A Software Framework for Developing High Quality Control Systems for Autonomous Robots

by

Darrell M. Lahey

B.Sc., Memorial University of Newfoundland, 2000

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 conforming
to the required standard

The University of British Columbia

November 2003

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Dec. 18, 2003

Abstract

This thesis presents a software framework and distributed execution system that allows developers to create control systems for autonomous robots. These control systems are behavior-based, and developers define them using networks of software components. The systems allow sequencing of behavior execution without coupling task sequencers with specific behaviors, and they allow groups of behaviors to be managed as single, cohesive, units. The presented software framework leverages the computational power of the Java programming language, while shielding developers from network communication details. In addition, it facilitates the development of high quality control systems, where quality refers to system usefulness and development ease.

The presented software framework supports the functionality required to create control systems that allow robots to perform human-like tasks in environments inhabited by humans. Most importantly, it supports both deliberation and reactivity, along with task sequencing and the coordination of action requests. A navigation system for a simulated mobile robot, developed using this software framework, shows the usefulness of the framework and execution system. The framework does show weaknesses, however, in areas such as robustness, efficiency, and synchronization. Nonetheless, these weaknesses can be overcome, and the software framework and execution system, as a whole, show great potential.

Contents

List of Figures

Acknowledgements

First, I want to thank Jim Little, my thesis supervisor, for giving me the opportunity and support to finish my Masters degree. Second, I want to thank Alan Mackworth for taking the role of my second reader. Third, I want to thank Pantelis, Jesse, Don, Stephen, and the rest of the Robuddies group for providing help when I needed it. Fourth, I want to thank my family for being supportive and caring thoughout my entire life. Finally, I want to thank my girlfriend, Kefi, for encouraging me, and putting up with me, through the hard times of my thesis writing.

This work was funded by the Institute for Robotics and Intelligent Systems (IRIS) and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

DARRELL M. LAHEY

The University of British Columbia
December 2003

Chapter 1

Introduction

An autonomous robot is a stand-alone computer system that, without external intervention, performs tasks in a physical environment. Such tasks require the robot to sense the state of its environment and to interact with, and manipulate, its environment. A control system for an autonomous robot processes the sensed information and chooses the physical actions the robot must take to complete its assigned tasks. Robot control systems can be implemented in both hardware and software. For extra processing power, their functionality can be distributed amongst different processing units, which may exist outside the physical robot. Processes that execute on different computer systems can communicate over a computer network.

For my thesis project, my main goal was to formulate a method for people to develop distributed control systems for autonomous robots. This method would leverage the computational power of common high-level programming languages, while shielding developers from network communication details. In addition, it would facilitate the development of high quality control systems that allow robots to perform human-like tasks in environments inhabited by humans.

The remainder of this chapter introduces my thesis in four parts. Section 1.1 gives a motivational example that indicates the type of functionality my development method should support. Section 1.2 discusses the importance of software quality and describes the

