(conc(E1,E2),H) :- final(E1,H), final(E2,H).
final(pconc(E1,E2),H) :- final(E1,H), final(E2,H).
final(iconc(_),_).


/*** ConGolog.trans.conc ***/

trans(conc(E1,E2),H,conc(E,E2),H1) :- trans(E1,H,E,H1).
trans(conc(E1,E2),H,conc(E1,E),H1) :- trans(E2,H,E,H1).


/*** ConGolog.trans.pconc ***/

/* original version with bugs
trans(pconc(E1,E2),H,E,H1) :-
    trans(E1,H,E3,H1) -> E=pconc(E3,E2) ; (trans(E2,H,E3,H1), E=pconc(E1,E3)).
*/

/* version #2 */
```

```
trans(pconc(E1,E2),H,pconc(E3,E2),H1) :- trans(E1,H,E3,H1).
trans(pconc(E1,E2),H,pconc(E1,E3),H1) :-
    \+ trans(E1,H,E3,H1), trans(E2,H,E3,H1).

/* version #3 (may be a more efficient way)
trans(pconc(E1,E2),H,E,H1) :-
    (trans(E1,H,_,_), !, trans(E1,H,E3,H1), E=pconc(E3,E2)) ;
    (trans(E2,H,E3,H1), E=pconc(E1,E3)).
*/


/*** ConGolog.trans.iconc ***/

trans(iconc(E),H,conc(E1,iconc(E)),H1) :- trans(E,H,E1,H1).


/*** Golog.final ***/

final([],_).
final([E|L],H) :- final(E,H), final(L,H).
final(ndet(E1,E2),H) :- final(E1,H) ; final(E2,H).

/* original version with bugs
final(if(P,E1,E2),H) :- holds(P,H) -> final(E1,H) ; final(E2,H).
*/
final(if(P,E1,E2),H) :- holds(P,H), final(E1,H).
final(if(P,E1,E2),H) :- \+ holds(P,H), final(E2,H).

/* new version for if(cond,progA,else,progB) */
final(if(P,E1,else,E2),H) :- holds(P,H), final(E1,H).
final(if(P,E1,else,E2),H) :- \+ holds(P,H), final(E2,H).

final(star(_),_).
final(while(P,E),H) :- \+ holds(P,H) ; final(E,H).
final(pi(V,E),H) :- subv(V,_,E,E2), final(E2,H).
final(E,H) :- proc(E,E2), final(E2,H).


/*** Golog.trans.sequence ***/
```

```
trans([E|L],H,[E1|L],H2) :- trans(E,H,E1,H2).
trans([E|L],H,E1,H2) :- L \== [], final(E,H), trans(L,H,E1,H2).
/* Hector had:
trans([E|L],H,E1,H2) :- not L=[], final(E,H), trans(L,H,E1,H2).
*/


/*** Golog.trans ***/

trans(?(P),H,[],H) :- holds(P,H), traceTest(P,1,H).
trans(?(P),H,[],H) :- \+ holds(P,H), traceTest(P,0,H), !, fail.

trans(ndet(E1,E2),H,E,H1) :- trans(E1,H,E,H1) ; trans(E2,H,E,H1).

/* original version with bugs:
trans(if(P,E1,E2),H,E,H1) :- holds(P,H) -> trans(E1,H,E,H1) ; trans(E2,H,E,H1).
*/
trans(if(P,E1,E2),H,E,H1) :- holds(P,H), traceTest(P,1,H), trans(E1,H,E,H1).
trans(if(P,E1,E2),H,E,H1) :- \+ holds(P,H), traceTest(P,0,H), trans(E2,H,E,H1).

/* new version for if(cond,progA,else,progB) */
trans(if(P,E1,else,E2),H,E,H1) :-
    holds(P,H), traceTest(P,1,H), trans(E1,H,E,H1).
trans(if(P,E1,else,E2),H,E,H1) :-
    \+ holds(P,H), traceTest(P,0,H), trans(E2,H,E,H1).

trans(star(E),H,[E1,star(E)],H1) :- trans(E,H,E1,H1).

trans(while(P,E),H,[E1,while(P,E)],H1) :-
    holds(P,H), traceTest(P,1,H), trans(E,H,E1,H1).
trans(while(P,E),H,[E1,while(P,E)],H1) :-
    \+ holds(P,H), traceTest(P,0,H), !, fail.

trans(pi([V|L],E),H,E1,H1) :- subvs([V|L],E,E2), trans(E2,H,E1,H1).
trans(pi(V,E),H,E1,H1) :- subv(V,_,E,E2), trans(E2,H,E1,H1).

/* added for meta_follow */
trans(execute_update(A,LS,[[A|H]|LS]),H,[],[A|H]) :-
    prim_action(A), poss(A,P), holds(P,H).
```

```
trans(E,H,E1,H1) :- proc(E,E2), trans(E2,H,E1,H1).

trans(E,H,[],[E|H]) :- prim_action(E), poss(E,P), holds(P,H).

trans(no_op,H,[],H).



/**************************************************************************
** Original Search                                                      **
**                                                                      **
** It ignores exogenous or other concurrent actions.                    **
**************************************************************************/

final(osearch(E),H) :- final(E,H).
trans(osearch(E),H,ofollowpath(E1,L),H1) :-
    trans(E,H,E1,H1), ofindpath(E1,H1,L), tracePath(L,H), traceLeft(E1).

ofindpath(E,H,[E,H]) :- final(E,H).
ofindpath(E,H,[E,H|L]) :- trans(E,H,E1,H1), ofindpath(E1,H1,L).

final(ofollowpath(E,[E,H]),H) :- !.
final(ofollowpath(E,_),H) :- final(E,H).
trans(ofollowpath(E,[E,H,E1,H1|L]),H,ofollowpath(E1,[E1,H1|L]),H1) :-
    traceLeft(E1), !.
trans(ofollowpath(E,_),H,E1,H1) :- trans(osearch(E),H,E1,H1).


/**************************************************************************
** Search (from the beginning)                                          **
**                                                                      **
** It ignores exogenous or other concurrent actions.                    **
**************************************************************************/

/* extactin(+list of snapshots, +[], +list of actions) and
   extractout(+list of snapshots, +[], +list of actions)
   It returns the list of actions AL that has been performed from the
   beginning of search by making extact(LS,[],AL) holds */

extactin([H0],AL,AL).
extactin([[sim(A)|H1],H|L],AL,R) :- extactout([H1,H|L],AL,R).
extactin([[A|H1],H|L],AL,R) :- extactout([H1,H|L],[inside(A)|AL],R).

extactout([H,H|L],AL,R) :- extactin([H|L],AL,R).
extactout([H,[sim(A)|H]|L],AL,R) :- extactin([[sim(A)|H]|L],AL,R).
extactout([[A|H1],H|L],AL,R) :- extactout([H1,H|L],[A|AL],R).


