An Event-based Robot Control Architecture

by

Jie Yu

B.Sc, Xi'an Jiaotong University, P.R.China, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as confirming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

June 1999

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia

2366 Main Mall

Vancouver, BC, Canada

V6T 1Z4

Date: June 15, 1999

Abstract

The robot control system is a subsystem of a robot designed to regulate its behaviours to meet certain requirements. For a mobile robot performing in a real-world environment, its control system must have the capability to be both deliberative and reactive. In addition to this fundamental requirement, robustness, timely response and coordination among multiple goals are also desirable for a mobile robot control system. This thesis addresses these requirements in building a mobile robot control system and proposes an Event-based Robot Control Architecture (ERA).

The unpredictable nature of the real-world environment leads to the development of our Event-based Robot Control Architecture. All communications inside the control architecture are done in the form of events. Our focus for the mobile robot control architecture is on using an exception mechanism for error recovery. Exceptions are one kind of events. We present a general design and a system prototype for an Event-based Robot Control Architecture. The implementation was used to conduct experiments and identify the benefits and limitations of the proposed paradigm. It was found that a mobile robot using this Event-based Robot Control Architecture meets the previously established requirements.

TABLE OF CONTENTS

LIST OF FIGURES

Acknowledgment

I owe Dr. James Little, my supervisor, a great deal of gratitude for the completion of this work. The time he committed to me and his genuine interest in my work helped me greatly. I am especially grateful for his encouragements that brought me to the world of computer vision and mobile robots.

I would also like to thank the reviewing reader, Dr. Alan Mackworth, for taking the time to review this thesis.

I take this opportunity to thank Don Murray, without whom my thesis would not have been done. His great patience and suggestions have always been my hope. Many thanks also go to Cullen Jennings, Rob Barman and Steward Kingdon for their kind help.

This thesis is dedicated to my family. All these could have not been possible without the love and support from them. They have always been there when I was lost in the darkness.

Thanks to everyone who has contributed to make my experience at UBC a very rewarding one. I have certainly learned a lot and have benefited greatly from the entire experience.

Chapter One

Introduction

For the past three decades, researchers have been working on building autonomous mobile robots in real-world environments. Wide varieties of mobile robot applications range from domestic robotics, warehouse management, to space and military purposes. It is generally desirable to have a robot assistant performing tasks such as mail delivery, trash collection or museum guidance. In situations where it would be dangerous or impossible for a human to do the task, mobile robots become the right choice.

This thesis presents a design for a mobile robot control architecture named ERA. ERA stands for Event-based Robot Control Architecture. The primary purpose of the project is to explore a robot control architecture which interleaves planning with reactive characteristic. The emphasis of this thesis is on the design and evaluation of the system in a real-world implementation.

1.1 Robot Control Architecture

In [ZM98], a robotic system is the coupling of a robot to its environment. A robot is an integrated system, with a robot controller embedded in its plant. A robot controller (or control system) is a subsystem of a robot, designed to regulate its behaviour to meet certain requirements. The planner generates plans and the robot control system actually carries out actions in the environment. Since each robotic system has its own requirements, it is quite difficult to apply a generic control system to all robotic systems. Different robot control architectures provide us different designs for a robot controller. In this thesis, robot control architecture is the primary topic. The topic refers to the software and hardware framework for controlling robots, in our case, a mobile robot. Different robot

control architectures will be briefly compared.

As mobile robots are used to accomplish complex tasks in a dynamic environment, a processor running C code to turn motors does not really constitute a control architecture by itself anymore. Both the development of code modules and communication between those modules begin to define the robot control architecture.

1.2 Motivation

This thesis was motivated by the work done at the University of British Columbia, Laboratory for Computational Intelligence on a stereoscopic visually guided mobile robot. Don Murray [Murray97], implemented a mobile robot named *Jose* developed in LCI. The mobile robot, *Jose,* embodies a sophisticated real-time vision system for the control of a responsive mobile robot. Dynamic environments are unpredictable, asynchronous, and require a low latency in response, while visual information processing requires high data-rate communications and significant computation. The aim of *Jose* is to explore the possibility of using the vision information to guide mobile robot's behaviour. The robot's functions that have been completed to date include mapping, navigation, exploration, and simple manipulation.

The original robot system developed in LCI consists of the mobile robot *Jose* and a host *Sol.* The communication between them is through a radio modem. Since the radio modem is much slower than ethernet, efforts were made to reduce the amount of data sent to the host. Image data captured from the cameras were influenced by this intention. Instead of transmitting whole images over from *Jose* to *Sol*, a radial map is built on *Jose* and then sent to *Sol.* Thus, the amount of data to be transmitted is significantly reduced. Detailed information on how to build such a radial map is described in [Murray97]. However this enhancement in communication reduces the flexibility of the whole system. *Jose* can hardly do any other tasks besides navigation.

The goal of this thesis is to add more complexity and flexibility to the existing

robot system. We hope that *Jose* can do more tasks such as tracking a person. To achieve such behaviour as tracking, the most important issue is to add image processing capability to the whole system. In addition, the system should retain the ability to avoid obstacles. This thesis is motivated by the above considerations. Research has been done to build a robot control architecture aimed at solving these problems.

1.3 Thesis Contribution

To provide more complexity and flexibility to the mobile robot system in LCI, one of the possible ways is to extend the current vision system. The vision system is now used to build a 2-D map of the environment for the mobile robot. This 2-D map is actually an integration of different radial maps over time. Extension of this vision system can be done to include direct image understanding from the images that are grabbed from the cameras. Thus, more complex tasks such as looking for a cup can be accomplished. One of the contributions of this thesis is this extension of the vision system.

The primary contribution of this thesis is to build a robot control system. Our mobile robot is supposed to survive in a dynamic real-world environment. This brings up several considerations for designing such a robot control system. First, in order for the mobile robot to accomplish tasks in a dynamic environment, it is very important for the robot to have both deliberative and reactive behaviours. Deliberative behaviours are goal-oriented. With reactive behaviours, the mobile robot can react to unpredictable events during its tasks' execution. Second, the robot must be robust and capable of error recovery. Especially for a mobile robot with more than one sensor, inconsistent or conflicting sensor readings have to be resolved in the robot control system. Finally, timely response is also a major consideration in designing the robot control system. Our attempt to have image understanding in the robot control system requires careful consideration to meet this timely response requirement.

We propose in this thesis an event-based robot control architecture (ERA). The design of the ERA is based on the considerations mentioned above. The real-world is not

static. Any environmental changes may cause failures for the robot's current task. These failures usually are not predictable. Since it is impossible to design a control architecture that can avoid all failures, exceptions are incorporated into this architecture to inform the robot of failures. Thus, failures can be handled in a proper way. The whole system can finally recover from failures.

The thesis has sub-emphasis on the implementation of systems using asynchronous data flow paradigms instead of procedural decomposition. In addition, coordination among subtasks is resolved by priority based scheduling.

A prototype of ERA design has been implemented on the current mobile robot *Eric* and robot host *Sol*. *Eric* is a cylindrical mobile robot, from Real World Interfaces Ltd. It was built upon the current existing software developed in LCI. Java is chosen as the implementation programming language because it provides a mechanism for handling events. Finally, tracking experiments are performed to test the ERA.

1.4 Thesis outline

Chapter 2 presents a discussion of issues relevant to robot control architecture. It also summarizes several notable papers that present typical robot control architectures.

Chapter 3 first proposes the specifications for the robot control architecture. Some design considerations are also described. Finally an overview of our event-based robot control architecture is presented.

Chapter 4 gives a few detailed design issues. Solutions for these issues are provided as well. Issues discussed are world presentation, virtual robot, sensor fusion, and finally each module is briefly introduced.

Chapter 5 presents a prototype implementation of our robot control architecture. The hardware and software environment of the prototype is introduced. A person tracking

experiment is performed using radial maps to demonstrate our Event-based Robot Control Architecture.

Chapter 6 presents the conclusions inferred from the experiment in Chapter 5, and future work is proposed for improving the system.

Chapter Two

Previous Work

Early robot control systems attempted to plan a complete list of actions in advance of execution. Those plans are generated based only on the collection of the world information. It makes sense to construct a detailed plan well ahead only if the world situation is highly predictable and can be fully controlled. The most typical system of this kind is usually referred as a functional decomposition or centralized system. In 1985, Rodney Brooks[Brooks86] published a report on a completely new robot control architecture called "reactive systems". Rather than attempting to model the world in advance, reactive systems had multiple task modules which are named behaviours, reacting directly to the sensory information.

The limitations of the above two systems are obvious. For the functional decomposition control system, it could only survive in an artificially created domain such as in a laboratory or on a factory floor. However, in a more dynamic world, where actions cannot be anticipated, the situation at execution time cannot be controlled, and a detailed plan cannot be built. The control system proposed by Brooks is a purely reactive system which can only wander around safely but aimlessly. In order for the robot to do some real world tasks, recent discussion on robot control architecture tends to combine planning and reactive behaviours together into one system. Most of the systems start with one approach and try to push their capabilities toward the other one.

In some circumstances, real-world systems not only are complex but also have hard real-time deadlines. This becomes a significant challenge for the AI community. Traditionally, AI techniques aim to provide careful and complete plans for a task. This con-

sideration retains both reactive and unpredictable mechanisms but does not guarantee hard real-time responses. To enable a control system to meet hard deadlines, some of the researchers are trying to embed real time control system into artificial intelligence domain.

The main part of this chapter describes some of the existing approaches to build robot control architecture. Section 2.1 describes the planning control systems. The reactive control systems are presented in Section 2.2. Section 2.3 gives three well-known hybrid robot architectures which have already been implemented and the S* proposal. Finally in Section 2.4, a brief discussion of real-time robot control architectures is presented.

2.1 Planning Control Systems

Planning control systems reason about and plan every action before task execution. These control architectures usually have a sophisticated world model and reason about how to accomplish the task goal based on this world model. The general approach is to sense the world, build a world model, plan actions which aim to achieve the goal, and finally execute the actions via sending commands to robot motor system. Classically the problem of planning control system has been addressed within a framework of functional decomposition into sensing, planning, and acting components (e.g., [Nilsson84], [Moravec83], [Crowley85]). Rodney Brooks [Brooks86] described such a system which decomposed the problem into a series of functional units as illustrated by a series of vertical slices. Figure 1 presents such a decomposition.

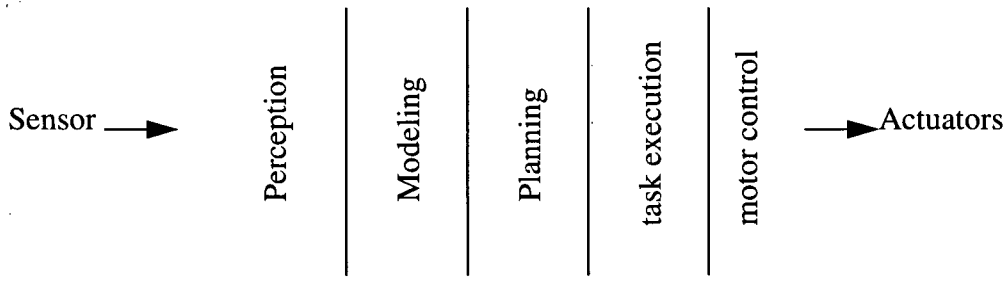

Figure 1: Functional Decomposition of A Mobile Robot Control System

Elaborate reasoning and planning, on one hand, demonstrate a high level of sophistication for robots. But on the other hand, they also require an accurate world model. Dynamic environments and sensor noise always make it unreliable. The planning control system is only applicable in a controlled situation. Additionally, because of elaborate reasoning and planning, the system is generally not fast enough to be used in real-world mobile robots.

2.2 Reactive Control Systems

In response to the lack of flexibility and reaction to the dynamic world existing in a planning control system, attempts were made to completely abandon the planning approach. A behaviour-based control architecture was proposed by Rodney Brooks in 1985. Brooks demonstrated his theory by using the behaviour-based control system on several mobile robots developed at the Artificial Intelligence Laboratory, Massachusetts Institute of Technology [Brooks 86]. This reactive control system provides fast reactions to a dynamically changing environment by short control loops.

The foundation for the reactive control architecture is the idea of "behaviour". As indicated in Section 2.1, a planning control architecture was split into functional tasks namely sensing, world modeling, planning and execution. In contrast to this, the reactive control architecture has multiple independent tasks running in parallel. These independent tasks are called behaviours, for example, avoiding obstacles, moving forward and following a target. Each behaviour decides by itself what is the relevant sensory information and is activated by this specific sensory information. Therefore, behaviours are triggered immediately to react to their environment. Response time is significantly decreased from the time in a functional-oriented architecture.

As Figure 2 indicated, the architecture proposed by Brooks took a radical departure from the traditional functional-oriented control systems. Instead of organizing the system from horizontal components, the behaviour-based control architecture is vertical. Each individual layer has an associated task and issues motor commands independently. In

order to coordinate among these motor commands, high level layers could subsume the lower level ones. Or the conflicts are resolved through the priority of the levels. The capability to override other commands is known as subsumption. Thus a reactive control system is a collection of competing behaviours. In the eye of an observer, it is a coherent pattern of behaviours.

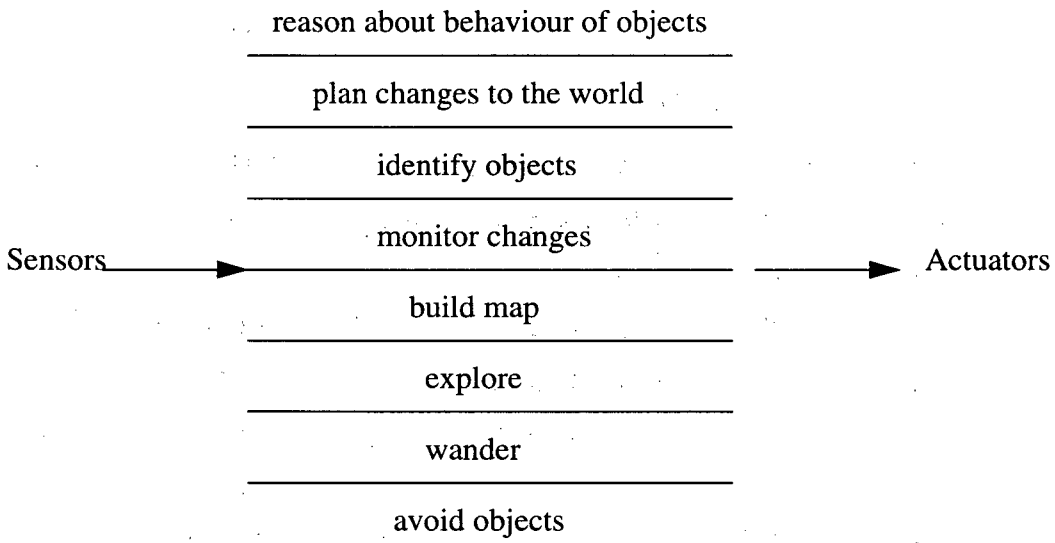

Figure 2: A Decomposition Based on Task Achieving Behaviours

Many systems are implemented based on this reactive control architecture, for example, six-legged walking robots and remote-controlled cars. They demonstrated navigation capabilities that were quicker and more capable than those of planning control systems. However, the architecture is not perfect. Mobile robots implemented in this architecture could only wander around safely but aimlessly. It is difficult to achieve high level interesting performance. Such systems are purely reactive, and lack the ability of sophisticated planning. [Tsotsos98] provides a strong theoretical argument that reactive control systems are only appropriate for a very limited domain of behaviours.

2.3 Hybrid Control Systems

Since the weaknesses of planning control systems and the reactive control systems are apparent, researchers began to move away from purely planning or purely reactive control systems and towards hybrid plan/reactive systems. A hybrid control system allows researchers to explore the benefits of each approach. To demonstrate the possibility and feasibility of such a hybrid architecture, several control architectures such as [Firby87], [Thorpe88], [Simmons90] have been proposed and implemented over the years.