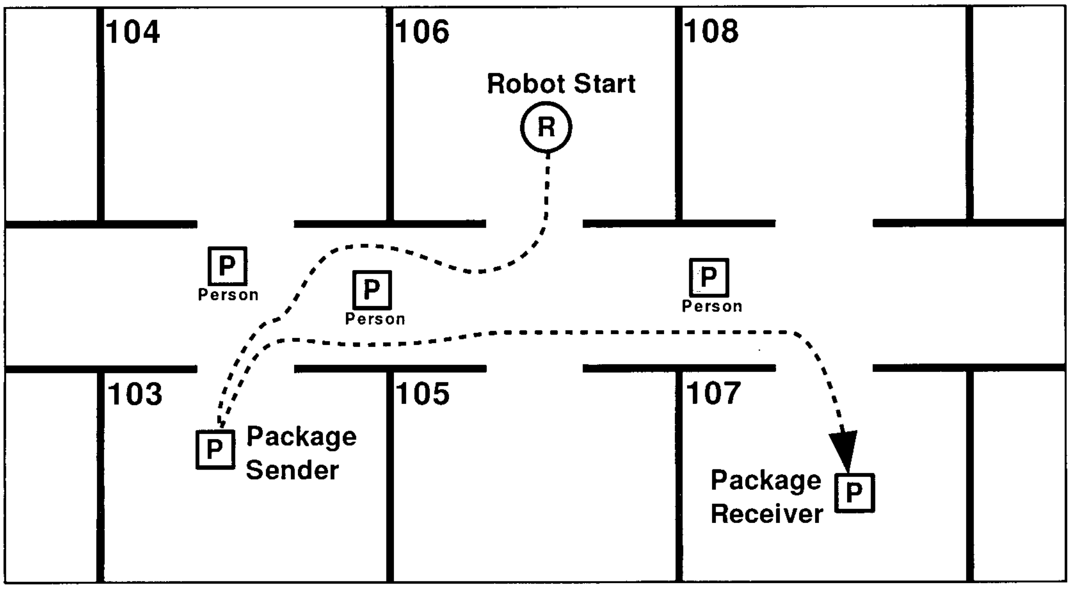

Figure 1.1: A Delivery Robot in Action

characteristics of a high quality software system. Section 1.3 gives an overview of my contributions to the field of developing control systems for autonomous robots. Section 1.4 outlines the contents of the remaining chapters.

1.1 Motivational Example

An example of an autonomous robot is one that performs delivery tasks in an office building. For these tasks, the robot must carry packages, through offices and corridors, to their destinations. The building contains working people, who may cross the path of the robot at any time. The robot must move quickly and smoothly while avoiding collisions with people and other obstacles.

Figure 1.1 depicts a robot performing a delivery task. In this example, the robot, starting in Room 106, must pick up a package in Room 103 and deliver it to Room 107. First, the robot must plan a path to Room 103 and follow it, while avoiding moving people. Upon reaching that room, it is given the package, and must move to Room 107 in the same manner. When the robot reaches Room 107, the recipient takes the package, and the task is complete. More complex examples are possible, where the robot must manage several delivery tasks simultaneously.

To perform its tasks autonomously, the robot must sense its environment to detect the locations of obstacles and its own location. It must rely on internal maps so that it may plan paths to its destinations. In addition, it must drive its motors properly such that it makes forward progress, while avoiding danger.

1.2 Software Quality

Robot control systems can consist of a huge number of heterogeneous parts that are arranged at various time and space scales and interact with each other in various ways. The many software parts may be implemented by different people using different programming languages and technologies. In addition, depending on the application, robot control systems may be used, and maintained, by different people for significant lengths of time. Such robot control systems, to be accepted by their users and developers, must consist of software parts that have high quality.

The quality of a software system has both external measures and internal measures, which are mostly qualitative. Software engineering textbooks commonly refer to these measures [McC93]. The external measures, such as efficiency and robustness, apply to users and how they perceive the software system. The internal measures apply to developers, and are related to the ease in developing, and maintaining, the system. High quality software systems rate high with respect to most, if not all, of these measures. Generally, such systems achieve a good balance among all the different measures.

The best software systems support all the functionality required by their users, while still maintaining high quality. Modern software engineering practice helps developers to construct high quality systems. It also allows developers to redesign existing systems such that they have higher quality. Software frameworks allow developers to implement the software components of a system in a common, consistent, way. Thus, they facilitate the development of high quality systems.

1.3 Thesis Contributions

For my thesis project, I created a software framework for developing distributed control systems for autonomous robots. This framework allows developers to specify models that define robot control systems as networks of software components. Along with the framework, I developed an execution system that has the ability to execute the systems that these models define. I implemented my software framework and execution system using the Java programming language. However, this does not imply that Java is the only possible choice.

I claim that my software framework, with its execution system, satisfies the goals of my thesis. First, it allows people to develop distributed control systems for autonomous robots. Second, it leverages the computational power of the Java programming language, while shielding developers from network communication details. I claim that my software framework facilitates the development of high quality control systems that allow robots to perform human-like tasks in environments inhabited by humans. However, because of time constraints, my software framework and execution system do have specific weaknesses. Nonetheless, I claim that, with particular additions and modifications, these weaknesses can be overcome.

1.4 Thesis Outline

This thesis presents my software framework and execution system, and it evaluates them to support my claims. Chapter 2 puts into perspective the field of creating control systems for autonomous robots. In particular, it gives an overview of the field, and it gives examples of existing robot control systems. Chapter 3 establishes several criteria for evaluating robot control systems. With the evaluation criteria, it provides a more formal definition of a software framework. Chapter 4 derives my software framework based on issues that I believed were most important. This derivation is conceptual in that it does not commit to any specific programming language or execution system.