/* posspath(+path, +current history)
   It checks to see if it is still possible to continue with the path */

posspath([E,H],CH) :- final(E,CH).
posspath([E,H,E1,H|L],CH) :- trans(E,CH,E1,CH), posspath([E1,H|L],CH).
posspath([E,H,E1,[A|H]|L],CH) :-
    trans(E,CH,E1,[A|CH]), posspath([E1,[A|H]|L],[A|CH]).


/* findpath(AL,E,H,L) holds if there is a legal execution of E in H that */
/* first xeqs the actions in AL, and then finishes with L, a pathlist as */
/* above to a final state                                                */

findpath([],E,H,[E,H]) :- final(E,H).
findpath([],E,H,[E,H|L]) :- trans(E,H,E1,H1), findpath([],E1,H1,L).

findpath([inside(A)|AL],E,H,L) :-
    prim_action(A), trans(E,H,E1,[A|H]), findpath(AL,E1,[A|H],L).
findpath([A|AL],E,H,L) :-
    A \= inside(_), findpath(AL,E,[A|H],L).
findpath([A|AL],E,H,L) :-    /* silent */
    trans(E,H,E1,H), findpath([A|AL],E1,H,L).


/*** search ***/

final(search(E),H) :- final(E,H).
trans(search(E),H,path(P1,LS1,E,H),H1) :-
    findpath([],E,H,P0), trans(path(P0,[H],E,H),H,path(P1,LS1,E,H),H1),
    tracePath(P1,H).
```

```
/* the structure path(L,LS,E0,H0) is used as a pseudo program where      */
/*     L is a list E1,H1,E2,H2...En,Hn such that                          */
/*     trans(Ei,Hi,Ei+1,Hi+1) and final(En,Hn) both hold                  */
/*     LS is a list of snapshots                                          */
/*     E0 and H0 is where the search started (in case we need to restart) */

final(path([E,H],LS,E0,H0),H) :- !.
final(path([E,H],LS,E0,H0),CH) :- final(E,CH), !.
final(path([E,H,E1,H1|L],LS,E0,H0),H) :- !, fail.
final(path([E,H|L],LS,E0,H0),CH) :-
     extactout([CH|LS],[],AL), !, findpath(AL,E0,H0,[E1,H1]).

trans(path([E,H,E1,H|L],LS,E0,H0),H,path([E1,H|L],LS,E0,H0),H) :-
    traceLeft(E1), !.
trans(path([E,H,E1,[A|H]|L],LS,E0,H0),H,path([E1,[A|H]|L],[[A|H]|LS],E0,H0),[A|H]) :-
    traceLeft(E1), !.
trans(path([E,H,E1,H|L],LS,E0,H0),CH,path([E1,H|L],LS,E0,H0),CH) :-
    posspath([E,H,E1,H|L],CH), traceLeft(E1), !.
trans(path([E,H,E1,[A|H]|L],LS,E0,H0),CH,path([E1,[A|H]|L],[[A|CH]|LS],E0,H0),[A|CH]) :
    posspath([E,H,E1,[A|H]|L],CH), traceLeft(E1), !.
trans(path([E,H,E1,H1|L],LS,E0,H0),CH,path(P2,LS2,E0,H0),H2) :-
    extactout([CH|LS],[],AL),
    findpath(AL,E0,H0,P),
    trans(path(P,LS,E0,H0),CH,path(P2,LS2,E0,H0),H2),
    tracePath(P2,CH).


/**************************************************************************
** Simulated Actions : sim(A)                                            **
**************************************************************************/

prim_action(sim(A)) :- exog_action(A).
poss(sim(A),P)      :- poss(A,P).


/**************************************************************************
** Interrupts                                                            **
**************************************************************************/

prim_action(start_interrupts).
prim_action(stop_interrupts).
prim_fluent(interrupts).

causes_val(start_interrupts, interrupts, running, true).
causes_val(stop_interrupts, interrupts, stopped, true).

poss(start_interrupts, true).
poss(stop_interrupts,  true).

proc(interrupt(V,Trigger,Body),            /* version with variable */
    while(interrupts=running, pi(V,if(Trigger,Body,?(neg(true)))))).
proc(interrupt(Trigger,Body),              /* version without variable */
    while(interrupts=running, if(Trigger,Body,?(neg(true))))).
proc(prioritized_interrupts(L),[start_interrupts,E]) :- expand_interrupts(L,E).

expand_interrupts([],stop_interrupts).
expand_interrupts([X|L],pconc(X,E)) :- expand_interrupts(L,E).


/***************************************************************************
** Hold (holds)                                                           **
***************************************************************************/

holds(and(P1,P2),H)    :- !, holds(P1,H), holds(P2,H).
holds(or(P1,P2),H)     :- !, (holds(P1,H) ; holds(P2,H)).
holds(neg(P),H)        :- !, \+ holds(P,H).   /* Negation by failure */
/* do we have to add the cut in the holds.some.[V|L]? */
holds(some([V|L],P),H) :- !, subvs([V|L],P,P1), holds(P1,H).
holds(some(V,P),H)     :- !, subv(V,_,P,P1), holds(P1,H).

/* shouldn't do subsitution if the condition contains a path */
holds(P,H) :- ( P = posspath(_,_) ;
                P = finalpath(_) ;
                P = transpath(_,_,_) ;
                P = finalpathseg(_) ;
                P = transpathseg(_,_,_) ;
                P = assignTo(_,_) ),
              !, call(P).

holds(P,H)             :- proc(P,P1), !, holds(P1,H).
holds(P,H)              :- subf(P,P1,H), call(P1).
/* Hector's original version
holds(P,H) :- proc(P,P1), holds(P1,H).
holds(P,H) :- not proc(P,P1), subf(P,P1,H), call(P1).
*/


/*************************************************************************
** Substitution                                                         **
*************************************************************************/

/*** subv(X1,X2,T1,T2) holds iff T2 is T1 with X1 replaced by X2 ***/
subv(_,_,T1,T2)   :- (var(T1);integer(T1)), !, T2 = T1.
subv(X1,X2,T1,T2) :- T1 = X1, !, T2 = X2.
subv(X1,X2,T1,T2) :- T1 =..[F|L1], subvl(X1,X2,L1,L2), T2 =..[F|L2].

subvl(_,_,[],[]).
subvl(X1,X2,[T1|L1],[T2|L2]) :- subv(X1,X2,T1,T2), subvl(X1,X2,L1,L2).

subvs([],T1,T1)    :- !.
subvs([V|L],T1,T2) :- subv(V,_,T1,T3), subvs(L,T3,T2).