2.3.1 A Blackboard Architecture

The Navlab developed of CMU is a mobile robot using a blackboard architecture as its control system. Figure 3 describes the blackboard control architecture. It was developed as a part of the CODGER (Communication Database with Geometric Reasoning) system[Shafer86].

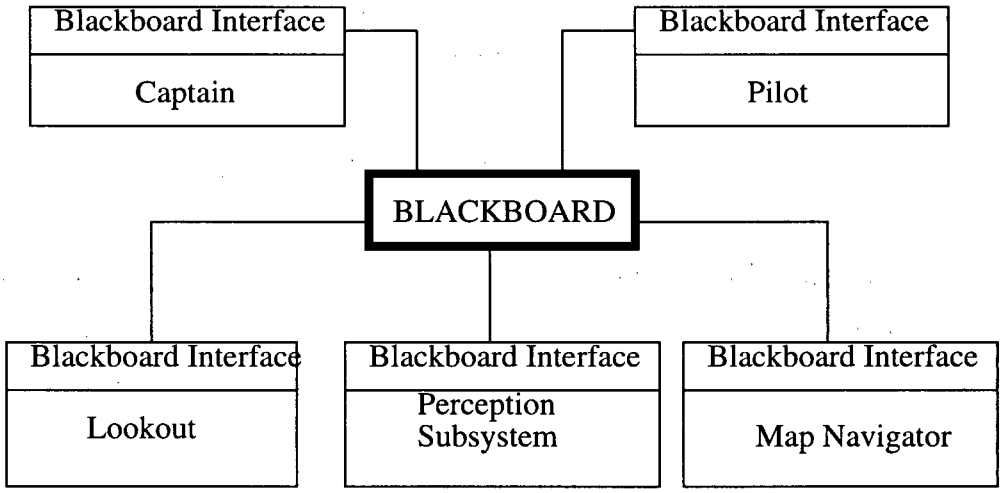

Figure 3: Navlab Robot Control Architecture ([Thorpe88])

The whole system consists of one central database and five modules. The central database is called the Local Map and is managed by Local Map Builder (LMB). The modules are Caption, Pilot, Lookout, Map Navigator, and Perception Subsystem. The

Caption is the overall supervisor for the system. The Pilot is the low level path planner and motor controller. The Lookout monitors the environment for landmarks. The Map Navigator is a high level path planner. The Perception Subsystem accepts the raw input from multiple sensors and integrates them into a coherent representation. In CODGER, each module is a separate, continuously running program. Communication among modules are done by storing and retrieving data in the central database through a set of subroutines called LMB interface. Synchronization is achieved by the facilities provided in LMB interface.

The blackboard control architecture integrates planning and reactive behaviour into one system. Planning was done by Pilot and Map Navigator. Lookout and Perception Subsystem gave the Navlab the ability to react to the outside world.

2.3.2 Reactive Action Package

In the late 80s, R. James Firby of Yale University proposed a concept of reactive planning [Firby87]. In contrast to strategic planning, in which the system is required to look ahead and detect failure situations before they occur, reactive planning systems generate or change their plans only in response to the shifting situation at execution time. All of the planning will take place during execution when the situation can be decided and not on anticipated states.

The whole reactive planning system was built on a mechanism called reactive action packages or RAPs. Each RAP is an independent entity competing with other RAPs and pursuing a planning goal. It will not stop until the goal is achieved or every possibility has been tried. Figure 4 shows a RAP execution environment. The current world representation is stored in the world model. The hardware interface controls the communication between the RAP interpreter and the world model. While each RAP executes, it can change the world representation. This modification is done through the hardware interface. Any hardware information change such as sonar information or visual information will be sent to the world model as well. The RAP interpreter and execution queue main-

tain the relationship between different RAPs.

The execution of the RAP queue is organized in a hierarchical style. Whenever there is a task to achieve, a RAP is chosen to start execution. Each RAP consists of two parts: goal check and task net selector. A task net is a partially ordered network of subtasks. All goal check does is to consult the world model and see whether the task has been achieved. If not, a collection of task nets are selected and inserted into the RAP execution queue. The set of task nets are essential elements to complete the task. These task net elements can be primitive commands or subtasks of the original task. When a primitive command is scheduled and sent to the interpreter, it will be passed onto hardware through the hardware interface. When a subtask is activated, it essentially invokes another RAP. Figure 5 is an illustration of RAP execution.

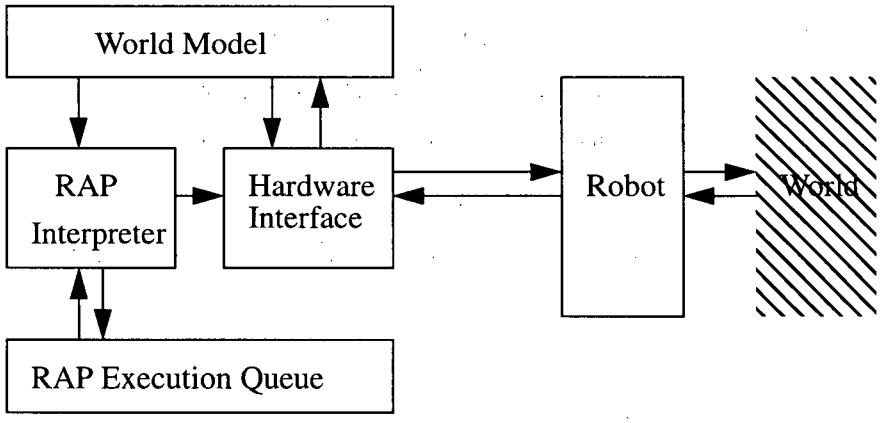

Figure 4: The RAP Execution Environment ([Firby87])

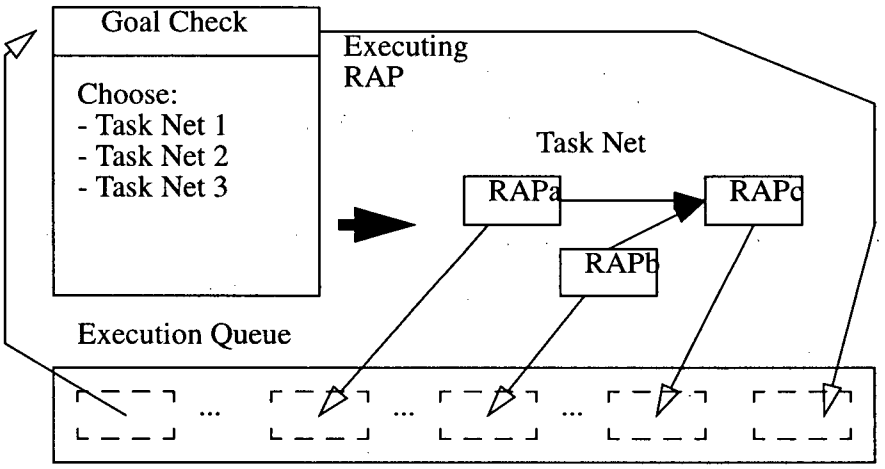

Figure 5: An Illustration of RAP Execution ([Firby87])

The advantage of this reactive planner is its adaptability in an uncertain domain. The shortcoming of the reactive planning system is also obvious. It cannot deal with problems that require thinking ahead. For example, for a robot on a searching crew, its expected behaviour is to bring a flashlight with it since it will get dark outside soon. But for a reactive planning system, it is not going to think about the future at all. Based on the current light condition, it will not bring the flashlight.

2.3.3 Task Control Architecture

Xavier, an office delivery robot developed at CMU, has been in daily use since December 1995 [Simmons97]. This mobile robot has to perform many tasks such as determining the order of the offices to visit, planning path to those offices, following path exactly and avoiding obstacles on its way. Xavier has to deal with incomplete environment information, dynamic situation as well as sensor noise.

A layered architecture is designed for the office delivery robot. The architecture consists of four abstraction layers: Obstacle Avoidance, Navigation, Path Planning, and Task Scheduling. The Obstacle Avoidance module keeps the robot moving in the desired direction without bumping into static or dynamic obstacles. The Navigation mod-

ule follows the paths generated by path planning module. The Path Planning module decides how to travel from one location to another efficiently. The Task Scheduling module determines the order of the offices to be visited. In Xavier, a higher layer works with more abstract representation and provides guidance to lower layers. Each layer is implemented as a separate code process. Interprocess communication and synchronization are provided by TCA (Task Control Architecture).

The Task Control Architecture aimed at exploiting a facility to combine reactivity within a planning framework. The Task Control Architecture was built around the framework of hierarchical task trees. Example of a task tree is shown in Figure 6. A task tree stands for the parent/child relationships between messages. A nonleaf node denotes a subtask or monitor. Leaf nodes are effector commands or queries to read sensors. There are two kinds of temporal constraints between nodes in a task tree: sequential-achievement and delay-planning. For the sequential-achievement constraint, the second task cannot begin until all the leaf nodes of the first task are executed. Delay-planning constraints indicate that the previous task should be completely achieved before the subsequent goal can be handled. TCA also defines facilities to kill subtrees, add new nodes and change temporal constraints.

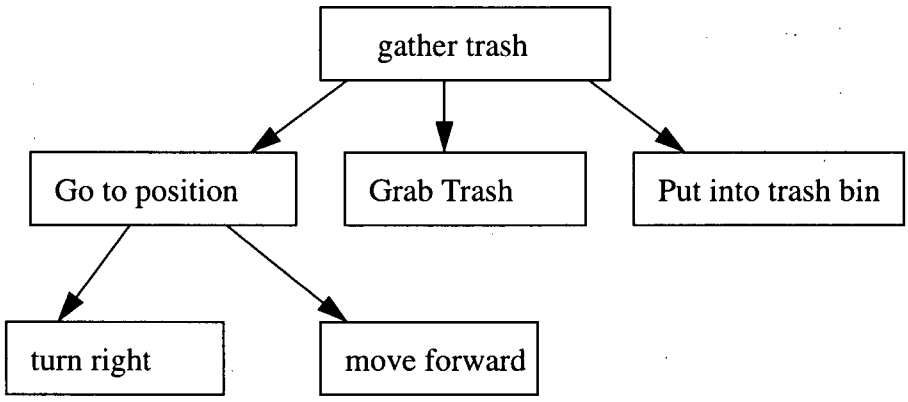

Figure 6: A Task Tree in Task Control Architecture

The Task Control Architecture successfully demonstrated its four main capabilities: interleaving planning and execution, change detection, error recovery, and coordination between multiple tasks. The first capability enables robot systems to act on partically specified plans, allowing them to plan in advance in spite of uncertainty. The other three capabilities enable systems to detect and intelligently handle plan failures, unexpected opportunities and contigencies.

2.3.4 S* Proposal

[Tsotsos98] proposes a control strategy S* which integrates an active vision system. He claims that S* combines deliberative as well as reactive behaviours by using visual attention.

The basic elements in the proposed S* control strategy actually are behaviours. The behaviours act on either an internal representation or an external representation. Each of the representation is a part of the world model. Thus each behaviour in S* is presented as an SMPA-W cycle, in another word, sense-model-plan-action-world cycle. Figure 7 shows this five nodes framework. The world node provides the inputs and outputs for a behaviour.

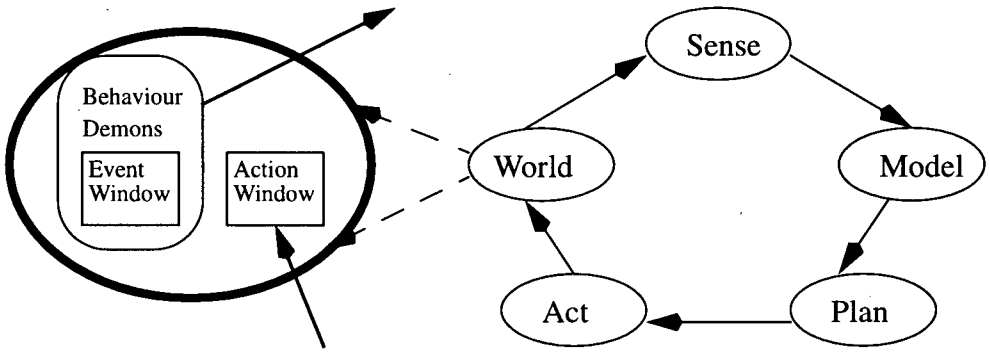

Figure 7: The SMPA-W cycle showing the world node ([Tsotsos98]

To be reactive to a dynamic environment and to recover from a failure, the S* pro-

posal uses exceptions in an active vision task. Failures might happen during the execution of a behaviour. Whenever a failure is detected, an exception will occur and another behaviour would be triggered to handle the failures. Examples of behaviours and their exceptions can be found in [Tsotsos98].

2.4 Real-time Robot Control System

To be reactive is one of the important aspects for a mobile robot to survive in a dynamic, uncertain real world. In addition, we also have to consider how to make a rapid reaction. This requirement is quite obvious. For example, for a completely autonomous vehicle, it must plan new paths and react quickly enough to avoid bumping into people in front of it. This raises another topic in the mobile robot control system-real time control. In AI research, there has always been building systems that are capable of providing solutions as perfectly as it can. This results in a unbounded retrieval and response time which implies unpredictability. On the other hand, in the real-time camp, the goal is to clearly define the resources and capabilities in order to predict the important deadlines and time constraints.

Edmund H. Durfee's definition for real-time control system [Durfee90] is:

> *We mean that a system must carry out its actions before the environment has a chance to change substantially. Put another way, a system must act on its environment more quickly than its environment can unpredictably act on it.*

Traditionally, there are two ways to deal with real time AI systems. One approach is to engineer AI systems to meet real-time deadlines [Laffey88]. The other approach is to tune the AI algorithm so that the plans it developed could meet the desired time constraints [BD89] [Horvitz87]. Durfee presented an alternative approach CIRCA (A cooperative Intelligent Real-time Control Architecture) in [Durfee90]. In CIRCA, real-time and AI components are treated as two separate, concurrent and asynchronous systems. The AI

system could still generate plans that do not satisfy real-time guarantees. Real-time system will ensure those time constraints on its own. This approach provides more flexibility to build a robot control system than the previous two.

In 1998, a group of researchers from NASA Ames Research Center describe a remote agent architecture in [Williams98]. This is a specific autonomous agent architecture which integrates constraint-based temporal planning and scheduling, robust multi-threaded execution, and model-based mode identification and reconfiguration. It also addesses the unique characteristics in a spacecraft domain. Such a spacecraft domain requires highly reliable autonomous operations over a long period of time with tight hard deadlines and resource constraints. The real time control is accomplished by using a temporal Planner/Scheduler (PS), with an associated mission manager (MM) which manages resources and develops plans that achieve goals in a timely manner.

2.5 Summary

We went through a brief history for the development of a robot control architecture in this chapter. Advantages and disadvantages of each system were discussed.

Early functional decomposition control architecture is inappropriate for mobile robots because it is not capable of reacting to a dynamic real-world environment. To overcome its problem, the classical purely reactive control system was proposed by Brooks in 1985. This reactive control system is based on the idea of "behaviours". Each behaviour reacts to the environment very well, but the robot using this control system lacks the ability of planning.

Researchers have been working on building hybrid systems which combine planning with reactive behaviours. Three well-known control systems are CMU's Navlab blackboard architecture, Firby's reactive action package, and CMU's Xavier's task control architecture. To integrate active vision systems with robot control systems, Tsotsos presented a S* proposal which currently does not have any implementation.

Finally, CIRCA and a remote agent architecture were introduced to explore a robot control system which at the same time has time critical requirements.

Chapter Three

System Architecture

The prime goal of this thesis is to explore the design and implementation of a robot control architecture that is capable of detecting and responding to a changing environment. This chapter presents the specifications of the required system, a discussion of important design considerations, and an overview of the final design. The detailed design issues will be presented in the next chapter.