Chapter 5 of this thesis describes my software framework and execution system implementation. In addition, it explains my reasons for choosing Java as the implementation language. Chapter 6 describes a robot control system I developed to demonstrate the usefulness of my software framework and execution system. This system allows a robot to perform navigation tasks. Chapter 7 evaluates my software framework and execution system in terms of the criteria established in Chapter 2. Chapter 8 makes additional notes, including suggestions for improving my framework and execution system. Chapter 9 concludes this thesis.

Chapter 2

Autonomous Robot Control Systems

Researchers have arrived at two main conclusions about control systems for autonomous robots [Ark98]. First, they have concluded that for a robot to complete complex tasks, which require proper sequencing of subtasks, that robot must use *deliberative* control, or *deliberation*. Second, they have concluded that for a robot to successfully inhabit dynamic, unpredictable, environments, that robot must use *reactive* control, or *reactivity*. *Hybrid* systems, which combine deliberation and reactivity, often work best in practice [AB97, BFG$^+$97, KM96, SGH$^+$97].

This chapter puts into perspective the field of creating control systems for autonomous robots. In particular, Section 2.1 describes the general capabilities of autonomous robots, focussing on robots that perform navigation tasks. Section 2.2 describes deliberative systems and gives specific examples of deliberative systems and architectures. Likewise, Section 2.3 describes reactive systems. That section also introduces *behaviors*, which are components often used in reactive systems. Section 2.4 describes hybrid systems and gives specific examples. Section 2.5 summarizes this chapter.

2.1 Autonomous Robots

As stated in the introduction to this thesis, an autonomous robot performs tasks that require it to sense the state of its environment and to interact with its environment. An autonomous robot senses the state of its environment through specialized hardware components called *sensors*. Similarly, it interacts with its environment through specialized hardware components called *actuators*. Often, these components have associated software that provide access to them at a more abstract level.

Many types of sensors and actuators exist. All sensors have a degree of uncertainty in their readings, which depends on the particular sensor and the operating environment. This means that the sensor readings are often not accurate. Actuators have similar uncertainties, since an actuator may not always do exactly what it is instructed to do. de Weerdt *et al.* provide a listing of the different sources of uncertainty for a mobile robot [dWdBvdHM98]. These uncertainties increase the difficulty for a robot to accurately model, and interact with, its environment.

Autonomous robots that can perform human-like navigation tasks in physical environments must have basic motion capabilities. Common actuators for robot motion include wheels, driven by motors [Nil84, SGH$^+$97], and leg-like structures [Bro89, Sim94]. Sensors that detect distances to solid objects are extremely useful for any autonomous robot that must navigate through an unpredictable environment. Knowing the distances to objects allows a robot to create maps of its environment [Elf89, Thr02] and localize itself within those maps [FBT99, SLL02]. Knowing that information also allows a robot to plan paths to its goal locations [LRDG90, MJ97], and detect oncoming obstacles [FBT98, Sim96]. Useful sensors for detecting distances to objects include vision sensors and sonar sensors.

Vision sensors use video cameras to acquire environment descriptions, which include distances to solid objects. They repeatedly capture images, where each image generally consists of a grid of pixels, and each pixel has a different color or intensity. An image from a video camera consists of a grid of pixels, where each pixel has a different color or intensity. With two or more cameras that view the volume of interest, a vision sensing system can estimate the distances to all points viewed in that volume [ML98]. The uncertainty in the distance measurements depend on the particular arrangement of the pixels in the cameras. Texture and reflection often have a significant impact on the estimated distances.

Sonar sensors find distances to objects using sound, where each sensor emits a sound signal in the direction it faces and then waits for that signal to return. When the sonar sensor detects the return of the sound signal, it can compute an upper bound on the distance to the closest object in the direction of emission. Factors such as the roughness and absorbency of surfaces, as well as the orientation of surfaces relative to the sensors, cause sonar sensors to yield excessive distance measurements.

Autonomous robots that perform navigation tasks should detect when they have collided with a solid object, or *obstacle*. Bump sensors provide one way to detect collisions. They are panels or extrusions on the surface of a robot that, when struck, declare that they are in contact with an obstacle. Robots often have several bump sensors spread about their surfaces so that they can detect collisions from a sufficient sampling of directions. Autonomous robots can benefit from sensing their internal properties, such as battery levels and odometry.