/*** subf(P1,P2,H) holds iff
         P2 is P1 with all fluents replaced by their values ***/
subf(P1,P2,_) :- (var(P1);integer(P1)), !, P2 = P1.
subf(P1,P2,H) :- prim_fluent(P1), has_val(P1,P2,H).
subf(P1,P2,H) :- \+ prim_fluent(P1), P1=..[F|L1], subfl(L1,L2,H), P2=..[F|L2].
/* Hector's original version
subf(P1,P2,H) :- prim_fluent(P1), has_val(P1,P2,H).
subf(P1,P2,H) :- not prim_fluent(P1), P1=..[F|L1], subfl(L1,L2,H), P2=..[F|L2].
*/

subfl([],[],_).
subfl([T1|L1],[T2|L2],H) :- subf(T1,T2,H), subfl(L1,L2,H).

/* for fluent now */
has_val(now,H,H)        :- !.

has_val(F,V,[])         :- initially(F,V).
has_val(F,V,[sim(A)|H]) :- !, has_val(F,V,[A|H]).
has_val(F,V,[A|H])      :- sets_val(A,F,V1,H) -> V = V1 ; has_val(F,V,H).

sets_val(A,F,V,H) :- A = e(F,V) ; (causes_val(A,F,V,P), holds(P,H)).


/***********************************************************************
** Program Tracing                                                    **
**                                                                    **
** There are five clauses which can be asserted for tracing:          **
**     tracingProg  - (useless, not supported)                        **
**     tracingTest  - show each test action and its result            **
**     tracingExec  - (covered by the clause execute(A,_), not supported) **
**     tracingPath  - show the path found by the planner              **
**     tracingLeft  - show the advanced program during path following **
***********************************************************************/

/* rePrintHist(+,+)
   [history list, level] */

rePrintHist([],_).
rePrintHist(_,0)     :- !, write(', ...').
rePrintHist([A|H],L) :- write(', '), write(A), L2 is L-1, rePrintHist(H,L2).

/* printHist(+,+)
   [history list, level (>0)]
   It prints the first 3 actions in the history list */

printHist([],_)    :- write('[]').
printHist([A|H],L) :-
    write('['), write(A), L2 is L-1, rePrintHist(H,L2), write(']').


/* addtail(+,+,-)
   [element, original list, list with element at the end] */
addtail(E,[],[E]).
addtail(E,[H|L1],[H|L2]) :- addtail(E,L1,L2).

/* reverseList(+,-)
   [original list, reversed list] */
```

```
reverseList([],[]).
reverseList([H|L1],L2) :- reverseList(L1,L3), addtail(H,L3,L2).

/* minusTail(+,+,-)
   [tail list, list with head and tail, head list] */

minusTail(H,H,[]).
minusTail(H,[E|H1],[E|H2]) :- minusTail(H,H1,H2).

/* getPath(+,+,-)
   [path list, current history, path] */
getPath([],H,[]).
getPath([E1,H1],H,P) :- !, minusTail(H,H1,H2), reverseList(H2,P).
getPath([E1,H1|L],H,P) :- getPath(L,H,P).


/* traceTest(+,+,+)
   [complex predicate, value, history]
   It prints the predicate, its value and also the history list */

traceTest(_,_,_) :- \+ clause(tracingTest,_), !.
traceTest(P,V,H) :-
     format('  test : ~p <- ',[V]), format('~p in ',[P]), printHist(H,3), nl, !.


/* tracePath(+,+)
   [path generated from search, history list]
   It prints the path or plan generated by search() */

tracePath(_,_)  :- \+ clause(tracingPath,_), !.
tracePath([],H) :- write('  path : [] in '), printHist(H,3), nl.
tracePath(L,H)  :-
    write('  path : '), getPath(L,H,P), printHist(P,15), write(' in '),
    printHist(H,3), nl.


/* traceLeft(+)
   [program]
   It prints the advanced program when the executor is following a path */
```

```
traceLeft(_) :- \+ clause(tracingLeft,_), !.
traceLeft(E) :- write('  left : '), write(E), nl.



/**************************************************************************
** Tools for External Path Manipulation                                  **
**************************************************************************/


/* now
   A primitive fluent for obtaining the current history in the user program. */
prim_fluent(now).
/* (added) has_val(now,H,H) :- !. */


/* path evaluation
   a) evaluate path step by step
      finalpath(+Path)
          - succeed when Path is [E,H]
      transpath(+Path,-Path,-Action)
          - advance the path by one step and return the involved action if
            there is one
   b) evaluate path as plan (also works for program with sim action)
      finalpathseg(+Path)
      - succeed when the following the Path does not involve any action
      transpathseg(+Path,-Path,-Action)
      - advance the path upto the point where an action is involved */

finalpath([E,H]).

transpath([E,H,E1,H|L],[E1,H|L],no_op).
transpath([E,H,E1,[A|H]|L],[E1,[A|H]|L],A).

finalpathseg([E,H]).
finalpathseg([E,H,E1,H|L]) :- finalpathseg([E1,H|L]).

transpathseg([E,H,E1,H|L],PS,A) :- transpathseg([E1,H|L],PS,A).
transpathseg([E,H,E1,[A|H]|L],[E1,[A|H]|L],A).
```

```
/* (added) holds(P,H) :- (P=posspath(_,_); P=finalpath(_); P=transpath(_,_,_);
                         P=finalpathseg(_); P=transpathseg(_)), !, call(P).
   Each of these prolog predicates contain a path. The values of the fluents
   inside this path should not be affected by the current history H. Therefore,
   no subsitution should be done on P. */


/* (added) trans(execute_update(A,LS,[[A|H]|LS]),H,[],[A|H]) :-
              prim_action(A), poss(A,P), holds(P,H).
   to support executing an action and updating the snapshots at the same time
   in meta_follow

   (?) I guess this can be removed by doing:
       if (ls1 = [[a|now]|ls],
       [ a, ... ]
       )
   instead of [ execute_update(a,ls,ls1), ... ] */


/* The prolog condition update_ls caches the list of snapshots in the
   memory. It can be retrieved by calling listOfSnapshots(LS) inside the
   condition for passing simulated action in the call of meta_follow.