3.1 System Specifications

Since the real world is a complex, dynamic environment for mobile robots, it is useful for us to list some requirements of a mobile robot. These requirements are the primary considerations in the design and implementation of a robot control architecture. Meeting those requirements is important to ensure that mobile robots will survive in a world full of tigers.

3.1.1 Robustness & Error Recovery

The mobile robot should be robust. It should not fail because of some minor changes in its environment. It should have some facilities to deal with unexpected events as well. When the environment changes, it should still be able to achieve some reasonable behavior and function well enough, rather than just wandering around aimlessly.

Inconsistent information from multiple sensors has been one of the problems challenging the robustness of a mobile robot. In the past, most typical mobile robots contain one visual sensor and one simple motion encoding mechanism. Planning is done off-line and the control system is straightforward. Those old-fashioned architectural features can-

not support perception and control in complex dynamic real world environments with very general task specifications. In more complex systems, there is a need to use multiple sensors such as a camera, a laser range finder, and sonar array together at the same time. Only by using multiple sensors, can a system provide a map of its environment with sufficient resolution and reliability to control a mobile robot on a complex mission. For example, a task might require avoiding obstacles along the way, which is best performed with a sensor such as sonar array; the same task might also require tracing a target such as a person wearing a T-shirt of a certain color, which is beyond the effective range of a sonar array yet is easily detected by a color camera. This type of trade-off occurs at all scales of perception, and the only solution currently available is to incorporate multiple sensors on a single mobile robot. As soon as multiple sensors are deployed, the system architecture requirements become very demanding. Sensor data from different times has to be integrated into a single coherent interpretation. Otherwise, the robot might be confused by the inconsistent information and fail to function accordingly.

When contradictory sensor readings occur, the control system should be able to detect an invalid world representation. Error recovery strategies can be employed to change the plan to reflect the current real world situation. In another word, the robot can act even when presented with incomplete and unreliable information.

3.1.2 Deliberative & Reactive

The architecture must accommodate deliberative and reactive behavior. In a dynamic real-world environment, all of the circumstances of the robot's operations can never be fully predicated. Thus, it is infeasible to have a complete course of action in advance. The robot has to coordinate the actions it deliberately undertakes to achieve its designated objective with the reactions forced on it by the environment. As an example, deliberative behaviour of an office mail delivery robot is to deliver a letter to Professor A's office. While on its way to that office, it has to react to avoid obstacles such as students walking around in the building. It should not bump into any obstacle on its way. More importantly, it should not lose its target, in this case Professor A's office, after giving way

to those students. This "postman" should be able to find a path to Professor A's office.

3.1.3 Timely Response

To survive in a highly dynamic environment, timely response and actions are necessary. For a soccer player mobile robot, it would be nonsense if its response time is 10 seconds. The soccer ball would have already moved when it tries to kick it. The time available for it to make a decision is limited. A mobile robot has to operate at the pace of its environment. Speeding up the hardware could make this issue less critical, but current technology still cannot meet our expectations. For robotics, the current solution is to use a compact, real-time operating system with high level programming languages, used to program a control architecture capable of dealing with interrupts from the outside world.

3.1.4 Coordinate Among Multiple Goals, Competing Activities and Functions

Often the robot will have multiple goals that it is trying to achieve. Some of the goals may conflict with each other. In some situations, some of the actions can be carried out concurrently while others have to carried out in sequence. A mobile robot might be trying to reach a certain point ahead of it while avoiding local obstacles. The control system must be responsive to both of the goals. In other cases, when it is impossible for the control system to ensure both of the goal at the same time, it should have some principles to decide which one has to achieve first.

3.1.5 Extensibility

The architecture must give certain flexibility to the designer to develop new features. Application development for mobile robots frequently requires experimentation and re-configuration. Changes in the task may also result in system modification. Easy extensibility is one of the considerations in the design and implementation of the robot control architecture.

3.2 Design Issues

Many robot control architectures have been developed in the AI community especially for indoor and outdoor mobile robots. Implementation of those control architectures usually involves a large amount of code. In addition, researchers in the mobile robot area tend to implement first a prototype, and then gradually evolve it into a complete system. These considerations lead to the following principles in the design of our system.

3.2.1 Modularity

Modularity is very important in current software engineering. A system is broken down into several components, each of which has a particular task. One of the goals for modularity is to reduce the duplicate knowledge among different components. For instance, rectified images from cameras are only necessary for a tracking module, but not for path planning module. In this case, there is absolutely no need to inform the path planning module of the newly arrived images.

Another critical issue in modularity is how to keep the interface between different modules simple. When building a system of many parts, one must pay attention to the interfaces. Poorly designed interfaces will cause heavy communications between modules. This definitely has to be avoided to ensure timely responsiveness for mobile robots. Either the interface needs to be redesigned or the decomposition of the components of the system needs redoing whenever a particular interface begins to challenge the simplicity of the components.

For our system, modularity provides another convenience. Each component can be programmed in the language which is the most appropriate and efficient. Different modules could also locate on separate processors. Therefore, parallel processing is possible and helps increase the computational ability of the whole system.

3.2.2 Virtual Robot

The concept of virtual robot is not original. It was put forward by Charles E. Thorpe in [Thorpe88]. They suggested that the details of the vehicles should be hidden. Under this consideration, "virtual robot" was proposed to hide the details of sensing and motion of the vehicle. It is actually an interface between the control system and the physical vehicle.

Our consideration of a simple interface for control of the mobile robot incorporates the idea of a virtual robot. In this way, the mobile robot will start moving forward only by a simple command such as "forward". The high-level programmers do not have to know every detail such as the trajectory to tell the robot to move.

3.2.3 Distributed Control

Early control systems expected a central module called master to know everything about how to make things work. This traditional control method is usually called centralized control in early blackboard architectures. The master module is in charge of scheduling other modules. It knows exactly when and how to execute each of the other functional components. The problem for this centralized control is obvious. As more and more components are involved, this central module could become a bottleneck. Systems built in this pattern will be difficult to expand.

In order to overcome this shortcoming in centralized control, researchers naturally began to think of ways to build a distributed control system. In a distributed control system, each individual module becomes an autonomous component. It decides by itself when to start or finish execution; however, proper inputs and outputs for each module must be carefully defined. Input for one module could be the output from another module.

Distributed Control gives the system flexibility and solves the bottleneck problem, but on the other hand, it brings forward other issues such as communication and synchronization between relevant modules.

3.2.4 Effective Communication

Inevitably, communication is one of the most significant issues resulting from distributed control architecture. To communicate effectively, it is worthwhile to find a way to exchange information as fast as it can. Communication could take place either within multiple processes on one machine or among processes which reside on different machines.

One of the issues in the design of our robot control system is how to coordinate between low bandwidth and high bandwidth sensor readings. This problem arose when using radio modem to communicate between the physical robot and the host. High bandwidth communication is usually involved in passing a large amount of data from the robot to the host or around in the control system. Rectified images and disparity images are examples of high bandwidth sensor readings. There is no such heavy burden on radio modem about the communication and computation time for low bandwidth sensor readings. For example, power supply for the mobile robot could be expressed using an 8 bit integer. Since transmitting high bandwidth sensor readings requires much more time than transmitting low bandwidth readings, the high bandwidth data will block the low bandwidth data without any special consideration. We need to find a way to transmit the low bandwidth data in between the high bandwidth ones. Our solution for this problem is quite simple. The high bandwidth data will be divided into several segments. Each segment will be sent separately. Therefore, it gives some time gap between two consecutive parts and this time gap can be used to send low bandwidth sensor readings.

Our consideration to divide the whole system into several modules brought forward another issue --- communication among modules on the same machine. The traditional solution for communication among processes on the same machine includes message queue, shared memory, or message passing. However, our design consideration is moving toward an event driven system. All of the messages passing within the system are in the form of events.

3.2.5 Synchronization

Two issues are involved in synchronization. One is how to synchronize the execution of multiple modules. There are times when one module has to wait for another to finish. This can be solved by using signal and wait mechanism. The other issue is how to synchronize the simultaneous access to a database from multiple threads of execution. The most common solution for this is to add a lock-step before and an unlock-step after access to the database.

3.3 A Few Definitions

To avoid any ambiguity which may occur in the following description, we provide some definitions first.

World Representation: This is where world information is stored. Not only external information but also information about robot itself are included in this world representation. From here, a mobile robot gets to know its environment and acts accordingly. A model of the environment is built and stored in this world representation.

Behaviour: The idea of a behaviour-based robot first appeared in [Brooks86]. Each behaviour is responsible for a particular goal such as avoid obstacle, follow wall and move forward. A set of related behaviours running concurrently and coordinating together result in the completion of certain task such as tracking a target. Originally, behaviour is used in a layered architecture. Each layer has one or more behaviours running concurrently.

Tasks and Subtasks: A task is what the mobile robot is trying to accomplish. Examples are person tracking, mail delivery and trash collection. A task could be divided into several parts. These parts are called subtasks. For example, a person tracking task consists of two steps (subtasks). They are looking for the person and visually focusing on the person respectively.

Plans: Plans are generated by the planner using various AI techniques. Plans are a set of subtasks aimed to accomplish a certain task.

Atomic action: An atomic action is a primitive operation on the world presentation by effector. No further decomposition of the action can be performed. It will result in the change of the internal or external representation. Telling the robot to go forward, backward, turn left or turn right are examples for atomic actions. Behaviours consists of a sequence of atomic actions.

3.4 System Overview

The goal of our control architecture design is to have a mobile robot that can survive in a real world environment and accomplish certain tasks. Since a mobile robot's environment is no longer static, or artificial, interaction between the mobile robot itself and the environment must be carefully investigated. In Section 3.1, we provide a brief discussion of these interactions while presenting the first two specifications. To let the control system have the ability to be reactive to the dynamic world and recover from error, exceptions are incorporated into the system as our fundamental mechanism. This section first presents the exceptions and exception handlers. The exception mechanism is that a part of the system throws an exception to be caught by an exception handler. An overview of the ERA (Event-based Robot Control Architecture) is presented as well. A comparison between our exception mechanism with [Tsotsos98] comes later in this section.

3.4.1 Exception Mechanism

The world representations store the internal and external information for a mobile robot. During the execution of a task, these world representations are continuously updated to reflect the robot's environment at that time. The mobile robot draws conclusion about its environment and carries out tasks based on these world representations. Even a trivial change of the environment may have a great impact on the robot's executing task. Some of the changes may result in the failure of the current executing task. An

exception occurs whenever a failure is detected by the robot. For example, for a visual tracking task, a failure takes place when the object moves out of the robot's viewpoint. This failure fires an exception.

The exception mechanism consists of two parts, because the robot control system should be capable of not only discovering a failure but also recovering from it. These two parts are the exception itself and the exception handlers respectively. Each exception has a corresponding exception handler. The exception handlers are responsible for recovering from failures. To recover from a failure in our robot control domain is to generate a set of new plans which still aim to achieve the current task goal under the new circumstance. The relationship between exceptions and exception handlers is depicted in Figure 8.

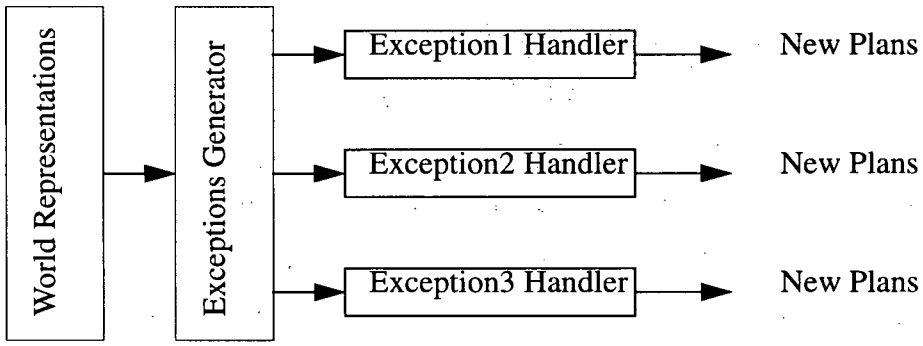

Figure 8: Exception Mechanism

3.4.2 Using Exceptions in an Active Visual Task

One interesting task for a mobile robot to accomplish is to play *Find and Follow* with human. As an example, we explain our exception mechanism used in such a task. A *Find and Follow* task usually is composed of several subtasks such as "searching for the person" and "following the person". At one time, there is only one subtask being executed. During the execution of a subtask, exceptions might occur. After a specific exception is fired, its handler is supposed to do replanning and another subtask will finally be triggered into action.

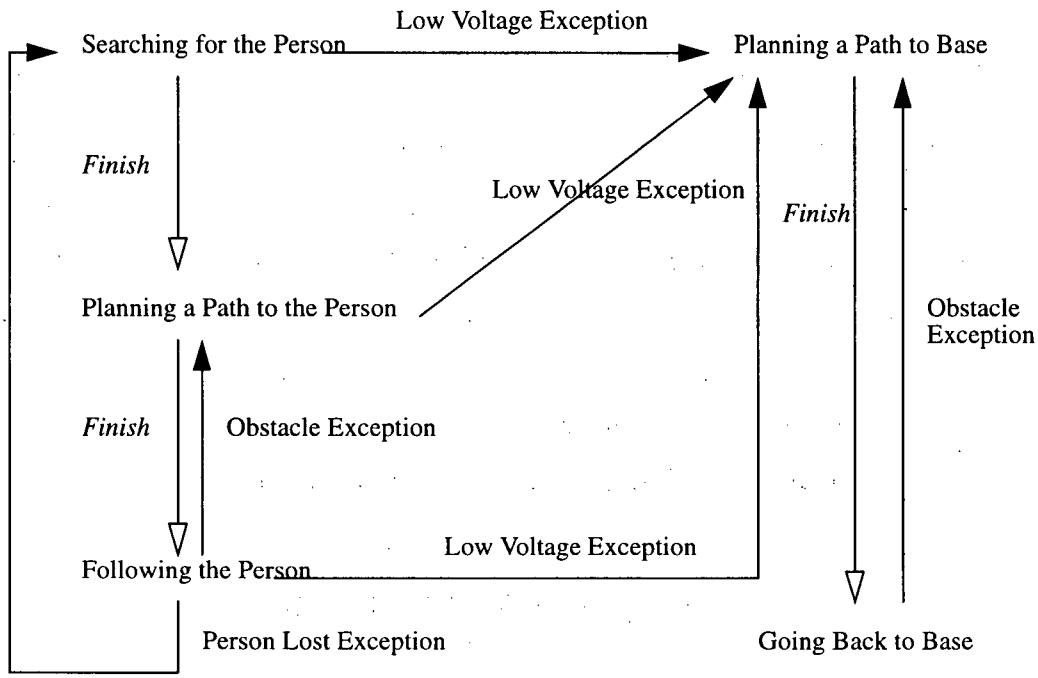

Figure 9: Exceptions and Subtask Transitions

Figure 9 shows the exceptions and subtask transitions in a *Find and Follow* task. The subtasks are "searching for the person", "planning a path to the person", "going to the person", "planning a path to base", and actually "going back to the base". As the figure indicates, for the "going to the person" subtask, there are three possible exceptions. They are *Obstacle Exception*, *Person Lost Exception*, and *Low Voltage Exception*. If the *Obstacle Exception* is detected, "Planning a Path to the Person" is triggered. If *Low Voltage Exception* is detected, "Planning a Path to Base" is triggered to direct the robot to go back to base to recharge the battery. If *Person Lost Exception* occurs, it will begin to search for the person again. This principle obviously applies to other exceptions in Figure 9.

3.4.3 ERA Overview

This section briefly describes the Event-based Control Architecture as a whole for mobile robots in a dynamic and uncertain environment. Figure 10 presents a robot control

system using the Event-based Robot Control Architecture embedded in a mobile robot. With the help of ERA, a mobile robot is able to explore reliably, and safely in real world environment. The embedded Event-based Control Architecture is capable of reacting to the dynamic environment, recovering from failures, and resolving conflict among subtasks.