2.2 Deliberative Systems

The term *deliberation* indicates thorough thinking and planning. A deliberative robot carefully considers the results of all possible actions before choosing its next action. In general, it works within the sense-model-plan-act (SMPA) framework [Bro91]. Within this framework, a robot repeats a cycle, as depicted in Figure 2.1, that begins with sensing its environment (sense) and updating models of that environment (model). The cycle continues with planning sequences of tasks to achieve goals (plan) and issuing motor commands to perform those tasks (act). This section begins by describing the characteristics of deliberative systems. After that, it describes notable deliberative systems and architectures.

2.2.1 Characteristics

Arkin identifies the following five common characteristics of deliberative reasoning systems [Ark98].

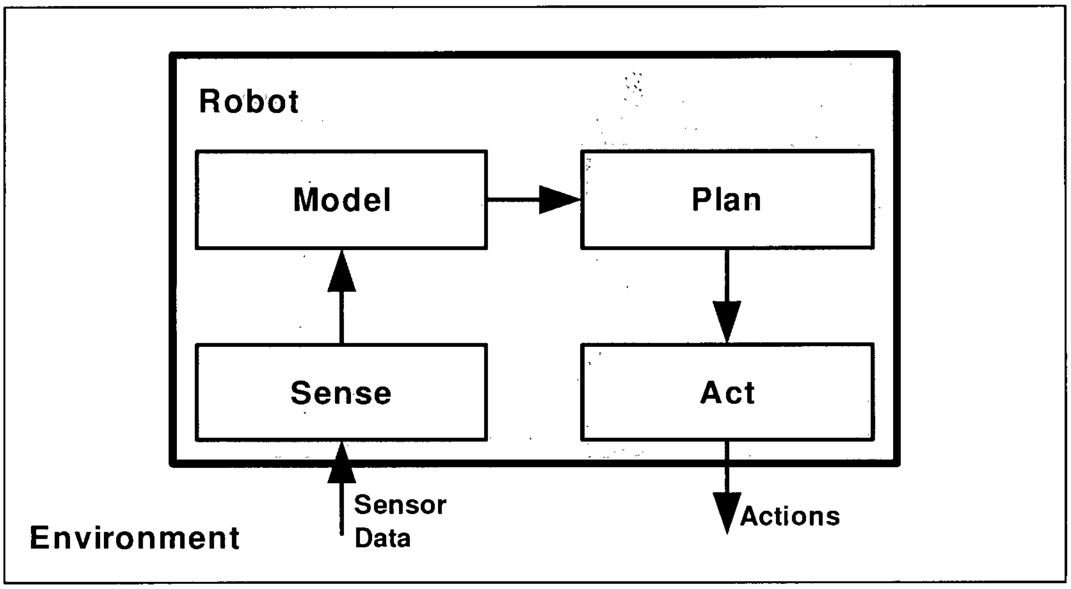

Figure 2.1: A Sense-Model-Plan-Act (SMPA) Cycle

- "They are hierarchical in structure with a clearly identifiable subdivision of functionality, similar to the organization of commercial businesses or military command."
- "Communication and control occurs in a predictable and predetermined manner, flowing up and down the hierarchy, with little if any lateral movement."
- "Higher levels in the hierarchy provide subgoals for lower subordinate levels."
- "Planning scope, both spatial and temporal, changes during descent in the hierarchy. Time requirements are shorter and spatial considerations are more local at the lower levels."
- "They rely heavily on symbolic representation world models."

Recall that deliberation allows a robot to complete complex tasks, which require proper sequencing of subtasks. An example of such a task, as described in the introduction, is a delivery task where a robot begins in a particular room and must deliver a package to a person in a different room. For simplicity, assume that the robot can predict, exactly, the outcomes of its actions and the movements of people and objects. This delivery task is extremely difficult unless the robot knows its own motion dynamics and the layout of the office building. For timely completion, it requires deliberation.

The control system for the delivery robot can be implemented as a hierarchy of planners. Planners take high-level goals and select partial orderings of tasks for achieving those goals. To do this, they generally employ thorough search strategies for finding the best sequence of tasks for achieving the goals. A classic example of a planner is STRIPS, which did planning for Shakey the Robot [Nil84].

STRIPS maintains a list of assertions that describe the current situation in a symbolic form. An example of an assertion is `on(A,B)`, which indicates that object A is resting on object B. The goal for planning is a logical combination of such assertions. Each action in STRIPS has preconditions, based on the assertions, which determine when the action can be applied. When STRIPS tests an applicable action, it modifies the list of assertions accordingly. Weld [Wel99] gives a survey of more recent planning systems.