   (?) Is this useless? */

update_ls(LS) :-
    clause(listOfSnapshots(_),_), !, retract(listOfSnapshots(_)),
    asserta(listOfSnapshots(LS)).
update_ls(LS) :- asserta(listOfSnapshots(LS)).


/***************************************************************************
** Meta Planner and Executor                                              **
***************************************************************************/

/* insert all new possible branches into the database */
queue_push(N) :-
    clause(branchR(N,AL,E,H,P),_),
    retract(branchR(N,AL,E,H,P)),
    asserta(branch(N,AL,E,H,P)),
    fail.
queue_push(_).

stack_push(N,[],E,H,P) :-
    trans(E,H,E1,H1), asserta(branchR(N,[],E1,H1,[E1,H1|P])), fail.
stack_push(N,[inside(A)|AL],E,H,_) :-
    trans(E,H,E1,[A|H]), asserta(branchR(N,AL,E1,[A|H],[E1,[A|H]])), fail.
stack_push(N,[A|AL],E,H,_) :-
    A \= inside(_), asserta(branchR(N,AL,E,[A|H],[E,[A|H]])), fail.
stack_push(N,[A|AL],E,H,_) :-
    trans(E,H,E1,H), asserta(branchR(N,[A|AL],E1,H,[E1,H])), fail.
stack_push(N,_,_,_,_) :-
    queue_push(N).


/* call as reverselist(AL,[],Result) */
reverselist([E,H],L,[E,H|L]).
reverselist([E,H|L1],L2,R) :- reverselist(L1,[E,H|L2],R).


/* get the first unvisited branch of search tree N */
firstbranch(N,AL,E,H,P) :- clause(branch(N,AL,E,H,P),_), !.


/* new version of findpath,
   which will return either a path [...] or search_log(N) */
contfindpath(branch(N,[],E,H,P),Cond,Path) :-
    final(E,H), reverselist(P,[],Path).
contfindpath(branch(N,AL,E,H,P),Cond,Path) :-
    stack_push(N,AL,E,H,P),    /* trans(branches) */
    ( (holds(Cond,[]), Path = search_log(N)) ;
      findpath(search_log(N),Cond,Path)).

findpath(AL,E0,H0,Cond,Path) :-
    reserveMemory(AL,E0,H0,SLog),
    findpath(SLog,Cond,Path).

findpath(search_log(N),Cond,Path) :-
    firstbranch(N,AL,E,H,P),
    retract(branch(N,AL,E,H,P)),
    contfindpath(branch(N,AL,E,H,P),Cond,Path).


/* pick a new number N and convert the search info into a search_log(N)
   object */
reserveMemory(AL,E0,H0,search_log(1)) :-
    \+ clause(memoryCounter(X),_),
    asserta(memoryCounter(1)),
    asserta(branch(1,AL,E0,H0,[E0,H0])).
reserveMemory(AL,E0,H0,search_log(N)) :-
    clause(memoryCounter(X),_),
    retract(memoryCounter(X)),
    N is X + 1,
    asserta(memoryCounter(N)),
    asserta(branch(N,AL,E0,H0,[E0,H0])).


/* Condition meta_search(+E0,+H0,+LS,-Path)
   It finds a path by searching from the initial configuration.
   E0   : the original history
   H0   : the initial history
   LS   : list of snapshots
   Path : the path that will be found and returned by meta_search */

proc(rsearch(E,PauseCond,SimCond,Remainer),
    pi(h0, [ ?(and(neg(E = remain(_,_,_,_)),
               and(neg(E = rpath(_,_,_,_)),
                   h0 = now))),
             rsearch(E,h0,[h0],PauseCond,SimCond,Remainer)
           ]
    )
).

/* for no subsitution on path assignment */
assignTo(E,E).

/* a version of rsearch that should not be called in the user program */
proc(rsearch(E0,H0,LS,PauseCond,SimCond,Remainer),
    pi([h0,al,branches,mypath],
        [ ?(and(h0 = now,
            and(extactout([h0|LS],[],al),
            and(reserveMemory(al,E0,H0,branches),
                findpath(branches,PauseCond,mypath))))),
          if(mypath = search_log(_),
              ?(assignTo(Remainer,remain(mypath,E0,H0,LS))),
          else,
              [ ?(tracePath(mypath,h0)),
                if(PauseCond,
                    ?(assignTo(Remainer,rpath(mypath,E0,H0,LS))),
                else,
                    rsearch(rpath(mypath,E0,H0,LS),PauseCond,SimCond,Remainer)
                )
              ]
          )
        ]
    )
).

/* a version of rsearch to handle remain structure */
proc(rsearch(remain(search_log(N),E0,H0,LS),PauseCond,SimCond,Remainer),
    pi(mypath,
        [ ?(findpath(search_log(N),PauseCond,mypath)),
          if(mypath = search_log(_),
              ?(assignTo(Remainer,remain(mypath,E0,H0,LS))),
          else,
              if(PauseCond,
                  ?(assignTo(Remainer,rpath(mypath,E0,H0,LS))),
              else,
                  rsearch(rpath(mypath,E0,H0,LS),PauseCond,SimCond,Remainer)
              )
          )
        ]
    )
).

/* a version of rsearch to handle rpath structure */
proc(rsearch(rpath(Path,E0,H0,LS),PauseCond,SimCond,Remainer),

    /* should not check PauseCond at here (pi.mypath = Path) */
    /* check if the path has been completed */
    if(finalpathseg(Path),
        if(some(ch1, and(ch1 = now, posspath(Path,ch1))),
            ?(Remainer = []),
        else,
    rsearch(E0,H0,LS,PauseCond,SimCond,Remainer)    /* replan */
        ),

    /* try to do a path transition */
    else,
        pi([p,a],
           [ ?(transpathseg(Path,p,a)),

             if(a = sim(_),
                 [ ?(and(update_ls(LS),
                         SimCond)),       /* wait for condition */
                   rsearch(rpath(p,E0,H0,LS),PauseCond,SimCond,Remainer)
                 ],

             else,
                 if(some(ch2, and(ch2 = now, posspath(Path,ch2))),
                     pi(ls1,
                        [ execute_update(a,LS,ls1),   /* perform action a */
                          rsearch(rpath(p,E0,H0,ls1),PauseCond,SimCond,Remainer)
                        ]
                     ),
                 else,
                     rsearch(E0,H0,LS,PauseCond,SimCond,Remainer)  /* replan */
                 )
             )
           ]
        )
    )
).