Based on the task to achieve, the Planner generates using various AI techniques a sequence of subtasks for the ERA to carry out. A discussion of the Planner is beyond this thesis. We will only focus on the ERA itself. After the ERA receives a set of new subtasks from the Planner, it consults the current world representation, decomposes each subtask into atomic actions, and actually sends commands to the robot hardware via virtual robot interfaces.

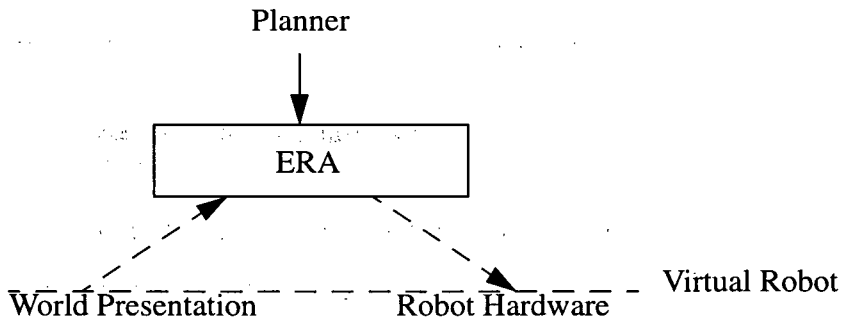

Figure 10: An Embedded Event-based Robot Control Architecture

Figure 11 depicts internal structure for the Event-based Robot Control Architecture. The ERA consists of six modules. They are Sensor, Effector, Monitor, Executor, Exception Manager and Scheduler.

The Event-based Robot Control Architecture is based on the exception mechanism described earlier. The Monitor keeps watch on the sensor readings. When it detects a failure, it fires an exception. There are several exception handlers registered in the Exception Manager module. Each one of them handles one specific exception from the Monitor. The Exception Manager is responsible for handling the exceptions and generating new

plans based on the current task and situation.

Therefore, there are two sources generating new plans. One is from the Planner, the other one is from the Exception Manager. Each of the plans contains a sequence of subtasks. All of the subtasks are inputs for Scheduler. To coordinate the subtasks from these two sources, each of the subtasks is assigned a priority. Subtasks with higher priority will be scheduled first. Those subtasks with the same priority are scheduled based on their order.

The communication between the modules is performed in the form of events. One nice aspect of using events instead of message passing is that a module does not have to sit in a loop waiting for information it is expecting. Obviously, sitting in a loop and waiting uses a lot of CPU cycles. By using events, the modules will be activated only when new information arrives.

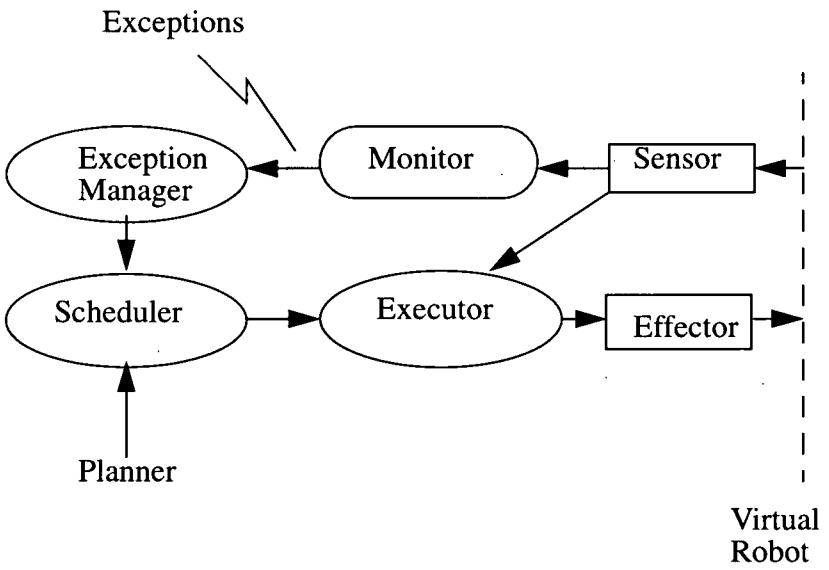

Figure 11: An Event-based Control Architecture

3.4.4 Tsotsos' Active Vision

A brief introduction for Tsotsos's Active Vision has been presented in Section

2.3.4. Here, a further explanation about exceptions in his proposal is presented.

In the proposed S* control architecture, Tsotsos indicates that failures during the execution of a behaivour must be detected. One of the representations in his architecture is Exception Record (ER). This exception record encodes the information to detect failures. Each exception contains a specification of what must be sensed in order to confirm that the exception occurred, the identity of the behaviour for which it occurred and a start time and expiry time. While being executed, a behaviour reads the representations and decides if the current situation is consistent with the description in the exception records. If it is, an exception is triggered and the system should react accordingly.

3.4.5 Comparison between ERA and Tsotsos's Active Vision.

[Tsotsos98] provides a distributed action vision framework using attention. He provided a new point of view for the original functional-oriented decomposition which they call SMPA (sense-model-plan-act). They used this decomposition for every behavior and added the world into this cycle, thus formed a new framework as SMPA-W. In their description, they used exception records to detect failures. The difference between our framework and theirs is that instead of having an exception detector for each behavior, our failure detector and handlers are on a subtask level. Since there are usually multiple behaviors working together in one subtask and each of the behaviours has its own exception detectors, it is highly possible that extra copies of detectors exist in SMPA-W system.

3.5 Summary

The specifications for developing a robot control architecture are discussed first in this chapter. They are robustness and error recovery, deliberative and reactive, timely response, coordination between multiple goals and functions, and extensibility respectively.

Five design issues, namely modularity, virtual robot, distributed control, effective communication and synchronization are also presented. Some of the solutions are pro-

vided.

A system overview indicates that our control system is built upon the exception mechanism. All the communications inside the control system are in the form of events. The exception mechanism is explained and a comparison between our ERA and Tsotsos's S* proposal is made.

Chapter Four

System Details

This chapter provides a detailed description of our Event-based Robot Control Architecture.

4.1 World Representation

The definition of world representation was first given in Section 3.3. In this section, we will further discuss on this topic. Both the internal representation and external representation are presented in the following paragraphs.

4.1.1 Internal Representation

A collection of information supports the internal representations of a mobile robot. The most common information used is:

Robot Position: The current position helps a mobile robot locate itself in its environment. It could be depicted in a triple (X, Y, H). X, Y are the world coordinates for the robot and H describes the orientation of the robot.

Power Status: Its value indicates the voltage level for the robot. Once the power status is below a certain value, the robot should stop and get the battery recharged. This piece of information is nontrivial and useful while implementing a robot control architecture on a mobile robot. For instance, a mobile robot working in deep water should be able to come back when it is about to run out of power. We certainly do not want it to sit in the bottom of the sea waiting for someone else to get it back.

Robot Mode: It would be nice if the mobile robot could either be operated manually

autonomously. The robot mode indicates the situation of the robot. In our system, the robot mode could be one of four choices: manual, explore, directed, or specified.

Robot Command: This stands for the current execution command. This information is necessary when an exception occurs. How to recovery from the failure largely depends on what the robot currently is doing.

4.1.2 External Representation

In order for the mobile robot to complete a task in a dynamic environment, interaction with its environment is inevitable. It is also important for the robot to build a map of its environment. Thus, several facilities are developed as external representation of the world model.

Sonar Array: Sixteen sonar sensors continuously send out sonars and try to protect the robot by detecting any obstacle around. Sonar sensor readings are stored in an array which can be read by robot control system through some interface to the world representation. This interface is called the virtual robot and will be explained later in Section 4.2.

Camera Images: These are the largest pieces of information in the world model. Each image is a two dimensional array capturing the real world ahead of the robot. Each pixel in the image represents intensity. Figure 12 shows a real camera image.

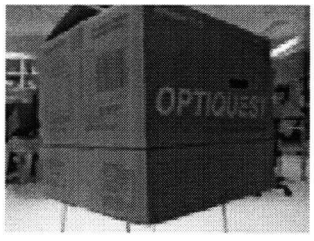

Figure 12: A Real Camera Image

Disparity Image: In addition to the above camera images, there is a disparity image resulting from a stereo vision algorithm. This is also a 160X120 array ([Tucakov97], [Murray97]). Some of the pixels could be invalid information. The value of each valid pixel represents the disparity. The larger the value, the closer the object. Figure 13 shows a disparity image.

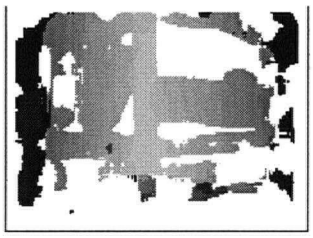

Figure 13: A Disparity Image

Radial Map: Since the disparity image consists of a large amount of data, it would be slow to pass the whole image around in the control system. A radial map is built instead. Details on how to build the radial map could be found in [Murray97]. It is a one dimensional array consisting of 160 elements. Each pixel in the radial map is projected column by column from the disparity map. It takes the maximum valid disparity in each column. Similar to disparity image, objects which are closer to the robot will have larger value in radial map. Figure 14 shows a radial map.

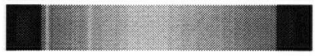

Figure 14: A Radial Map

Maps: There are a total of four maps, namely, zoom map, plan map, obstacle map and distance map. They are gradually built as the robot moves around in its environment. Occupancy grid map reflects the robot's understanding of the world ([ME85], [Elfes89]). The pixels in the map are put into the following three categories. Black pixels in the map means that those places are occupied by obstacles. The robot cannot go to those areas. Clear areas without obstacles are depicted as white pixels. The area the robot has not yet seen and does not know about is

shown as grey. The algorithm to build such a map is described in [Murray97]. Figure 15 depicts a map generated during exploration.

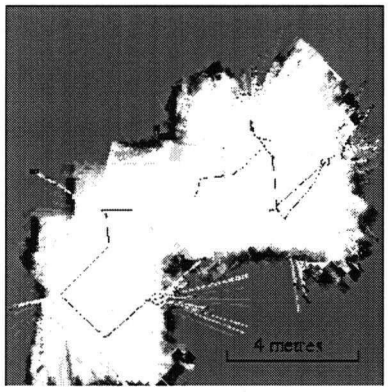

Figure 15: A Map Generated During Exploration

4.2 Virtual Robot

Access to robot control processors and sensor data takes place through a set of subroutines called the virtual robot. These virtual robot subroutines are divided into two groups, one for robot control and the other for retrieving world representation.

4.2.1 Robot Control

Robot control commands are messages sent from the control system to the hardware to control actions of a mobile robot. There are altogether six control commands. The first five commands will lead to either orientation or world coordinates changes

1. forward: Move forward until the control system tells the robot to stop; the orientation of the robot does not change.

2. backward: Move backward until the control system tells the robot to stop; the orientation of the robot does not change.

3. turn left: Turn left until the control system tells the robot to stop; the orientation of the robot changes while world coordinates do not.

4. turn right: Turn right until the control system tells the robot to stop; the orientation of the robot changes while world coordinates do not.

5. turn 360: Turn around for 360 degrees and stop when the orientation of the robot is back to the original position. The world coordinates do not change.

6. halt: Halt the robot immediately.

Controlling the movement of the mobile robot is simply calling these six subroutines. This hides away complicated details of moving the robot around.

4.2.2 World Representation Retrieval

Subroutines are also available to retrieve information from the world representation database such as maps, images, and sonar readings.

4.2.2.1 Internal Representation

Subroutines designed for getting internal representation are simple. All you need are getVoltage/RobotPosition/CurrentCommand/CurrentMode subroutines.

4.2.2.2 External Representation

1. Map

updateZoomMap/updatePlanMap/updateObstacleMap/updateDistanceMap: These subroutines update the control system's copy of the specified map.

2. Image

updateImage: Given the image type which is specified as a parameter, this subroutine updates the control system's copy of the specified image. The image size is 160X120.

updateImageNRows/NCols/RowInc: Given the image type, these subroutines update the number of rows/number of columns/row increment of specified image.

updateImageRows: Give the image type, starting row and number of rows N as parameters, updateImageRows will update the control system's copy of the specified image but only updates N rows from the starting row. This is necessary in order to reduce the time spent on copy image from one block of memory to another. If carefully used, it will help improve the responsiveness of the control system.

3. Radial Map

getRadial: retrieve radial map from database.

4. Sonar Readings

As sonars are used to detect obstacles around the robot, and the robot can only move forward or backward, it is not necessary to pass the whole sonar array within the system. What we want to know are only two values, in another word, whether there is obstacle in front or behind the robot. A notify flag is sufficient to hold this information. A subroutine called getNotifyFlag is designed to interpret the sonar readings.

4.3 Sensor Fusion

As presented in Section 3.1, current mobile robot systems incorporate multiple sensors at the same time to achieve reliable performance. Usually, sonar sensors, laser ranger finder, tactile and cameras will be mounted together on one mobile robot. Therefore, when some objects are beyond one sensor's capability, other sensors could be complementary to it. For example, cameras can detect an object at least half meter away from robot, while sonar is functional to find the object within the range of half meter. Figure 16 shows a system with multiple sensors.

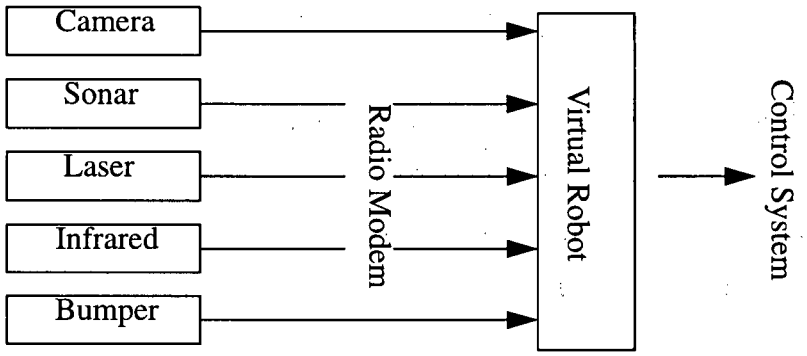

Figure 16: A System with Multiple Sensors

4.3.1 Sensor Fusion Methods

Since there is more than one sensor in one system, the issue of how to coordinate among them is brought up. This issue is called sensor fusion and discussed in [CY90] in detail. Three primary methods for sensor fusion were discussed in [Shafer86].

Competitive: Competitive fusion is typically used for sensors that generate the same type of data such as two sonar sensors. Each of the sonar sensor will produce a hypothesis. Two sensor readings may conflict or reinforce with each other during the process of fusion.

Complementary: Each individual sensor is used differently. The aim of using different kinds of sensors is to enhance the advantages and cover the disadvantages of each of them. The best example of this type is using sonar with camera together as we have already discussed earlier.

Independent: In this method, a sensor is used independently for a specific task. It provides enough information for that task.

When designing our robot control system, we believe all the complex tasks the robot aim to achieve need more than one sensor. Either competitive or complementary sensor fusion method is used in different situations.

4.3.2 Sensor Invocation

In [Shafer86], the strategies used to invoke sensors in sensor fusion are also discussed.

Fixed: This strategy embodied in special-purpose code that specifically invokes sensor processing.

Language-Driven: A strategy defined in a general "perceptual language"; each object in the object will have a description of how to recognize that object.

Adaptive: A language-driven system with the ability to select alternative plans based on the current status of sensor processing or the vehicle's environment.

However, our design involves two ways to invoke sensors. We call them polling and adaptive invocation.

Polling: the sensor module issues the condition queries at a fixed frequency. For example, we use a polling invocation that reads the mobile robot's power supply at a fixed time interval. This time interval can be set dynamically. Whenever the power is below certain threshold, the control system would detect it and replan to recharge the battery.