In a hierarchy of planners, higher level planners take goals and select partial orderings of tasks to be handled by lower level planners. Each task may have its own goals, and lower level planners process these goals in the same manner. For the delivery task, the goal is to have a specific person receive the package carried by the robot. At this level of detail, a planner can select the tasks of picking up the item, moving the robot to the destination office, and giving the item to the specified person. The planner need not be concerned with low-level motor controls. Essentially, dividing planning systems into hierarchies of individual planners significantly simplifies the search space at each level.

For the delivery robot, a lower level planner can process the task of moving the robot to the destination office. The goal of this task is the robot being at its required destination. This goal can be achieved by selecting a series of local waypoints that the robot must pass to reach its destination. Planners at the lowest level of the planner hierarchy can output specific actuator commands for controlling the robot. This includes selecting the motor speed combinations that allow the robot to reach each waypoint. Clearly, from these descriptions, control flows down the hierarchy, and it becomes more localized at lower levels.

As Arkin stated, deliberative systems rely heavily on symbolic representations. These representations include models of the environment inhabited by the robot. They also include models of the robot, itself, since each action may affect the robot in a different way. Often, a robot requires more than its immediate perception of its environment. The delivery robot, for example, requires internal maps of the office building so it can plan paths to its destination. For optimal performance, the robot must be able to predict the actions of its environment, relative to its own actions.

Each level of the planner hierarchy depends on suitable abstractions of environment representations. For example, at a higher level, an office building can be represented as, simply, a collection of rooms and corridors, with connections among them. At this level, considering details such as the exact contents of rooms could make the planning tasks intractable. These details, however, may be useful at a lower level. At all levels of abstraction, accurate planning requires that these representations accurately reflect the real world. Deliberative systems utilize processes that take environment representations, starting from actual sensor readings, and create suitable abstractions for the different planners.

2.2.2 Examples

Probably the most notable example of a deliberative robot system is Shakey [Nil84]. A more recent architecture for developing deliberative robot systems is NASREM, the NASA/NBS Standard Reference Model [AML87]. The following paragraphs describe, and compare, Shakey and NASREM.

Shakey

Shakey [Nil84] was developed in the late 1960s at the Stanford Research Institute (now known as SRI International). Shakey could move throughout its environment using wheels driven by two independent motors. Its sensors included a video camera, an optical range finder (for measuring distance), and whisker-like bump sensors. It performed tasks such as pushing a solid block to a given location, and it planned these tasks using STRIPS, which executed on an external computer system.

Shakey was inherently slow at completing tasks. This slowness partly reflects computing technology of its time. Regardless of computational speed, Shakey was slow because it was completely deliberative, and it strictly followed the SMPA cycle. In other words, it fully constructed plans to achieve its goals before executing those plans. Shakey was tested in environments that were fabricated for its tasks, and it was successful in these environments.

NASREM

NASREM [AML87] defines a hierarchy of layers, where each layer consists of a sensory module, a world modeling module, and a planning module. Each sensory module, going up the hierarchy, forms more-abstract representations of sensor readings and updates environment models stored in global memory. Each planning module, going down the hierarchy, decomposes goals into lower level tasks. Each world modeling module allows its corresponding planning module to make queries about the current environment state, and it provide prediction and evaluation functionality.

Systems developed using NASREM are generally faster than Shakey at completing tasks. They are faster because, rather than executing one large SMPA cycle, they essentially execute a different SMPA cycle at each layer. Each layer, going up the hierarchy, executes its cycle at an increasing time scale. Lower levels, for controlling motors, execute with short time constraints, while higher levels have longer cycles.

2.3 Reactive Systems

The main problem with deliberative robot control systems is that, while they are effective in predictable environments, they fail in dynamic, unpredictable, environments [Bro91]. An example of an unpredictable environment is a large, crowded, room filled with randomly-moving people. Suppose a robot had recently moved from the Northwest corner of the room to the Southeast corner. Because of the random motion of people, any information the robot had about the locations of people in the Northwest corner is no longer valid. Internal representations of these dynamic, unpredictable, environments quickly become inaccurate and useless. Therefore, only immediate sensor data can be assumed to be valid.

Recall that a robot in a dynamic, unpredictable, environment must be able to quickly react to changes in that environment. For example, the robot must react quickly to avoid randomly-moving people. To react as quickly as possible, a robot control system must directly map environmental stimuli to physical actions. That is, it must quickly, and repeatedly, read input data from its sensors and, based on that data, send appropriate commands to its actuators. Such a robot control system is called a *reactive* system. This section describes characteristics of reactive systems, followed by descriptions of notable architectures for developing reactive systems.