/****************************************************************************
** Meta Planner and Executor (simple version)                              **
****************************************************************************/
```

```
mfindpath([],E,H,Cond,[E,H]) :- final(E,H).
mfindpath(AL,E,H,Cond,[]) :- holds(Cond,[]), !.
mfindpath([],E,H,Cond,P) :-
    trans(E,H,E1,H1), mfindpath([],E1,H1,Cond,L),
    ((L = [], P = []) ; P = [E,H|L]).

mfindpath([A|AL],E,H,Cond,[]) :- holds(Cond,[]), !.
mfindpath([inside(A)|AL],E,H,Cond,L) :-
    prim_action(A), trans(E,H,E1,[A|H]), mfindpath(AL,E1,[A|H],Cond,L).
mfindpath([A|AL],E,H,Cond,L) :-
    A \= inside(_), mfindpath(AL,E,[A|H],Cond,L).
mfindpath([A|AL],E,H,Cond,L) :-
    trans(E,H,E1,H), mfindpath([A|AL],E1,H,Cond,L).


proc(executepath(Path,E0,H0,LS,PCond,SCond,Status),
    if(finalpathseg(Path),
        if(some(ch1, and(ch1 = now, posspath(Path,ch1))),
            ?(Status = []),
        else,
            ?(Status = remain(E0,H0,LS))
        ),
    else,
        if(PCond,
            ?(assignTo(Status,rpath(Path,E0,H0,LS))),
        else,
            pi([p,a],
                [ ?(transpathseg(Path,p,a)),
                  if(some(ch2, and(ch2 = now, and(posspath(Path,ch2),
                                  a = sim(_)))),
                      [ a,   /*?(and(update_ls(LS), SCond)),*/
                        executepath(p,E0,H0,LS,PCond,SCond,Status)
                      ],
                  else,
                      if(some(ch3, and(ch3 = now, posspath(Path,ch3))),
                          pi(ls1,
                              [ execute_update(a,LS,ls1),
                                executepath(p,E0,H0,ls1,PCond,SCond,Status)
                              ]
```

```
                         ),
                     else,
                         ?(Status = remain(E0,H0,LS))
                     )
                 )
               ]
            )
        )
    )
).

proc(msearch(E,PCond,SCond,Status),
    pi(h0, [ ?(and(neg(E = remain(_,_,_)),
               and(neg(E = rpath(_,_,_,_)),
                   h0 = now))),
             msearch(E,h0,[],PCond,SCond,Status)
           ]
    )
).

proc(msearch(E0,H0,LS,PCond,SCond,Status),
    pi([mypath,h0,ls1,al],
       [ ?(and(h0 = now,
           and(addtail(H0,LS,ls1),
           and(extactout([h0|ls1],[],al),
           and(mfindpath(al,E0,H0,PCond,mypath),
               tracePath(mypath,h0)))))),

         if(mypath = [],   /* is this useless? */
             ?(Status = remain(E0,H0,LS)),
         else,
             if(PCond,
                 ?(assignTo(Status,rpath(mypath,E0,H0,LS))),
             else,
                 pi(ans,
                    [ executepath(mypath,E0,H0,LS,PCond,SCond,ans),
                      if(ans = remain(_,_,_),
                          pi(ls2,
                             [ ?(remain(E0,H0,ls2) = ans),
                               msearch(E0,H0,ls2,PCond,SCond,Status)
                             ]
                          ),
                      else,
                          ?(assignTo(Status,ans))
                      )
                    ]
                 )
             )
         )
       ]
    )
).

proc(msearch(remain(E0,H0,LS),PCond,SCond,Status),
    msearch(E0,H0,LS,PCond,SCond,Status)
).

proc(msearch(rpath(Path,E0,H0,LS),PCond,SCond,Status),
    pi(ans,
       [ executepath(Path,E0,H0,LS,PCond,SCond,ans),
         if(ans = remain(_,_,_),
             pi(ls2,
                [ ?(remain(E0,H0,ls2) = ans),
                  msearch(E0,H0,ls2,PCond,SCond,Status)
                ]
             ),
         else,
             ?(assignTo(Status,ans))
         )
       ]
    )
).
```

Appendix F

Routines for TCP/IP Communication for IndiGolog

```
/***************************************************************************
**                                                                       **
** ICP/IP Connection Routines for IndiGolog (April 17, 2000)             **
**                                                                       **
***************************************************************************/

:- use_module(library(tcp)).


/* init_tcpip (initialization) */

init_tcpip :-
    prolog_flag(fileerrors,_,on),
    prolog_flag(syntax_errors,_,error).


/* create_socket(+Host,+Port)                                            */
/* It generates a socket for tcpip communication.                        */
```

```
create_socket(Host,Port) :-
    tcp_create_listener(address(Port,Host),PassiveSocket).


/* accept_link                                                          */
/* It initializes the communication link to another socket if an        */
/* acknowledgement has been sent from a connection. It initializes the  */
/* link by adding the clause connection(NAME,SOCKET) into the system    */
/* database, where NAME is the name that represents the other end of the */
/* link, and SOCKET is the socket number.                               */

accept_link :-
    tcp_select(0, term(GuiSocket,GuiName)),
    asserta(connection(GuiName,GuiSocket)),
    write('Accepted connection '), write(GuiSocket),
    write(' from '), write(GuiName), nl.


/* create_link(+Name,+Listener,+Host,+Port)                             */
/* Name is the name of the client, Listener is the name of the server,  */
/* Host is the host name of the server, Port is the port number.        */

create_link(Name,Listener,Host,Port) :-
    tcp_connect(address(Port,Host),Socket),
    asserta(connection(Listener,Socket)),
    send_term_message(Listener,Name),
    write('Created connection '), write(Socket),
    write(' from '), write(Listener), nl.


/* close_connections                                                    */
/* It closes all connections.                                           */

close_connections :-
    tcp_connected(Socket),
    tcp_shutdown(Socket),
    fail.
close_connections.
```

```
/* send_message(+GuiName,+Message)                                        */
/* It sends the input Message to the GUI GuiName.                         */

send_message(GuiName,Message) :-
    connection(GuiName,GuiSocket),
    tcp_output_stream(GuiSocket,Stream),
    format(Stream, '~w~n', [Message]),
    flush_output(Stream).


/* send_message(+GuiName,+Message)                                        */
/* It sends the message term(Message). to the GUI GuiName.                */

send_term_message(GuiName,Message) :-
    connection(GuiName,GuiSocket),
    tcp_output_stream(GuiSocket,Stream),
    format(Stream, 'term(~w).~n', [Message]),
    flush_output(Stream).