Adaptive Invocation: In this method, sensors will only be invoked when needed. Sensors are triggered and cancelled on demand. For example, the perception system needs to be directed and controlled by the agent's current action and task. Especially for a sensor which needs high bandwidth communication such as cameras, it is better to turn it on only when necessary.

4.3.3 Selective Attention

When discussing effective communication in Section 3.2.4, we give a solution to

coordinate between high bandwidth and low bandwidth sensor readings in order to provide timely response. In addition to that, selective attention is also an effective way to reduce communication and computation time. Especially for the rectified images and disparity image, usually not all of the 160X120 pixels are needed in the image processing at the same time. Thus retrieving region of interest (ROI) is much more efficient than getting the entire 160X120 images.

4.4 Event-Driven Communication

There are mainly six modules in our Event-based Robot Control Architecture. Effective communication between these components certainly contributes to the feasibility of this control architecture.

The nature of a dynamic and uncertain world is its unpredictability. It is not desirable to have a busy waiting process running in each module waiting for incoming messages. When there is no environmental change for a period of time, using a daemon process would exhaust much of the CPU time. Therefore, an event-driven communication is employed in our control architecture.

4.4.1 Event and Event Handler

Everything passing around within the control architecture is in the form of an event. An event is generated by a sender component, and thrown to a receiver component or multiple components in the system. An event carries the information needed by the receivers such as control command, sonar readings and so on. An event handler is in the receiver component. As the name indicates, its responsibility is to handle a specified event. The components which are expecting an event are the event's listeners. Each sender component maintains a list of event listeners for each event that it will possibly fire. Each receiver component has one or more exception handlers.

4.4.2 Events

All the events are put into four categories according to the type of information they are carrying.

ActionEvent: This kind of events is fired by Executor and handled by Effector. Its information is the atomic action such as moving forward and backward which is to be carried out on the physical mobile robot.

SubtaskEvent: SubtaskEvent is fired by Scheduler and handled by Exception Manager. It includes the information about the current executing subtask of the robot. Exception Manager uses this piece of information to decide what kinds of exception it is expecting.

DataEvent: World representation of the mobile robot is wrapped in this event. Data in DataEvent could be sonar readings, voltage, radial map, or even images.

ExceptionEvent: Each exception is also an event in our ERA. It is fired by Monitor and handled by Exception Manager.

Events can also be categorized as unicast events and multicast events. Figure 17 shows a unicast event. Figure 18 shows a multicast event.

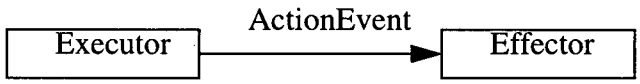

Figure 17: A Unicast Event

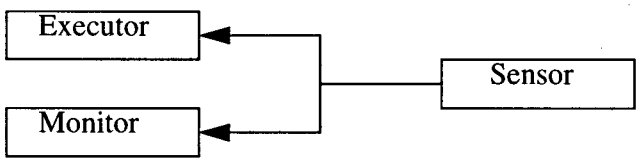

Figure 18: A Multicast Event

If there is only one event listener registered for an event, this event is a unicast event. If there is more than one event listener registered for the event, the event is a multicast one.

4.5 Modules

In this section, each of the modules introduced in Section 3.4.3 is explained in detail. As we mentioned in Section 4.4, communication between these modules is in the form of events.

4.5.1 Effector

The Effector has almost no intelligence in it. Its main job is to perform atomic actions. Those atomic actions are received as action events from the Executor. The Effector sends commands to the robot hardware via the virtual robot interface. The Effector spends most of its time in sleep mode. It only wakes up when an action event arrives. After the atomic actions are carried out, the world representation, either external or internal, will be modified to keep it consistent with the real world.

4.5.2 Sensor

The Sensor is responsible for reading world representations and forwarding this information to the Monitor. There are two ways to invoke the sensor readings, namely polling and adaptive invocation. Therefore, there is always a daemon running inside the Sensor. It reads world representation at fixed frequency. The other part of the Sensor retrieves information from the world representation only on demand. The sensor readings are sent to the Monitor by *DataEvent*. However, not all of the readings are sent to the Monitor. Given the current subtask being executed, only those that the Monitor is interested in and has been waiting for will be sent. In the case of safe navigation task, preventing the mobile robot from bumping into obstacles is indispensable. During navigation, the pieces of information that the Monitor is interested in are sonar map and radial map. They will be sent to the Monitor as events. Other information such as the disparity image is not

necessary for the navigation task, thus it will not be sent over to the Monitor.

4.5.3 Monitor

The Monitor module is an essential part of the active control system. This is where any failure is detected and where the exceptions occur. Whenever a new task or subtask is executing, it dynamically registers its intention to the Sensor. The Monitor is activated only when *DataEvents* come in. After its activation, it processes the data and fires an exception to the Exception Manager if any failure is detected. For instance, for a visual orienting task, the Monitor receives images as incoming events. It generates an exception when the object moves out of its view area.

4.5.4 Exception Manager

In fact, the Exception Manager is a goal-oriented replanning mechanism. Its inputs are the exceptions generated by the Monitor. Its output is a set of new subtasks. These new subtasks are generated based on the current world representation and the current designated task. The replanning in the Exception Manager makes the system reactive to environmental changes. For example, while the mobile robot is on its way to an office, an exception occurs indicating that there is an obstacle ahead. After receiving the exception from the Monitor, the Exception Manager generates a new path leading the robot to the previously given office.

4.5.5 Scheduler

This part of the system handles the coordination, scheduling and arbitration among competing subtasks. These competing subtasks may be derived solely from the Planner or may come from both the Planner and the Exception Manager. An example in terms of scheduling is as follows. The Scheduler receives "visually pursue moving object" from Planner and at the same time, gets a subtask from Exception Manager telling it to go back to the starting point because of low battery warning. In this situation, the Scheduler has to decide which one will be executed.

The subtasks are basically scheduled based on first come first serve principle. But some of the subtasks might be more urgent than others. For example, usually those coming from the Exception Manager have to be scheduled first to avoid fatal errors. This coordination among subtasks can be realized using priority-based method. A priority is attached with every subtask. Those with higher priority will be scheduled first. Those with the same priority, subtask will be scheduled according to their sequence.

4.5.6 Executor

The Executor is where the subtasks are actually carried out. Concurrent execution of subtasks would need more than one executors. It is activated when a subtask is scheduled to be executed. After being triggered, the Executor parses subtasks into atomic actions and sends those atomic actions to effector.

4.7 Summary

A few issues are discussed in the previous sections. Internal and external world representation are described in the first section. A virtual robot is designed to abstact away the hardware details. Robot control can be achieved by control subroutines. World representations can be retrieved by a set of retrieving subroutines. Three issues are presented and solutions for them are provided in the section of sensor fusion. An event-driven communication mechanism is introduced in Section 4.4. Finally, each of the six modules in our ERA is discussed in detail.

Chapter Five

Prototype Implementation

The prototype described here is produced as proof-of-concept demonstration of the ERA's feasibility. The programming goal of the implementation is to conduct experiments and identify the benefits and limitations of the proposed paradigm. Experiments are described later in this chapter.

5.1 Implementation Environment

This section introduces the hardware and the software environment in which a prototype of an Event-based Robot Control Architecture is implemented.

5.1.1. Hardware

We use a Real World Interfaces (RWI) B-14 mobile robot, *Eric*, to conduct our experiments for our Event-based Robot Control Architecture. *Eric* is equipped with a Pentium^{TM} PC running the Linux operating system as its onboard processing. It also has an Aironet ethernet radio modem that allows communication to a host computer *Sol*. A Triclops trinocular stereo vision camera is mounted on top of the mobile robot. Figure 19 shows our mobile robot *Eric*.

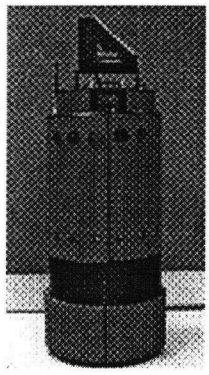

Figure 19: A Mobile Robot *Eric*

Active Vision of our system is achieved by using the Triclops stereo vision camera [Murray98]. Figure 20 illustrates a Triclops stereo head. The Triclops stereo vision module was developed at the UBC Laboratory for Computational Intelligence (LCI) and is being marketed by Point Grey Research, Inc (www.ptgrey.com). The stereo vision module has 3 identical wide angle (90 degrees field of view) cameras. The system is calibrated using Tsai's approach [Tsai87]. Correction for lens distortion, as well as misalignment of the cameras, is performed in software to yield three corrected images. These corrected images conform to a pinhole camera model with square pixels. The camera coordinate frames are co-planar and aligned so that the epipolar lines of the camera pairs lie along the rows and columns of the images.

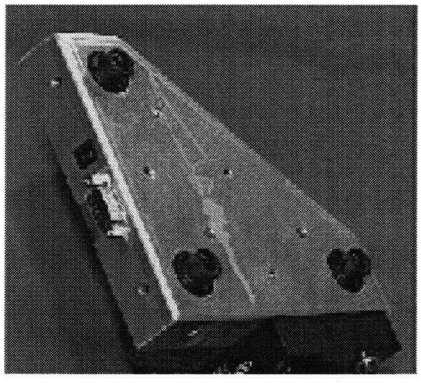

Figure 20: The triclops stereo head

To easily debug the system and monitor the progress during the experiment, we implement most of the control system on our host machine *Sol*. The host *Sol* is a also Pentium^{TM} PC with Linux operating system installed.

5.1.2 Software Environment

The prototype of the robot control system on *Eric* is not implemented from scratch, but based on existing software modules developed by Don Murray, LCI. This section gives a brief introduction for the previous software architecture and its important modules.

5.1.2.1 Software Architecture

Figure 21 presents the software architecture. The dashed line in the figure represents the physical separation between the robot and the host. Communication between these two parts is implemented by sockets.

The software implemented on the robot is in charge of sensing and control. The RadialServer continuously grabs images from the triclops camera, generates radial maps and stores the radial maps into shared memory. The RobotServer, on the one hand, receives commands from the host and sends them to the robot motor, and on the other hand, collects information about the robot such as radial maps and sends it to the host. The software implemented on the host does data integration, reasoning, and interacts with a human operator. As the figure indicates, all the modules on the host exchange their knowledge through the shared memory.

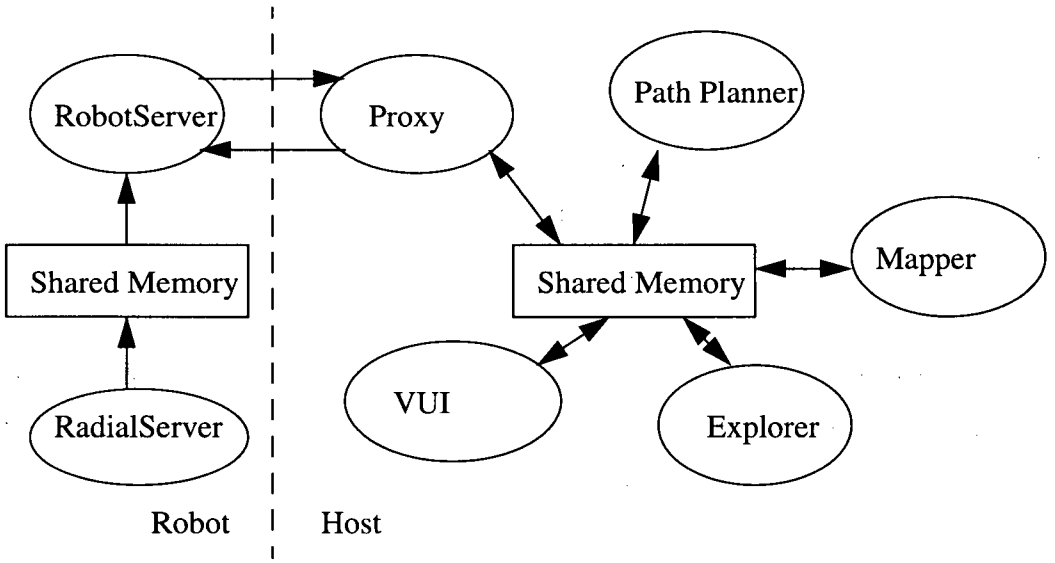

Figure 21: Previous Software Architecture

5.1.2.2 Important Modules

1. Mapper

The Mapper integrates radial maps over time into a 2-D map represented by an

occupancy grid. The value of each grid is related to the probability that this space is occupied by any part of an obstacle. The Mapper initializes the map to contain only values at 50% probability, indicating that the entire space is unknown. As new radial maps arrive, the Mapper updates the occupancy grid so that each cell contains an updated probability that the cell is occupied by an object. Every point between the current position of the robot and the nearest obstacle in a given direction is marked clear. Cells beyond the object detected are unaffected.

2. Path Planner

The Path Planner produces paths for the robot to follow. These paths are used to move the robot from one position to another. The inputs for the Path Planner are a map of the environment produced by the Mapper, a goal position, and an initial position. The output for the Path Planner is a sequence of significant waypoints along the path. The path is generated by a simple wavefront expansion algorithm. This algorithm is briefly described in [Murray98].

3. Explorer

The grid locations are classified into three basic types: blocked, clear and unknown. The goal of Explorer is to reduce all unknown regions until all reachable areas are either clear or blocked. This was implemented by re-using our path planning algorithm. The robot will be guided to the nearest accessible unknown region. With periodic reevaluation and re-planning, the robot will explore from unknown region to unknown region until no more unknown regions are reachable.

5.2 Implementation

5.2.1 Software Architecture

With the robot control system, the software architecture is described in Figure 22. Instead of having a RadialServer, a new module called ImageServer is implemented.

The ImageServer grabs images from camera and generates radial maps. Both the images and radial maps are stored in the shared memory. In addition, a control system based on the ERA idea and a user interface are added on the host side. The internal structure of control system has been described in Section 3.4.3. The control system and user interface are implemented in Java. The Sensor, Effector, Monitor, Exception Manager, Scheduler and Executor are all implemented as threads. The virtual robot is actually an interface between the shared memory and the control system.

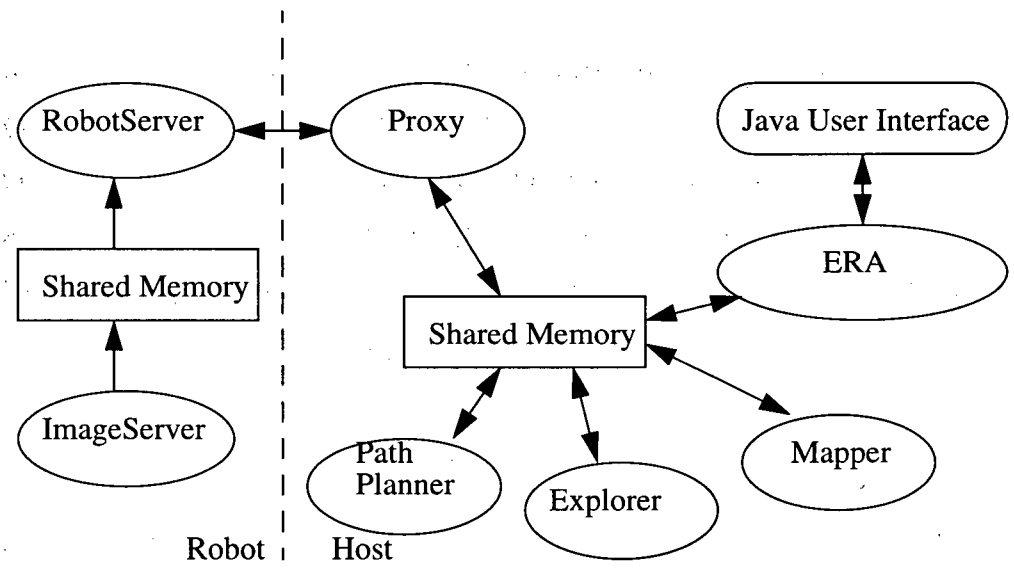

Figure 22: Software Architecture with Control System

5.2.2 Event and Event Handler in Java

It is not complicated to implement events and event handlers in Java. Figure 23 describes the relationship between an event and its event handler.