2.3.1 Characteristics

Brooks [Bro91] has advocated the development of robot control systems that are, predominantly, reactive. Using his Subsumption architecture, he was successful in developing robots that exhibited basic animal behavior such as locomotion and object recognition. However, his robots do not have the planning capabilities made directly possible by deliberative robot control systems. He notes four important characteristics of reactive behavior-based systems. These are *situatedness*, *embodiment*, *intelligence*, and *emergence*.

- *Situatedness* means that a robot exists at a specific location, and in a specific configuration, in a physical environment. Purely reactive systems do not maintain any internal environment representations. Instead, they make decisions based on their immediate environments.

- *Embodiment* means that a robot is a physical being within its environment. Robots can sense their environments, and their actions directly influence their environments. All computation has a grounding in the real world, unlike in many symbolic reasoning systems.

- *Intelligence* means that a robot, to its observers, appears to employ complex reasoning and representation. However, reactive systems do not have complex reasoning abilities, and they do not maintain complex internal representations of their environments. Their perceived complexity, then, comes from their interactions with their environments.

- *Emergence* means that a robot, through interactions among its internal components, forms an overall intelligence. In other words, the apparent intelligence of purely reactive systems arises from these interactions. Complex reasoning and representation result from these interactions.

Behaviors

Behaviors are components that, depending on their perceived stimuli, give specific responses [Ark98]. Researchers have used other terms, such as "schema" [Ark98] and "skills" [BFG$^+$97], to describe this type of component. In software, behaviors are generally threads or processes that repeatedly read input data from specific sources, process that data, and send their results to specific destinations. An example of a behavior is one that uses input from vision sensors to compute commands for controlling the speed and heading of a robot.

For tight reactivity, behaviors interact directly with sensors and actuators. That is, their input sources are sensors, which provide representations of sensed data, and their output destinations are actuators, which accept commands. Behaviors can be used at higher levels of a robot control system. For example, Lenser *et al.* [LBV02] describe how they use "virtual" sensors and actuators for Carnegie Mellon University's entry at RoboCup 2000. Its behaviors retrieve abstract representations of sensor readings from the virtual sensors, and they issue high-level commands to virtual actuators. Low-level behaviors update virtual sensors based on real sensor data, and they break down virtual actuator commands into commands that can be executed on a physical robot. Still, behavior inputs and outputs are not restricted to abstractions of sensors and actuators.

Simple tasks can be performed by collections of behaviors executing concurrently,

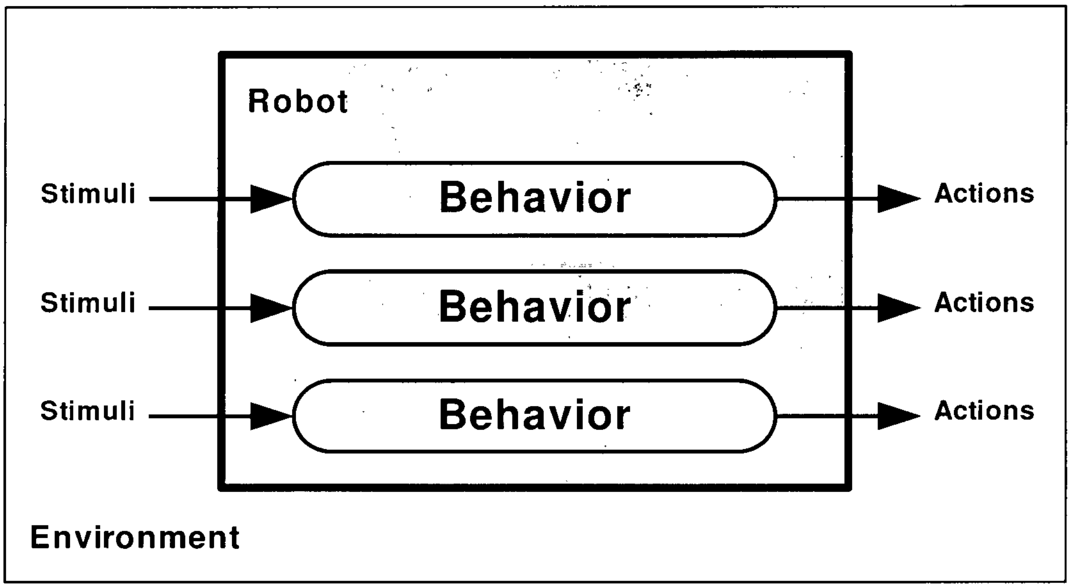

Figure 2.2: A Behavior-based Robot Control System

as indicated by Figure 2.2. For example, the task of moving a robot to a given location requires the robot to make progress toward that location, maintain smooth motion, and avoid obstacles. This navigation task can be achieved through the simultaneous execution of behaviors for each of these requirements. A robot can, then, perform more complex tasks by sequencing these simple tasks. Behavior-based systems perform tasks through the appropriate enabling and disabling of their behaviors.