/* read_message(+GuiName,-Message)                                        */
/* It reads a Message from the GUI GuiName.                               */

read_message(GuiName,Message) :-
    connection(GuiName,GuiSocket),
    tcp_input_stream(GuiSocket,Stream),
    read(Stream,Message).


/* read_poll_message(-GuiName,-Message)                                   */
/* It reads a message from some connected link. The argument Message is  */
/* the obtained message and GuiName is where the message came from.       */
/* Notice that it is able to get the message only if the message was     */
/* sent in the format term(Message). .                                    */

read_poll_message(GuiName,Message) :-
    tcp_select(0, term(GuiSocket,Message)),
    connection(GuiName,GuiSocket).
```

```
/* get_buf_message(-Sender,-Message)                                    */
/* It reads a Message from the Sender. If there is a message in the     */
/* buffer, it gets that message. If the buffer is empty, it gets a      */
/* message from the socket.                                             */

get_buf_message(Sender,Message) :-
    clause(message_buffer(Sender,Message),_),
    !,
    retract(message_buffer(Sender,Message)).
get_buf_message(Sender,Message) :-
    read_poll_message(Sender,Message).


/* peek_buf_message(-Sender,-Message)                                   */
/* It reads a Message from Sender and then leaves the message in the    */
/* buffer.                                                              */

peek_buf_message(Sender,Message) :-
    clause(message_buffer(Sender,Message),_), !.

peek_buf_message(Sender,Message) :-
    read_poll_message(Sender,Message),
    assertz(message_buffer(Sender,Message)).


/* no_buf_message                                                       */
/* It checks if there is a message for read.                            */

no_buf_message :- \+ peek_buf_message(Sender,Message).
```

Appendix G

Graphical User Interface for Shipment Delivery

```
::::::::::::::
Connector.java
::::::::::::::

import java.io.*;
import java.net.*;


/*****************************************************************************
* Class Connector                                                           *
* It provides facilities for the GUI to send and receive messages from the  *
* IndiGolog control through a tcpip communication link.                     *
* Written by Ho Ng (March 29, 2000)                                         *
*****************************************************************************/
public class Connector
{
    private String linkName;      // name of the communication link
    private String hostName;      // name of the host
private int    portNum;       // port number

private Socket         sckt;
private PrintWriter    output;
private BufferedReader input;
private boolean        connected;


/**
* Constructor of class Connector
*/
public Connector(String name, String host, int port) {
    linkName  = new String(name);
    hostName  = new String(host);
    portNum   = port;
    connected = false;
}

/**
* It sends the input message to the IndiGolog control
* Pre : connectedToIndi() == true
*/
public void sendMessage(String message) {
    output.println("term(" + message + ").");
}

/**
* It reads a message from the IndiGolog control. If there is no
* incoming message, then it will return nil
* Pre : connectedToIndi() == true
*/
public String receiveMessage() {
    String message = null;

    try {
        /* check if there is any message from the IndiGolog control */
        message = input.readLine();

        /* remove "term(" and ")" */
message = message.substring(5,message.length()-2);
```

```
    }
    catch (IOException e) {
        System.err.println("Could not read message from socket");
    }

    return message;
}

/**
* It creates a communication link to the IndiGolog control. Notice that
* it will send its name as an acknowledgement to the IndiGolog control
* after the connection is created successfully
* Pre : connectedToIndi() == false
*/
public void connectToIndi() {
    try {
        /* establish the connection to the IndiGolog program */
        sckt   = new Socket(hostName, portNum);
        input  = new BufferedReader(new InputStreamReader(sckt.getInputStream()));
        output = new PrintWriter(sckt.getOutputStream(), true);

        connected = true;

        /* acknowledge the IndiGolog control */
        sendMessage(linkName.toLowerCase());
    }
    catch (IOException e) {
        System.err.println("Failed to create the I/O stream");
    }
    catch (NullPointerException e) {
        System.err.println("Could not establish the connection");
    }
}

/**
* It closes the communication link to the IndiGolog control
* Pre : connectedToIndi() == true
*/
public void disconnectFromIndi() {
    try {
            /* close the connection */
            input.close();
            output.close();
            sckt.close();
            connected = false;
        }
        catch (IOException e) {
            System.err.println("Could not close the connection");
        }
    }

    /**
    * It returns true if the communication link to the IndiGolog control
    * has been established
    */
    public boolean connectedToIndi() {
        return connected;
    }

    /**
    * A dumb test program
    */
    public static void main(String[] s) {
        Connector x = new Connector("test", s[0],
                                    Integer.valueOf(s[1]).intValue());
        x.connectToIndi();
        if (x.connectedToIndi() == true) {
            System.out.println("Connected to IndiGolog control");
            x.disconnectFromIndi();
        }
        else
            System.out.println("Unable to establish the connection");
    }
};


::::::::::::::
OrderConnector.java
::::::::::::::
```

```
/***********************************************************************
** Class Connector
** It provides the facilities for a client to generate the connection for
** sending and receiving messages from ConGolog/Indigolog interpreter.
***********************************************************************/

import java.awt.*;
import java.io.*;
import java.awt.event.*;
import java.net.*;


class OrderConnector extends Thread
{
    private Socket         sckt;    // socket for communication
    private BufferedReader in;      // input stream descriptor
    private PrintWriter    out;     // output stream descriptor
    private boolean        connected;

    private String          hostName;    // host name of the robot
    private int             portNum;    // port number of the robot
    private String          userName;
    private ClientInterface uinf;

    private boolean msgToSend;
    private String  message;


    /**
    * Constructor of class Connector
    * Input : host - host name of the robot control (interpreter)
    *       : port - port number of the robot control (interpreter)
    *       : name - name of the client
    */
    public OrderConnector(String name, String host, int port)
    {
        hostName = new String(host);
        portNum  = port;
        userName = new String(name.toLowerCase());
        uinf     = null;
    sckt      = null;
    in        = null;
    out       = null;
    connected = false;

    msgToSend = false;
    message   = null;
}

/**
* Purpose : It stores the link for refering the textual user interface
* Pre     : inter != null
*/
public void setInterface(ClientInterface myInterface)
{
    uinf = myInterface;
}

/**
* Purpose : It shuts down the connection
*/
public void shutdown()
{
    connected = false;
}

/**
* Purpose : It stores a message in its buffer
* Pre     : outMessage != null
*/
public void writeMessage(String outMessage)
{
    message   = outMessage;
    msgToSend = true;
}

/**
* Purpose : It sends the message that is in the buffer to the
*           control (indigolog program)
```