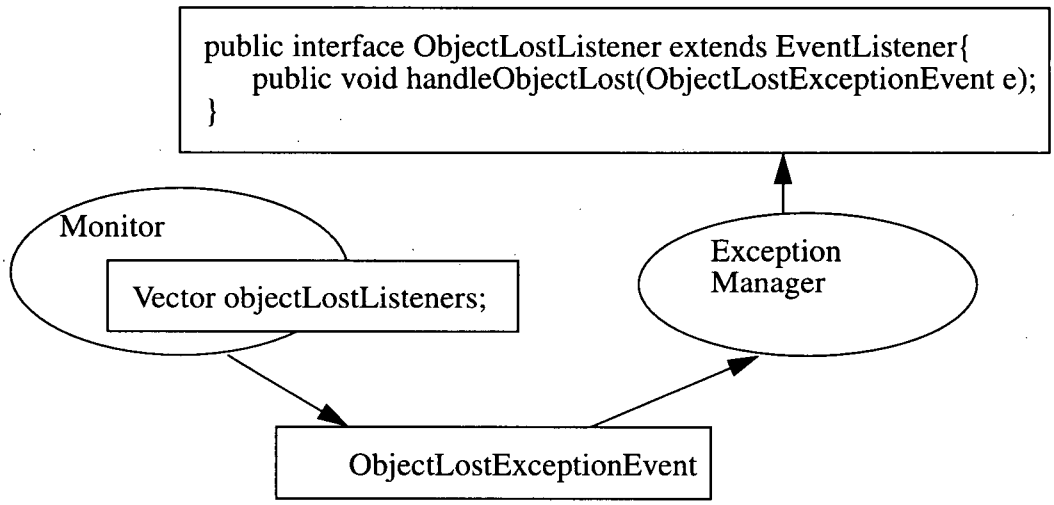

Figure 23: Event and Event Handler

Vector objectLostListeners is a list of registered exception event listeners on the Monitor side. The Exception Manager is one of the listeners for ObjectLostException-Event. It has a method called handleObjectLost which will be triggered when ObjectLos-tExceptionEvent arrives. The Monitor and the Exception Manager Modules are described in Section 4.5.

5.3 Experiment

5.3.1 Using Radial Map

In the experiment, our mobile robot Eric tracks a person in a real environment. To track a person, the fundamental method is to transfer images, grabbed from the camera, from the mobile robot itself to the host *Sol* where the control system is currently located. Then image understanding is performed for the consecutive images to locate the person in each image. The difficulties for this method are obvious. Each of the images has 120X160 pixels. It takes a significant portion of time for the control system to actually process these images. One way to reduce this large amount of data is to use ROI (Region of Interest) in which only a relative small portion of the data will be transferred and processed. But still a significant amount of data has to be transferred. The timely response requirement of the

system is affected.

To reduce the amount of data that has to be processed during the tracking and to meet the real-time requirement, we use the radial maps instead of real images to conduct the tracking experiment. The person being tracked is depicted as (Disparity, Column) in a radial map. The Disparity illustrates how far the person is from the robot. The Column indicates the vertical position of the person in the robot's view point. From these two values, the person's position in the robot's coordinates can be calculated. Thus, during the next time step, searching is performed around the this position to locate the person. Figure 24 shows an example of a radial map. The person in the radial map is described as (13, 80).

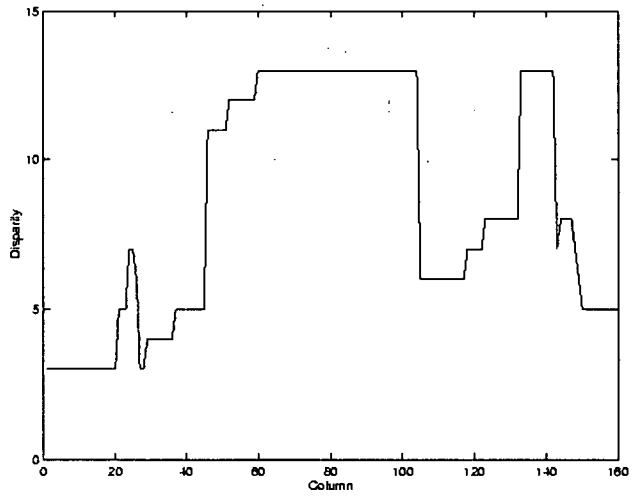

Figure 24: A Radial Map: Disparity vs. Column

5.3.2 Description of Tracking

The goal of our experimental task is to track a person in a real-world. This tracking task thus can be divided into two steps since the mobile robot has to find the target person first and then begin to track the person. Therefore, two major subtasks are *searching for*

the target person and *moving toward the target person*. To keep the target person in the middle of the robot's viewpoint, there are two other subtasks, namely, *tracking by turning left* and *tracking by turning right*.

At the beginning of the tracking task, the *searching for the target person* subtask is scheduled to execute by the Scheduler module. During the execution of this subtask, real images are grabbed and sent over to the robot host at fixed time interval while the mobile robot is spinning around and trying to locate the person in its viewpoint. To simplify the person recognition processing, the target person holds a concentric circle sign as its identification. Figure 25 shows the concentric circle. Simple image understanding is performed to detect the circle instead of the person himself. Once the person is located in the mobile robot's viewpoint, the *searching for the target person* subtask finishes and the second subtask *moving toward the target person* will be executed. At the same time, the position of the target person in the radial map is recorded as (disparity, column).

Figure 25: The Concentric Circle Sign

During the execution of the *moving toward the target person* subtask, only radial maps are used to locate the person in the robot's viewpoint. The robot moves toward the target person while keeping him in the center of its viewpoint. In this process, various exceptions could occur. For example, the exceptions indicate that the robot is running out of power, or sonar is blocked, or most importantly, the target person is moving out of the robot's viewpoint. When these exceptions are generated by the Monitor module, they are sent to the Exception Manager module. The Exception Manager generates new subtasks based on the current robot's subtask and the incoming exceptions. These new sub-

tasks might change the current goal.

Either the *tracking by turning left* subtask or the *tracking by turning right* subtask is executed to keep the target person in the middle of the robot's viewpoint. Each of them begins executing when the target is in the robot's viewpoint but not in the middle of it. It will finish executing when the target moves into the middle of the robot's viewpoint.

5.3.3 Events

5.3.3.1 Action Events

An action event is fired by the Executor and handled by the Effector. Its information is one of the atomic actions such as moving forward and backward which is to be carried out on the physical mobile robot.

For the *searching for the target person* subtask of our tracking experiment, there are two action events. One of them lets the robot spin around once it is received and carried out by the Effector. The other one sends a command and asks the robot to grab the real images from the camera.

For the *moving toward the target person* subtask of our tracking experiment, the action event carries the command to tell the robot to move forward.

For the *tracking by turning left* subtask, the action event carries the command of turning left. For the *tracking by turning right* subtask, the action event carries the command of turning right.

5.3.3.2 Subtask Events

The subtask events are fired by the Planner and handled by the Scheduler when a new plan is generated. A subtask event is also fired from the Scheduler to the Exception Manager when one subtask begins execution. Each subtask event includes the informa-

tion about the current executing subtask of the robot. The Exception Manager uses this piece of information to decide which kinds of exception it is expecting.

Since there are *searching for the target person*, *moving toward the target person*, *tracking by turning left* and *tracking by turning right* subtasks to complete the tracking task, in total four subtask events are needed. Each one of them corresponds to a subtask.

5.3.3.3 Data Events

The world representation of the mobile robot is wrapped in this kind of event. The data in data events could be the sonar readings, the power reading, the robot position, the radial maps, or the images. Mainly it would be fired by the Sensor and handled by the Monitor.

For the *searching for the target person* subtask, there are two kinds of data events. Each of them carries a real image grabbed from the camera or the power reading.

For the *moving toward the target person* subtask, there are three kinds of data events. Each of them carries a radial map, the power reading or the sonar readings.

For the *tracking by turning left* and *tracking by turning right* subtasks, there are two kinds of data events. They are for the radial maps and power reading respectively.

5.3.3.4 Exception Events

Exceptions are generated when the Monitor module detects any failure. Each exception is also an event in our ERA. It is fired by the Monitor and handled by the Exception Manager. After receiving an exception, the Exception Manager will generate new subtasks according to the current subtask and the exception.

For all of the subtasks, one failure which could happen is that the voltage is dangerously low. After the Monitor reads the power reading and detects the failure, a Power-

LowException is fired and leads the robot to a final stop.

For the *moving toward the target person* subtask, other exceptions that could be fired by the Monitor are the TargetLostException, the TargetNotInMiddleException, and the SonarBlockException. The TargetLostException and the SonarBlockException will lead the mobile robot to search for the target person again. To recover from the TargetNotInMiddleException, the mobile robot will execute either the *tracking by turning left* subtask or the *tracking by turning right* subtask depends on the tendency of the target person's moving.

For the *tracking by turning left* and *tracking by turning right* subtasks, another exception is the TargetLostException. This also leads the robot to search for the target person again.

5.3.4. State Transition

While the robot is executing a subtask, it is in a state observed by us. This robot state has two aspects. One is the robot's current pursuing goal, the other is its current action. Whenever it changes its current executing subtask, the robot has a state transition. Two of the above four events could finally result in such a state transition. They are the subtask events and the exception events. Since an action event results in the change of the robot's action only, it is not considered as one of the causes of the state transition. When new subtasks are executed, they always result in a robot's state transition. The exception events do not cause state transition directly, but they do let the control system generate new subtasks to recover from failure. These new subtasks will finally be executed. Thus, the robot will have a state transition. Figure 26 describes the state transition of the tracking task.

The person tracking experiment successfully demonstrates the feasibility of the Event-based Robot Control Architecture.

5.4 Summary

This chapter gave an implementation of our prototype for an Exception-based Robot Control Architecture. The control system is built on top of the previous system developed by Don Murray. A RWI mobile robot Eric is used to conduct a person tracking experiment. The exceptions that might happen during tracking are described as well.

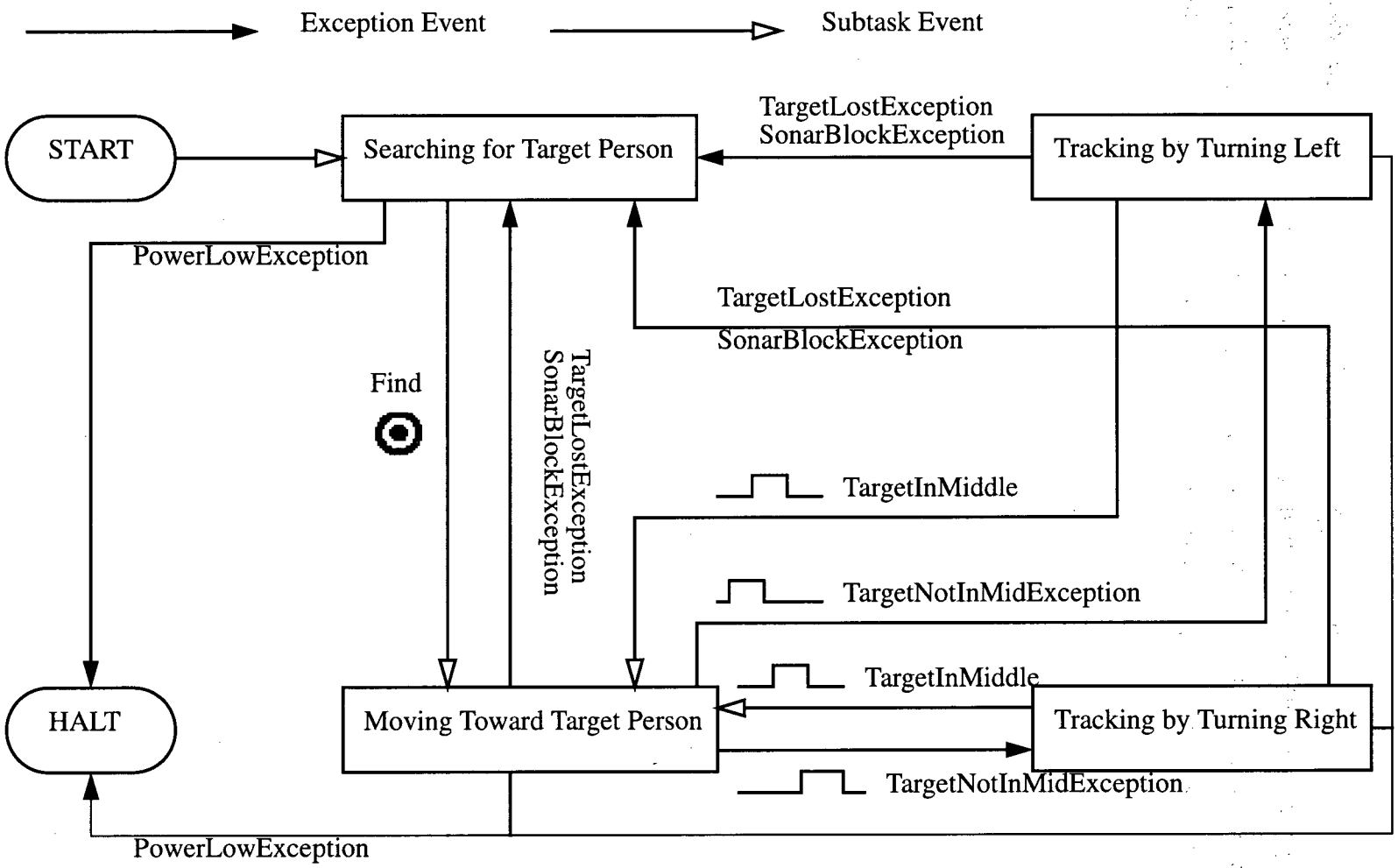

Figure 26: State Transition in Tracking Experiment

Chapter Six

Conclusion and Future Work

In this chapter, the system presented will be evaluated with comparison to the specifications outlined in Section 3.1. These comparisons will show that the current prototype implementation meet those system specifications. More work, however, could be done to improve the performance.

6.1 Specification Evaluation

This section evaluates the current ERA implementation against each of the specifications outlined in Section 3.1.

6.1.1 Robustness and Error Recovery

Our prototype implementation of the Event-based Robot Control Architecture demonstrates that minor changes in the environment will not cause fatal errors for the mobile robot. Neither will the robot end up wandering aimlessly while encountering changes.

Once an error is detected, efforts are made to recover from it instead of just giving up the current task. This error recovery is accomplished by using the exception mechanism. The system will generate a new set of subtasks to achieve the current goal based on this exception information. The tracking experiment fully demonstrates that our control system meets this specification.

6.1.2 Deliberative and Reactive

Neither a purely deliberative nor a purely reactive control system fits the current requirement for a mobile robot. The Event-based Robot Control System combines both

deliberative and reactive characteristics into one system. The deliberative aspect of the system carries out the plans toward the goal. Meanwhile, the reactive aspect of the system adapts the mobile robot to the continuously changing environment. As the tracking experiment shows, the mobile robot reacts to the real-world very well while keeping the deliberative goal.

6.1.3 Timely Response

In order to meet this specification, some of the time-consuming processing has been addressed earlier in this thesis. To reduce the time which the low bandwidth data waiting for its turn to transmit through radio-modem, high-bandwidth data is cut into pieces to be transmitted. To reduce both the transmission time and computation time, images are retrieved based on ROI (Region of Interest).

In addition to ensure timely response for subtasks that handle urgent situations, high priority is assigned to those subtasks that have to be executed first.

The prototype implementation of the Event-based Robot Control Architecture shows that these methods do have positive effect on the timely response issue. On the other hand, more improvement could result in a better response especially if the system has hard real-time deadlines.