Coordination

One inherent problem with robot control systems is that several components may be allowed to send output to the same destination. This destination may be an actuator, where different components may send different commands to that actuator, or it may be a data structure that is accessible by the different components. This problem can, particularly, arise in behavior-based systems, where several behavior may execute asynchronously. Such systems require coordination mechanisms that make the robot take appropriate actions in these situations. These coordination mechanisms generally require that each behavior has a weight, or a priority, with respect to other behaviors. Pirjanian [Pir99] gives a survey of coordination mechanisms.

Consider a behavior-based navigator robot that has two behaviors. Suppose that one behavior attempts to move the robot toward its goal location and the other attempts to move the robot away from oncoming people. It is possible that these two behaviors may, simultaneously, attempt to make the robot move in opposite directions. One behavior may set the direction first, and the other behavior may, quickly, cancel that setting. If each behavior repeatedly tries to set the direction, then the robot may viciously cycle between the two directions and yield unwanted actions. This robot requires a coordination mechanism.

2.3.2 Examples

Probably the most notable example of an architecture for developing reactive systems is Subsumption [Bro86] A more recent example is the Distributed Architecture for Mobile Navigation (DAMN) [Ros95]. The following paragraphs describe both of these architectures.

Subsumption

The Subsumption architecture [Bro86] was developed by Brooks in the 1980s at the MIT Artificial Intelligence Laboratory. It allows developers to specify robot control systems as layers of behaviors, where higher layers "subsume", or augment, lower layers. Each new layer gives a robot a higher level of competence. For example, the lowest layer could be responsible for making the robot avoid contact with obstacles. A higher layer would add the capability of wandering, such that the robot wanders without colliding with obstacles.

In the Subsumption architecture, each behavior is, essentially, a finite state machine with input ports and output ports. Repeatedly, each behavior reads its inputs, writes new data to its output ports, and updates its state. Each input port in each behavior is connected to a sensor or the output port of another behavior. Likewise, each output port is connected to an actuator or the input port of another behavior. A layer of a robot control system, then, consists of a collection of behaviors and their interconnections. Connections between these behaviors are low-bandwidth data channels.

Behaviors in higher layers augment behaviors in lower layers by writing data to the ports of the lower level behaviors. Whenever two behaviors write data to the same input port of another behavior, the higher level signal dominates. In particular, the higher level signal *suppresses* the lower level signal for a specified time period. Similarly, a behavior in a higher layer can *inhibit* the outputs of a behavior in a lower layer by writing to its output port. Thus, the coordination mechanism used in the Subsumption architecture is priority-based.

DAMN

DAMN [Ros95] was developed by Rosenblatt in the 1990s. It uses a vote-based coordination mechanism that overcomes problems caused by each input source specifying only a single preferred value. In a navigator robot, for example, if each behavior chooses a single rotational velocity, then the robot cannot make a mutually satisfactory control choice. Instead, through a priority-based mechanism or a blending mechanism, it may choose a bad command that, for example, makes the robot collide with an obstacle. Systems developed using DAMN make more informed choices of actions, which satisfy all contributing behaviors.

In DAMN, each behavior issues a vote value for each command in a set of possible commands. A set of commands, for example, may be a discretization of the velocities at which a robot can travel. For each command, a component, called the "arbiter", computes a weighted average over all vote values received for that command. It then applies, to the physical robot, the command with the highest average. Another component, called the "mode manager", provides the weights for each contributing behavior. These weights depend on the changing goals of the robot.

Rosenblatt mentions that work must be done to allow both coupled and independent voting. For example, votes for the translational velocity of a robot may depend on the rotational velocity, since certain combinations may be unsafe. Generally, both velocities should be reasoned about together (coupled voting), but behaviors should have the ability to work with only one (independent voting). Rosenblatt has formulated a different coordination technique that is based on utility fusion [Ros00]. With utility fusion, each behavior outputs desirabilities for different states that are reachable by the robot. The robot then chooses the commands that best satisfy the desirabilities of all the behaviors.