```
    */
    public void sendMessage() throws InterruptedException, IOException
    {
        String incomingMsg = null;

if (connected)
{
    try
    {
/* send the message */
out.println("term(" + message + ").");
//sckt.setSoTimeout(0);

msgToSend = false;
    }
    catch (InterruptedException e)
            {
    }
    catch (IOException e)
    {
    }
}
    }

    /**
    * It creates a communication link to the IndiGolog control. Notice that
    * it will send its name as an acknowledgement to the IndiGolog control
    * after the connection is created successfully
    * Pre : connectedToIndi() == false
    */
    public void connectToIndi()
    {
        try
        {
            /* establish the connection to the IndiGolog program */
            sckt = new Socket(hostName, portNum);
            in   = new BufferedReader(new InputStreamReader(sckt.getInputStrea
m()));
            out  = new PrintWriter(sckt.getOutputStream(), true);
            connected = true;
        /* acknowledge the IndiGolog control by send the name */
/* of the user */
        writeMessage(userName);
        sendMessage();
    }
    catch (IOException e)
    {
        System.err.println("Failed to create the I/O stream");
    }
    catch (NullPointerException e)
    {
        System.err.println("Could not establish the connection");
    }
}

/**
* It closes the communication link to the IndiGolog control
* Pre : connectedToIndi() == true
*/
public void disconnectFromIndi()
{
    try
    {
        /* close the connection */
        input.close();
        output.close();
        sckt.close();
        connected = false;
    }
    catch (IOException e)
    {
        System.err.println("Could not close the connection");
    }
}

/**
* Purpose : Thread that listens to the socket
*/
public void run()
```

```
    {
        String  incomingMsg = null;
        String  suspendMsg  = "term(notifyStopServing(" + userName + "))."; 
        long    delay = 50;

/* try to connect to the control */
connectToIndi();

        /* loop to receive and send messages */
        while (connected)
        {
            try
            {
                /* check if some messages to send */
                if (msgToSend)
                    sendMessage();

                /* check if some messages to read */
                sckt.setSoTimeout(50);
                incomingMsg = in.readLine();
                if (incomingMsg != null)
                    uinf.parseMessage(incomingMsg);
            }
            catch (InterruptedException e)
            {
                // it is fine in this case
            }
            catch (InterruptedIOException e)
            {
                // it is fine in this case
            }
            catch (IOException e)
            {
                System.err.println("Connector: Fail to read from/write to socket");
                System.exit(1);
            }

            /* delay for a few seconds */
            try
            {
                sleep(delay);
            }
            catch (InterruptedException e)
            {
            }
        }

        /* close all I/O streams and the socket */
disconnectFromIndi();
    }
};


::::::::::::::
OrderGUI.java
::::::::::::::

/*****************************************************************************
* Program : OrderGUI
*
* A graphical user interface for a client to communicate with the shipment
* delivery robot system.
*
* Written by: Ho Ng (March 29, 2000)
*
*
* To compile this program, use the command:
*
*     javac OrderGUI.java
*
* To run the this program when the indigolog control program is running,
* use the command:
*
*     java OrderGUI <client> <indi_host> <indi_port>
*****************************************************************************/

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;


/* the structure of each input message */
class InputMessage
{
    public String command = new String("");
    public String robot = new String("");
    public String shipmentNo = new String("");
    public String sender = new String("");
    public String recipient = new String("");
};

/* the structure of each shipment record */
class OrderStatus
{
    public String num   = new String("");
    public String from  = new String("");
    public String to    = new String("");
    public String state = new String("");
};


/**
* Class OrderGUI
*/
public class OrderGUI extends Thread
{
    private int TOTALRECORDS = 4;      // maximum number of records

    /* names of the clients */
    private String[] client = {"grad", "graph", "inout", "ref", "store"};

    private String        username;    // name of the user
    private OrderGUIFrame frame;       // link to the java frame
    private Connector     connector;   // link to the robot system
    private OrderStatus[] record;      // shipment records
    private int           total;       // numbers of records
```

```
/**
* A function for blocking the execution of the thread for a specific
* amount of time
* @param time time in seconds
*/
private void longWait(int time)
{
    try {
        sleep((long)time);
    }
    catch (InterruptedException e) {
        // do nothing
    }
}

/**
* Constructor of class OrderGUI
* @param name name of the user
* @param host host's name of the robot system
* @param port port number of the robot system
*/
public OrderGUI(String name, String host, int port)
{
    /* initialize all member variables */
    username  = new String(name.toLowerCase());
    frame     = new OrderGUIFrame("Ordering System", name, this);
    connector = new Connector(name, host, port);
    record    = new OrderStatus[TOTALRECORDS];
    for (int i = 0; i < TOTALRECORDS; i++)
        record[i] = new OrderStatus();
    total     = 0;

    /* establish the connection to robot system */
    connector.connectToIndi();
    if (connector.connectedToIndi() == true) {
        System.out.println("Connected to IndiGolog control");
        start();
    }
    else {
        System.out.println("Unable to establish the connection");
        //start();
    }
}

/**
* Update the information of a row on the table
* @param row the row number (0 - total-1)
*/
public void updateFrameTable(int row) {
    frame.setEntryText(row, 0, record[row].num);
    frame.setEntryText(row, 1, record[row].from);
    frame.setEntryText(row, 2, record[row].to);
    frame.setEntryText(row, 3, record[row].state);
}

/**
* It analyzes the input button event and performs the corresponding
* reaction
* @param button the name of the button
*/
public void handleButtonEvent(String button) {
    /* user pressed the exit buttion (open/close the connection) */
    if (button.equals("exit") == true) {
        if (connector.connectedToIndi() == true) {
            connector.disconnectFromIndi();
        }
        else {
            connector.connectToIndi();
            if (connector.connectedToIndi() == true) {
                System.out.println("Connected to IndiGolog control");
                start();
            }
        }
        return;
    }

    /* if there is no connection, do nothing */
    if (connector.connectedToIndi() == false) {
        System.out.println("Not connected to IndiGolog control");
        return;
    }

    /* user pressed one of the ordering buttons */
    System.out.println("button : " + button + " " + username + " " + total);
    record[total].from  = new String(username);
    record[total].to    = new String(button);
    record[total].state = new String("requested");

    String message = "order(" + username + "," + button + ")";
    connector.sendMessage(message);
    updateFrameTable(total);
    total++;
    System.out.println("To indi : " + message);
}