6.1.4 Coordination Among Competing Activities

At the same time, there might have several subtasks waiting to be executed. Conflicts among these subtasks are resolved by assigning a priority to each of the subtask. The high priority ones will be executed first. Subtasks with the same priority are executed according to the first come first serve principle. The implementation indicates that this priority-based scheduling meets our specification and works well in our experiment domain.

6.1.5 Extensibility

By using event-driven communication between different modules, we are hoping

to extend the system without much difficulty later on. To extend the system to a larger scale, all that must be done is to add more events in one module and corresponding event handlers in other modules.

There is also a limitation in terms of extensibility in the Event-based Robot Control Architecture. It results from the nature of the exceptions. Every replanning process for recovery from a failure is based on both the failure and the current executing subtask. And every replanning leads the system to execute a new subtask. Thus, the relationship between subtasks is a network structure. To extend the system, the relationships between subtasks must be carefully considered.

6.2 Future Work

This section enumerates several improvements that could be made to the current architecture.

6.2.1 Real Time Control

Moving our mobile robot to a new environment with strict hard deadlines would require more consideration on real-time controlling. One possible solution to extend our system into a real-time control system is to have a real time scheduling algorithm. This could be done by improving the module Scheduler. Each of the subtasks which have hard deadlines requirement should have a deadline attached to it in addition to its priority. Thus, the subtasks will be executed according to not only its priority but also its deadline. The scheduler should be able to deal with the situation when some subtasks fail to be executed before its hard deadline. At least, it should be capable of halting the whole system and waiting for the command from human operator.

6.2.2 Coordination Among Multiple Subtasks

Our current Event-based Robot Control Architecture coordinates multiple subtasks based on their priorities. Let us consider the case where the robot is asked to go to two

places. The implemented ERA will schedule the two subtasks by their priorities. The one with higher priority will be scheduled first. If they have the same priority, the one that arrives earlier is executed first. Ideally we would like the system to evaluate the cost to go to these two places and to execute these two subtasks that could minimize the cost.

Another example in terms of coordination is as follows, the Scheduler receives "go to Office A" from the Planner and, at the same time, gets a subtask from the Exception Manager telling it to go back to the starting point to recharge the battery. Since the latter one has higher priority, the system will give up executing the former subtask and guide the robot to go back to its base. Improvement could be done to evaluate the voltage value first, decide whether the robot can go to Office A and go back to the base before the battery becomes dangerously low. If it can, the system should guide the robot to Office A first and then the starting point. It will only give up when the evaluation becomes false.

6.2.3 Using RAP in ERA

We gave a brief introduction in Section 2.3.2 about the RAP (Reactive Action Package) by R. James Firby. The execution of RAP queue is organized in a hierarchical style. When a primitive command is scheduled and sent to the RAP interpreter, it will be passed onto hardware through the hardware interface.

The world model in RAP is actually the world representation in ERA. The hardware interface to the robot becomes the virtual robot in ERA. The RAP execution queue could replace the priority queues of the Scheduler module in ERA. Subtasks in ERA are replaced by RAP. While each RAP is executing, exceptions in ERA could be used to detect failures and new RAPs will be generated and added to the RAP execution queue.

6.3 Summary

Each of the specification in Section 3.1 is evaluated first in this chapter. Section 6.2 provides the possible future work that could be done to the current Event-based Robot Control System. Issues discussed are real-time control, coordination among multiple sub-

tasks, and using RAP in our ERA.

Bibliography

[BD89] M. Boddy and T. Dean, Solving Time-dependent Planning Problems, Proc. Int. Joint Conf. Artif. Intell, 1989, pp 979-984.

[Blaasvaer94] Hans Blaasvaer, Paolo Pirjanian and Henrik I. Christensen, AMOR: An Autonomous Mobile Robot Navigation System, IEEE, Int. Conference on Systems, Man, and Cybernetics, 1994, Vol. 3, pp 2266-2271.

[Brooks86] Brooks, R. A., A Robust Layered Control System for A Mobile Robot, IEEE Journal of Robotics and Automation, Vol. 2, No. 1, 1986, pp 14-23.

[Burgard98] W. Burgard, A.B. Cremers, D. Fox, D. Haehnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun, The Interactive Museum Tour-Guide Robot, AAAI-98, pp 11-18, 1998.

[Crowley85] James L. Crowley, Navigation for An Intelligent Mobile Robot, IEEE Journal of Robotics and Automation, RA-1, 1985, pp 31-41.

[CY90] James J. Clark and Alan L. Yuille, Data Fusion for Sensory Information Processing Systems, Kluwer Academic Publishers, 1990.

[Durfee90] Edmund H. Durfee, A Cooperative Approach to Planning for Real-Time Control, Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, pp 277-283, Morgan-Kaufmann Publishers, 1990.

[Elfes89] A.Elfes, Using Occupancy Grids for Mobile Robot Perception and Navigation, Computer, 22(6): 46-57, 1989.

[Firby87] J. Firby, An Investigation into Reactive Planning in Complex Domains, Pro-

ceedings of the National Conference on Artificial Intelligence (AAAI), 1987.

[Gat92] Erann Gat, Integrating Planning and Reacting in A Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots, AAAI 1992: 809-815, 1992.

[Horvitz87] E. J. Horvitz, Reasoning About Beliefs and Actions Under Computational Resource Constraints, Proc. Workshop Uncertainty in AI, 1987.

[Laffey88] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao and J. Y. Read, Real-time Knowledge-based Systems, AI Mag., Vol. 9, no. 1, pp 27-45, 1988.

[ME85] H.Moravec and A. Elfes, High-resolution Maps from Wide-angle Sonar, Proceeding of IEEE International Conference on Robotics and Automation, St. Louis, Missouri, pp 116-121, 1985.

[Moravec83] Hans P. Moravec, The Stanford Cart and The CMU Rover, Proceedings of the IEEE, 71, pp 872-884, 1983.

[Murray97] Don Murray and Cullen Jennings, Stereo Vision Based Mapping and Navigation for Mobile robots, ICRA'97, Albuquerque NM, 1997.

[Murray98] Don Murray and Jim Little, Using Real-time Stereo Vision for Mobile Robot Navigation, Workshop on Perception for Mobile Agents, CVPR'98, Santa Barbara, CA, 1998.

[Musliner93] D. J. Musliner, CIRCA: The Cooperative Intelligent Real-Time Control Architecture Ph.D. Thesis, The University of Michigan, Ann Arbor, MI, 1993.

[Nilsson84] Nils J. Nilsson, Shakey The Robot, SRI AI Center Technical Note 323, April 1984.

[Rosenblatt97] Rosenblatt, J., DAMN: A Distributed Architecture for Mobile Navigation,

Journal of Experimental and Theoretical Artificial Intelligence, Vol. 9, no. 2/3, pp 339-360, April-September, 1997.

[Shafer86] Shafer, S.A., Stentz, A., and Thorpe, C.E., An Architecture for Sensor Fusion in A Mobile Robot, CRA86(2002-2011). BifRef 8600 And: CMU-RI-TR-86-9, April 1986.

[Simmons90] Simmons, R., An Architecture for Coordinating Planning, Sensing, and Action, Procs. DARPAWorkshop on Innovative Approaches to Planning, Scheduling and Control, 292--297, 1990.

[Simmons97] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J. O'Sullivan, A Layered Architecture for Office Delivery Robots, First International Conference on Autonomous Agents, Marina del Rey, CA, February 1997.

[Thorpe88] Charles Thorpe, Martial H. Hebert, Takeo Kanade, and Steven A. Shafer, Vision and Navigation for the Carnegie-Mellon Navlab, IEEE transaction on Pattern Analysis and Machine Intelligence, Vol. 10, No 3, May1988.

[Tsai87] R. Y. Tsai, A Versatile Camera Calibration Technique for High-accuracy 3d Machine Vision Metrology Using Off-the-shelf TV cameras and Lenses, IEEE Journal of Robotics and Automation, 3:323-344, 1987.

[Tsotsos98] John K. Tsotsos, Attention within A Distributed Action Vision Framework, Second International Workshop on Cooperative Distributed Vision, pp 57-81, 1998.

[Tucakov97] Vladimir Tucakov, Michael Sahota, Don Murray, Alan Mackworth, Jim Little, Steward Kingdom, Cullen Jennings, and Rod Barman, Spinoza: A Stereoscopic Visually Guided Mobile Robot, Hawaii International Conference on System Sciences, 1997.

[Williams98] Nicola Muscettola, P.Pandurang Nayak, Barney Pell, and Brian C. Williams,

Remote Agent: To Boldly Go Where No AI System Has Gone Before, Artificial Intelligence, 103: 5-47, 1998.

[ZM98] Y. Zhang and A. Mackworth, A Constraint-based Controller for Soccer-playing Robots, IROS'98 (Intelligent Robot Systems) Conference Proceedings, IEEE, Victoria, B.C., Canada, Oct 13-17, pp 1290 - 12951998.

Appendix A: A Few Issues For Programming The Robot

a. Off-board vs. On-board

It is always desirable to have all the system on-board, in another word, all the code located on the mobile robot itself. But in reality, having the whole system on-board causes difficulties in debugging the system. Therefore, during our implementation, most of the control system is off-board and located on the host. Although this separation helps with the debugging, it needs more consideration on the communication through radio-modem. Transferring large image data between the host and the robot via radio-modem will lead to a delay for the process of image understanding. To speed up the system, we could always develop and debug our system on the remote host, and migrate the system to the robot later on. It doesn't matter whether the current system is in Java or not, because the robot itself is also a Pentium machine running LINUX. However, the current program module robotServer on the robot must encounter some changes to communicate directly with the control system instead of via radio-modem through sockets.

b. Remote Invocation

The current Java robot APIs do not support remote invocation. This could be done by using Java RMI. Since the Java robot APIs are implemented by calling JNI (Java Native Interface) to access the shared memory which is implemented in C++, remote invocation might cause a security problem. My suggestion to this problem is to investigate the SecurityManager class in Java.

c. Temporal Facilities

Temporal facilities are useful while developing a system with hard real-time requirements. The current system has the temporal facilities. They are implemented as the timestamps for each commands and each piece of information received by the host. They are located in the shared memory.

d. Exception Class

The Exception Class in the current implementation inherits from EventObject class. Different exceptions simply inherit again from Exception Class. Each exception not only has a name to indicate the type of the exception but also carries the information which describes the failures. This information can be simply implemented as variables in the Exception Class.

e. Efficiency

One problem in building this Java control system on top of C++ code is its efficiency. In order to get information such as image data from shared memory to the Java program, it is inevitable to have memory copies. In terms of efficiency, it might be better to develop the whole system in C++.

f. Java Robot API

Different systems can be built upon the following basic Java Robot APIs. They are interfaces to the shared memory developed in C++.

Class robot.image.ImageMemory

```
java.lang.Object
    |
    +----robot.image.ImageMemory
```

public class **ImageMemory**
extends Object
The methods in ImageMemory class are interfaces to the shared memory which stores the image information. Through this class, images can be requested and retrieved from these methods.

Variable Index

- **disp**
The disparity image.
- **leftRaw**
The left raw image.
- **leftRect**
The right rectified image.
- **rightRaw**
The right raw image.
- **rightRect**
The right rectified image.
- **topRaw**
The top raw image.
- **topRect**
The top rectified image.

Constructor Index

- **ImageMemory**()
Attach to the image shared memory created already in other program modules.
- **ImageMemory**(boolean)
Create the image shared memory or attached to the image shared memory that has already created by other program module.

Method Index

- **destroy**()
Destroy the image shared memory.
- **detach**()
Detach from the image shared memory.
- **getImage**(int)
Return one of the image.
- **getUpdate**()
Update the shared memory pointer.
- **requestImages**(int, int)
Request specific image.
- **requestImages**(int, int, int, int, int, int)
Request specific image based on Region of Interest.
- **requestReady**()
Return whether the image requested is available to be retrieved or not.
- **setBaseline**(float)
Set baseline.
- **setComplete**(boolean)
When the requested image arrives, set complete flag in the shared memory.
- **setFocalLength**(float)
Set focallength.
- **setResolution**(int, int)
Set resolution.
- **updateImage**(int)
Copy the specific image data from the shared memory to the corresponding variable.
- **updateImageNCols**(int)
Copy the number of columns of the specific image from the shared memory to the corresponding variable.
- **updateImageNRows**(int)
Copy the number of rows of the specific image from the shared memory to the corresponding variable.
- **updateImageStartCol**(int)
Copy the starting column of the specific image from the shared memory to the corresponding variable in case of retrieving ROI only.
- **updateImageStartRow**(int)
Copy the starting row of the specific image from the shared memory to the corresponding variable in case of retrieving ROI only.

Variables

- **disp**

```
public ImageInfo disp
```

The disparity image.

- **rightRect**

```
public ImageInfo rightRect
```

The right rectified image.

- **leftRect**

```
public ImageInfo leftRect
```

The left rectified image.

- **topRect**

```
public ImageInfo topRect
```

The top rectified image.

- **rightRaw**

```
public ImageInfo rightRaw
```

The right raw image.

- **leftRaw**

```
public ImageInfo leftRaw
```

The left raw image.

- **topRaw**

```
public ImageInfo topRaw
```

The top raw image.

Constructors

- **ImageMemory**

```
public ImageMemory() throws MemNotAttachableException
```

Attach to the image shared memory that has already created by another program module.
Throws: MemNotAttachableException
if the shared memory can not be attached.

- **ImageMemory**

public ImageMemory(boolean create) throws MemNotAttachableException
Create the image shared memory or attach to the image shared memory that has already created by another program module.
Parameters:
create - a boolean indicates whether to create or to attach to the shared memory. If it is true, create the shared memory.
Throws: MemNotAttachableException
if the shared memory can not be attached.

Methods

- **destroy**

```
public boolean destroy() throws CanNotDestroyMemException
```

Destroy the image shared memory.
Throws: CanNotDestroyMemException
if the image shared memory can not be destroyed.

- **requestImages**

```
public synchronized boolean requestImages(int robot,
                                          int images)
```

Request specific images.
Parameters:
robot - which robot's image is going to request.
images - which image is being requested.
Returns:
true if the request is sent to the image shared memory successfully.
false if the request can not be sent to the image shared memory.

- **requestImages**

```
public synchronized boolean requestImages(int robot,
                                          int images,
                                          int r,
                                          int c,
                                          int rows,
                                          int cols)
```

Request specific images based on ROI (Region of Interest).
Parameters:
robot - which robot's image is going to request.
images - which image is being request.
r - start row of the requested image
c - start column of the requested image
rows - number of rows of the requested image
cols - number of columns of the requested image

Returns:
true if the request is sent to the image shared memory successfully.
false if the request can not be sent to the image shared memory.

- **setComplete**

```
public synchronized boolean setComplete(boolean complete)
```

When the requested image arrives, set complete flag to true in the shared memory to indicate that the image is ready to be retrieved. After the image request is sent, the complete flag is set to false.
Parameters:
complete - whether the requested image is available or not.
Returns:
true if the flag is set successfullly.
false if the flag can not be set.

- **updateImageStartRow**

```
public synchronized boolean updateImageStartRow(int image)
```

Copy the starting row of the specific image from the shared memory to the corresponding variable in case of retrieving ROI only.
Parameters:
image - the image type
Returns:
true if update is done.
false if update can not be done.

- **updateImageStartCol**

```
public synchronized boolean updateImageStartCol(int image)
```

Copy the starting column of the specific image from the shared memory to the corresponding variable in case of retrieving ROI only.
Parameters:
image - the image type
Returns:
true if update is done.
false if update can not be done.

- **updateImageNRows**

```
public synchronized boolean updateImageNRows(int image)
```

Copy the number of rows of the specific image from the shared memory to the corresponding variable in case of retrieving ROI only.
Paramters:
image - image type.
Returns:

true if update is done.
false if update can not be done.

- **updateImageNCols**

```
public synchronized boolean updateImageNCols(int image)
```

Copy the number of columns of the specific image from the shared memory to the corresponding variable in case of retrieving ROI only.
Parameters:
image - image type
Returns:
true if update is done.
false if update can not be done.

- **updateImage**

```
public synchronized boolean updateImage(int image)
```

Copy the specific image data from the shared memory to the corresponding variable.
Parameters:
image - image type
Returns:
true if update is done.
false if update can not be done.

- **requestReady**

```
public synchronized boolean requestReady()
```

When the requested image is completely transfered from the robotServer to the host side, a complete flag in the shared memory is set to true. This method returns the complete flag to indicate whether the image is avaible or not.
Returns:
true if the specific image is available.
false if the specific image is not avaible.

- **getUpdate**

```
public synchronized boolean getUpdate()
```

Update the shared memory pointer
Returns:
true if the update is done.
false if the update can not be done.

- **setResolution**

```
public synchronized boolean setResolution(int nrows,
                                          int ncols)
```

Set resolution
Parameters:
nrows - rows of the image
ncols - columns of the image
Returns:
true if resolution is set in the shared memory successfully.
false if resolution can not be set.

- **setBaseline**

```
public synchronized boolean setBaseline(float b)
```

Set baseline
Parameters:
b - baseline
Returns:
true if baseline is set in the shared memory successfully.
false if baseline can not be set.

- **setFocalLength**

```
public synchronized boolean setFocalLength(float f)
```

Set focallength
Parameters:
f - focallength
Returns:
true if focallength is set in the shared memory successfully
false if focallength can not be set.

- **getImage**

```
public ImageInfo getImage(int imgType) throws NoSuchImageException
```

return one of the image variable of this class.
Parameters:
imgType - image type
Throws: NoSuchImageException
if there is no such a image, or in another word, the image type is not valid.
Returns:
the specified image

- **detach**

```
public boolean detach() throws MemNotDetachableException
```

Detach the image shared memory.
Throws: MemNotDetachableException
if the image shared memory can not be detached.

Returns:
true if detach is done successfully.
false if detach can not be done.

Class robot.info.InfoMemory

```
java.lang.Object
    |
    +----robot.info.InfoMemory
```

public class **InfoMemory**
extends Object
The methods in this class are interfaces to the shared memory that stores the robot information such as robot position, power information and radial map.

Constructor Index

- **InfoMemory**(boolean, ImageMemory)
Create the robot info shared memory or attach to the shared memory that has already been created by other program modules.
- **InfoMemory**(ImageMemory)
Attach to the robot info shared memory that has already been created by other program modules.

Method Index

- **getNotifyFlag**(int)
Get sonar information .
- **getPlan**(int)
Get path plan information.
- **getRadialInfo**(int)
return radial map.
- **getRobotPosition**(int)
Get robot position.
- **getVoltage**(int)
Get robot's voltage information.
- **setBackward**(int)
Set command backward.
- **setForward**(int)
Set command forward.
- **setGoal**(int, int, int)

Set robot's goal position.
- **setGoal**(int, RobotPosition)

Set robot's goal position.
- **setHalt**(int)

Set command halt.
- **setLeft**(int)

Set command turn left.
- **setMode**(int, int)

Set robot's mode.
- **setNotifyFlag**(int, long)

Set robot's notifyFlag.
- **setPlan**(int, PathPlan)

Set path plan.
- **setRight**(int)

Set command turn right.
- **setTurn360**(int)

Set command turn 360.
- **updateRadial**(int)

Update radial map.

Constructors

InfoMemory

```
public InfoMemory(ImageMemory imageMem) throws MemNotAttachableException
```

Attach to the robot info shared memory that has already been created by another program module.
Paramters:
imageMem - image shared memory
Throws: MemNotAttachableException
if the shared memory can not be attached.

InfoMemory

```
public InfoMemory(boolean create,
                  ImageMemory imageMem) throws MemNotAttachableException
```

Create the robot info shared memory or attach to the shared memory that has already been created by another program module.
Parameters:
create - A boolean indicates whether to create or to attach to the shared memory. If it is ture, robot info shared memory will be created.
imageMem - image shared memory
Throws: MemNotAttachableException
if the shared memory can not be attached.

Methods

- **getVoltage**

```
public synchronized int getVoltage(int robot)
```

Get robot power information.
Parameters:
robot - which robot's power information is requested, eric or jose?
Returns:
the power of the specific robot.

- **getRobotPosition**

```
public synchronized RobotPosition getRobotPosition(int robot) throws GetRobotP
```

Get robot's current position (X, Y, H).
Parameters:
robot - which robot's position is requested, eric or jose?
Throws: GetRobotPositionException
if getRobotPosition cannot be finished successfully by calling java native method.
Returns:
the robot's current position.

- **updateRadial**

```
public synchronized RadialInfo updateRadial(int robot) throws GetRadialInfoExc
```

Copy the robot's radial map from the shared memory to the object of this class.
Parameters:
robot - which robot's radial map is requested, eric or jose?
Throws: GetRadialInfoException
if getRadialInfo cannot be finished successfully by calling java native method.
Returns:
the robot's radial map.

- **getPlan**

```
public synchronized PathPlan getPlan(int robot) throws GetPathException
```

Get robot's path plan. This path plan is generated by another program module planner.
Parameters:
robot - which robot's path plan is requested, eric or jose?
Throws: GetPathException
if getPath cannot be finished succesfully by calling java native method.
Returns:
the robot's current path plan.

- **setPlan**

```
public synchronized boolean setPlan(int robot,
```

```
                                    PathPlan plan)
```

Set robot's path plan.
Parameters:
robot - which robot's path plan is going to be set, eric or jose?
plan - the path plan that is going to be set for the robot.
Returns:
true if plan is set successfully to the shared memory.
false if plan cannot be set to the shared memory.

- **setTurn360**

```
public synchronized boolean setTurn360(int robot)
```

Set command turn 360 to the shared memory. After this command is read by proxy module, it will be sent over to the physical robot. The robot will turn until it finishes 360 degrees.
Parameters:
robot - the robot which will turn 360.
Returns:
true if command is set successfully to the shared memory
false if command cannot be set to the shared memory.

- **setHalt**

```
public synchronized boolean setHalt(int robot)
```

Set command halt to the shared memory. After this command is read by proxy module, it will be sent over to the physical robot. The robot will stop its current movement.
Paramters:
robot - the robot which will halt.
Returns:
true if command is set successfully to the shared memory.
false if command cannot be set to the shared memory.

- **setForward**

```
public synchronized boolean setForward(int robot)
```

Set command forward to the shared memory. After this command is read by proxy module, it will be sent over to the physical robot. The robot will move forward.
Parameters:
robot - the robot which will move forward.
Returns:
true if command is set successfully to the shared memory.
false if command cannot be set to the shared memory.

- **setBackward**

```
public synchronized boolean setBackward(int robot)
```

Set command backward to the shared memory. After this command is read by proxy module, it will be sent over to the physical robot. The robot will move backward.
Parameters:
robot - the robot which will move backward.
Returns:
true if command is set successfully to the shared memory.
false if command cannot be set to the shared memory.

- **setLeft**

```
public synchronized boolean setLeft(int robot)
```

Set command turn left to the shared memory. After this command is read by proxy module, it will be sent over to the physical robot. The robot will turn left.
Paramters:
robot - the robot which will turn left.
Returns:
true if command is set successfully to the shared memory.
false if command cannot be set to the shared memory.

- **setRight**

```
public synchronized boolean setRight(int robot)
```

Set command turn left to the shared memory. After this command is read by proxy modules, it will be sent over to the physical robot. The robot will turn right.
Parameters:
robot - the robot which will turn right.
Returns:
true if command is set successfully to the shared memory.
false if command cannot be set to the shared memory.

- **setMode**

```
public synchronized boolean setMode(int robot,
                                    int mode) throws NoSuchRobotModeException
```

Set mode to the shared memory. Current modes are manual, explore, directed.
Parameters:
robot - indicates which robot's mode is going to change.
mode - which mode is going to be sent over to the robot. One of the above current modes.
Throws: NoSuchRobotModeException
if it is not a valid mode.
Returns:
true if the mode is set successfully to the shared memory.
false if the mode cannot be set to the shared memory.

- **setGoal**

```
public synchronized boolean setGoal(int robot,
                                    RobotPosition goal)
```

Set robot's goal position to the shared memory. This is only effective while the robot is in directed mode.
Parameters:
robot - indicates which robot's goal position is going to be set.
goal - the goal position in (X, Y, H).
Returns:
true if the goal position is set successfully to the shared memory.
false if the goal position cannot be set to the shared memory.

- **setGoal**

```
public synchronized boolean setGoal(int robot,
                                    int x,
                                    int y)
```

Set robot's goal position to the shared memory. This is only effective while the robot is in directed mode.
Parameters:
robot - indicates which robot's goal position is going to be set.
x - the X of the goal position (X, Y, H).
y - the Y of the goal position (X, Y, H).
Returns:
true if the goal position is set successfully to the shared memory.
false if the goal position cannot be set to the shared memory.

- **setNotifyFlag**

```
public synchronized boolean setNotifyFlag(int robot,
                                          long flag)
```

Set Notifyflag in the shared memory.
Parameters:
robot - indicates which robot's notifyflag is going to be set.
flag - the notify flag.
Returns:
true if the flag is set successfully to the shared memory.
false if the flag cannot be set to the shared memory.

- **getNotifyFlag**

```
public synchronized int getNotifyFlag(int robot)
```

Get Notifyflag from the shared memory. This is used to get the sonar information.
Parameters:
robot - indicates which robot's notifyflag is going to get.
Returns:
the notifyflag.

● getRadialInfo

```
public RadialInfo getRadialInfo(int robot)
```

Return the robot's radial map.
Paramters:
robot - indicates which robot's radial map is being asked for.
Returns:
the specific robot's radial map.

Class robot.map.MapMemory

```
java.lang.Object
    |
    +----robot.map.MapMemory
```

public class **MapMemory**
extends Object
The methods in this class are interfaces to the shared memory that stores the maps such as plan map, zoom map, obstacle map and distance map.

Constructor Index

- **MapMemory**()
Attach to the robot map shared memory.

Method Index

- **getDistMap**()
Return distance map.
- **getEricZoomMap**()
Return eric zoom map.
- **getJoseZoomMap**()
Return jose zoom map.
- **getObstMap**()
Return obstacle map.
- **getPlanMap**()
Return plan map.
- **mapCentre**(int)
Get the center coordinates of the map.
- **mapToWorld**(int, int, int)
Transform the map coordinates (row, column) to world coordinates (X, Y).
- **recentre**(int, int, int)
Recenter the map.
- **recentre**(int, int, MapInfo)
Recenter the map.
- **updateDistMap**(int, int, int, int)

Copy the distance map from the shared memory to the object of this class.
- **updateEricZoomMap**()

Copy the eric zoom map from the shared memory to the object of this class.
- **updateJoseZoomMap**()

Copy the jose zoom map from the shared memory to the object of this class.
- **updateObstMap**(int, int, int, int)

Copy the obstacle map from the shared memory to the object of this class.
- **updatePlanMap**(int, int, int, int)

Copy the plan map from the shared memory to the object of this class.
- **worldToMap**(int, int, int)

Transform the world coordinates (X, Y) to the map coordinates (row, column).

Constructors

MapMemory

```
public MapMemory() throws MemNotAttachableException
```

Attach to the robot map shared memory.
Throws: MemNotAttachableException
if the shared memory cannot be attached.

Methods

updateJoseZoomMap

```
public boolean updateJoseZoomMap() throws GetMapErrorException
```

Copy the jose zoom map from the shared memory to the object of this class.
Throws: GetMapErrorException
if map cannot be get by calling java native method.
Returns:
true if map is successfully updated
false if map cannot be udpated.

updateEricZoomMap

```
public boolean updateEricZoomMap() throws GetMapErrorException
```

Copy the eric zoom map from the shared memory to the object of this class.
Throws: GetMapErrorException
if map cannot be get by calling java native method.
Returns:
true if map is successfully updated.
false if map cannot be updated.

- **updatePlanMap**

```
public boolean updatePlanMap(int r,
                             int c,
                             int getr,
                             int getc) throws GetMapErrorException
```

Copy the plan map from the shared memory to the object of this class.
Parameters:
r - row of the center of the map
c - column of the center of the map
getr - the number of rows to be updated
getc - the number of columns to be updated
Throws: GetMapErrorException
if map cannot be get by calling java native method
Returns:
true if map is successfully updated.
false if map cannot be updated.

- **updateObstMap**

```
public boolean updateObstMap(int r,
                             int c,
                             int getr,
                             int getc) throws GetMapErrorException
```

Copy the obstacle map from the shared memory to the object of this class.
Parameters:
r - row of the center of the map
c - column of the center of the map
getr - the number of rows to be updated
getc - the number of columns to be updated.
Throws: GetMapErrorException
if map cannot be get by calling java native method
Returns:
true if map is successfully updated.
false if map cannot be udpated.

- **updateDistMap**

```
public boolean updateDistMap(int r,
                             int c,
                             int getr,
                             int getc) throws GetMapErrorException
```

Copy the distance map from the shared memory to the object of this class.
Parameters:
r - row of the center of the map
c - column of the center of the map
getr - the number of rows to be updated
getc - the number of columns to be updated

Throws: GetMapErrorException
if map cannot be get by calling java native method
Returns:
true if map is successfully updated.
false if map cannot be updated.

- **getJoseZoomMap**

```
public MapInfo getJoseZoomMap() throws NoMapException
```

Return the jose zoom map.
Throws: NoMapException
if no jose zoom map is available.
Returns:
the jose zoom map.

- **getEricZoomMap**

```
public MapInfo getEricZoomMap() throws NoMapException
```

Return the eric zoom map.
Throws: NoMapException
if no eric zoom map is available.
Returns:
the eric zoom map.

- **getPlanMap**

```
public MapInfo getPlanMap() throws NoMapException
```

Return the plan map.
Throws: NoMapException
if no plan map is available.
Returns:
the plan map.

- **getObstMap**

```
public MapInfo getObstMap() throws NoMapException
```

Return the obstacle map.
Throws: NoMapException
if no obstacle map is available.
Returns:
the obstacle map.

- **getDistMap**

```
public MapInfo getDistMap() throws NoMapException
```

Return the distance map.
Throws: NoMapException
if no distance map is available.
Returns:
the distance map.

- **worldToMap**

```
public MapPoint worldToMap(int mapType,
                           int x,
                           int y)
```

Change a position from world coordinates (X, Y) to map coordinates (rows, column).
Parameters:
mapType - map type
x - X in world coordinates (X, Y).
y - Y in world coordinates (X, Y).
Returns:
the map coordinates (row, column) of (X, Y).

- **mapToWorld**

```
public WorldPoint mapToWorld(int mapType,
                             int row,
                             int col)
```

Change a position from map coordinates (row, column) to world coordinates (X, Y).
Parameters:
mapType - map type
row - row in map coordinates (row, column).
col - column in map coordinates (row, column).
Returns:
the world coordinates (X, Y) of (row, column).

- **mapCentre**

```
public WorldPoint mapCentre(int mapType)
```

Get the map center in world coordinates of a specific map.
Parameters:
mapType - map type
Returns:
the world coordinates of the center of the map.

- **recentre**

```
public void recentre(int mapType,
                     int x,
                     int y)
Recenter the specific map to (X, Y).
```

Parameters:
mapType - map type
x - X coordinate of new center (X, Y)
y - Y coordinate of new center (X, Y)

- **recentre**

```
public void recentre(int x,
                     int y,
                     MapInfo plan)
```

Recenter the plan map to (X, Y).
Parameters:
x - X coordinate of new center (X, Y)
y - Y coordinate of new center (X, Y)
plan - plan map