/**
* Parse the input message and save the information in output
* The input message should be in the form :
*   <type>(<robot>,<sno>,<sender>,<recipient>)
* @param input input message
* @param output an InputMessage structure for holding the parsed message
*/
public boolean parseInputMessage(String input, InputMessage output) {
    int pos1 = input.indexOf('(');
    if (pos1 == -1)
        return false;
    output.command = input.substring(0, pos1);

    int pos2 = input.indexOf(',', pos1+1);
    if (pos2 == -1)
        return false;
    output.robot = input.substring(pos1+1, pos2);

    int pos3 = input.indexOf(',', pos2+1);
    if (pos3 == -1)
        return false;
    output.shipmentNo = input.substring(pos2+1, pos3);

    int pos4 = input.indexOf(',', pos3+1);
    if (pos4 == -1)
```

```
        return false;
    output.sender = input.substring(pos3+1, pos4);

    int pos5 = input.indexOf(')', pos4+1);
    if (pos5 == -1)
        return false;
    output.recipient = input.substring(pos4+1, pos5);

    return true;
}

/**
* It analyzes the input message and performs the corresponding reaction
* @param message an input message
*/
public void handleInputMessage(String message)
{
    if (message == null)
        return;

    System.out.println("From Indi : " + message);

    /* parse the input message */
    InputMessage tokens = new InputMessage();
    if (parseInputMessage(message, tokens) == false)
        return;

    /* follow the instruction of the message */
    if (tokens.command.equals("ackShipment") == true) {
        for (int j = 0; j < total; j++) {
            if ((record[j].from.equals(tokens.sender) == true) &&
                (record[j].to.equals(tokens.recipient) == true) &&
                (record[j].state.equals("requested") == true)) {
                record[j].num   = new String(tokens.shipmentNo);
                record[j].state = new String("serving by " + tokens.robot);
                updateFrameTable(j);
                return;
            }
        }
    }
    else if (tokens.command.equals("rejectShipment") == true) {
        for (int j = 0; j < total; j++) {
            if ((record[j].from.equals(tokens.sender) == true) &&
                (record[j].to.equals(tokens.recipient) == true) &&
                (record[j].state.equals("requested") == true)) {
                record[j].state = new String("rejected");
                updateFrameTable(j);
                return;
            }
        }
    }
    else if (tokens.command.equals("shipmentDelivered") == true) {
        for (int j = 0; j < total; j++) {
            if ((record[j].num.equals(tokens.shipmentNo) == true) &&
                (record[j].from.equals(tokens.sender) == true) &&
                (record[j].to.equals(tokens.recipient) == true)) {
                record[j].state = new String("delivered");
                updateFrameTable(j);
                return;
            }
        }
    }
}

/**
* A thread for receiving messages from the IndiGolog program during
* the execution of the whole application
*/
public void run() {
    String message;

    while (connector.connectedToIndi() == true) {
        /* try to get the next message */
        message = connector.receiveMessage();
        handleInputMessage(message);

        /* wait for a while before accepting the next message */
        longWait(10);
    }
}
```

```
    /**
    * Main function
    */
    public static void main(String[] s)
    {
        if (s.length != 3) {
            System.out.println("Usage: java OrderGUI <client> <host> <port>");
            return;
        }

        OrderGUI control = new OrderGUI(s[0], s[1],
                                        Integer.valueOf(s[2]).intValue());
    }
};


::::::::::::::
OrderGUIFrame.java
::::::::::::::

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;


/*****************************************************************************
* Class OrderGUIFrame                                                       *
* A graphical user panel of the IndiGolog shipment delivery robot control.  *
* Written by Ho Ng (March 29, 2000)                                         *
****************************************************************************/
public class OrderGUIFrame extends JFrame implements ActionListener
{
    final int TOTALROWS = 4;
    final int TOTALCOLS = 4;

    private String[] columnNames = {"Shipment #", "Sender", "Recipient",
                                    "Status"};
```

```
private String[] buttonText = {"grad", "graph", "inout", "ref", "store",
                              "exit"};

private JPanel      upanel, dpanel;
private JLabel      label;
private String[][]  entry;
private JTable      table;
private JScrollPane scrollPane;
private JButton[]   button = new JButton[buttonText.length];

private OrderGUI    handler;


/**
* Constructor of class OrderGUIFrame
*/
public OrderGUIFrame(String title, String username, OrderGUI gui) {
    super(title);

    handler = gui;

    // upper panel
    upanel = new JPanel();
    upanel.setOpaque(true);
    upanel.setPreferredSize(new Dimension(500,30));
    label = new JLabel(username + "'s control panel");
    label.setOpaque(true);
    label.setHorizontalAlignment(SwingConstants.CENTER);
    upanel.add(label);

    // middle panel
    entry = new String[TOTALROWS][];
    for (int i = 0; i < TOTALROWS; i++)
        entry[i] = new String[TOTALCOLS];

    table = new JTable(entry, columnNames);
    table.setPreferredScrollableViewportSize(new Dimension(500, 70));
    scrollPane = new JScrollPane(table);

    // lower panel
    dpanel = new JPanel();
    dpanel.setLayout(new GridLayout(0,buttonText.length,0,0));
    dpanel.setOpaque(true);
    dpanel.setPreferredSize(new Dimension(500,50));

    for (int i = 0; i < buttonText.length; i++) {
        button[i] = new JButton(buttonText[i]);
        button[i].setActionCommand(buttonText[i]);
        button[i].addActionListener(this);
        dpanel.add(button[i]);
    }

    // add all panels to the frame
    getContentPane().add(upanel, BorderLayout.NORTH);
    getContentPane().add(scrollPane, BorderLayout.CENTER);
    getContentPane().add(dpanel, BorderLayout.SOUTH);

    // make it visible
    pack();
    setVisible(true);

    // dumb code to display something on screen
    /*
    table.setValueAt("1", 0, 0);
    table.setValueAt("grad", 0, 1);
    table.setValueAt("reference", 0, 2);
    table.setValueAt("serving by rb1",0,3);
    table.setValueAt("2", 1, 0);
    table.setValueAt("grad", 1, 1);
    table.setValueAt("graphics", 1, 2);
    table.setValueAt("requested",1, 3);
    */
}

/**
* Set the text of a cell on the table
*/
public void setEntryText(int row, int col, String text) {
    table.setValueAt(text, row, col);
}
```

```
    /**
    * It is the button event handler
    */
    public void actionPerformed(ActionEvent e) {
        handler.handleButtonEvent(e.getActionCommand());
    }

    /**
    * A simple test driver
    */
    public static void main(String[] s) {
        OrderGUIFrame frame = new OrderGUIFrame("Ordering System", "test", null);
    }
};
```