2.4 Hybrid Systems

Real-world environments are not completely predictable. In an office building, for example, the specific movements of people cannot be correctly predicted at all times. Therefore, representations of these environments always have a degree of uncertainty, which can cause robots to choose sequences of tasks that do not help it achieve its goals. However, real-world environments do have a sufficient level of predictability, which allows robots to make correct decisions most of the time. For example, office buildings have walls and desks that rarely move, so these objects can be considered to be static objects. In addition, the motions of people and other obstacles generally follow the laws of physics, so they can be correctly predicted within those constraints.

Recall that deliberative systems work well in predictable environments where robots must perform complex tasks. Reactive systems work better in unpredictable environments where robots must quickly react to environment changes. For real-world environments, which have a balance of predictability and unpredictability, systems that combine deliberation and reactivity should work well. Following this argument, researchers have created successful *hybrid* systems [AB97, BFG$^+$97, KM96, SGH$^+$97]. This section describes characteristics of hybrid systems, followed by descriptions of notable hybrid systems.

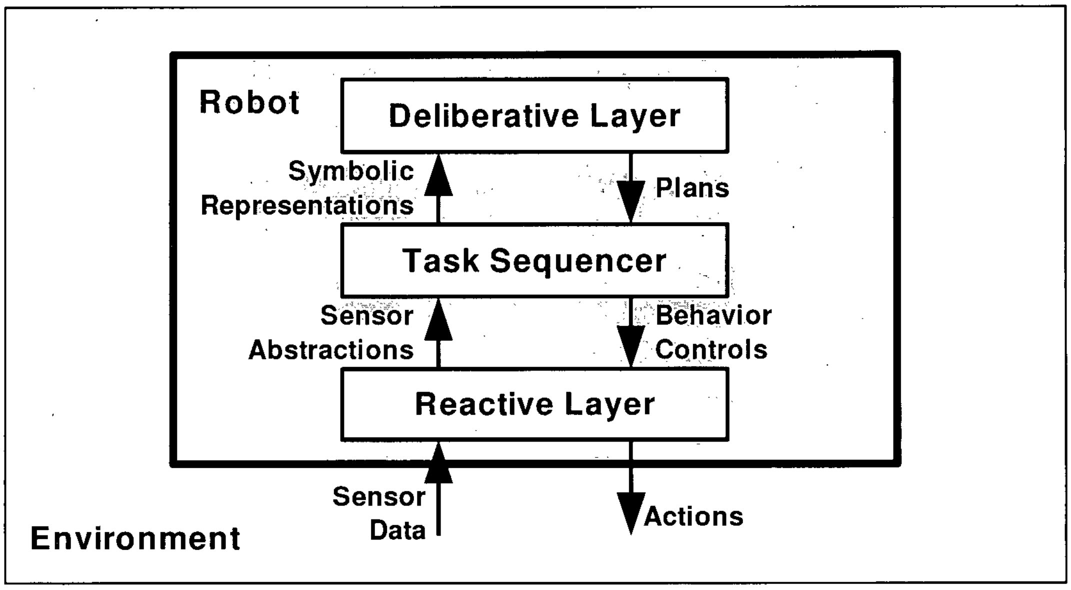

Figure 2.3: A Three-layered Hybrid System

2.4.1 Characteristics

Recall that hybrid systems combine deliberation with reactivity. The exact division between deliberation and reactivity, however, is different with each system and architecture. Layered approaches have been proven successful for developing hybrid systems [AB97, BFG+97, SGH+97]. Three-layered systems, for example, are common [BFG+97, SA98], where each layer may be subdivided into further layers.

In a three-layered hybrid system, as depicted in Figure 2.3, the lowest layer is the reactive layer. This layer communicates directly with sensors and actuators and provides immediate responses to environmental stimuli. It can be behavior-based, which means that it consists of behaviors that work together to perform tasks. The highest layer is the deliberative layer, which takes goals and computes sequences of tasks for achieving these goals. It maintains the accurate representations of the robot and its environment.

The middle layer of a three-layered hybrid system connects the deliberative layer with the reactive layer. In particular, it takes abstractions of sensor data from the reactive layer and provides the deliberative layer with symbolic representations of them. It also takes sequences of tasks from the deliberative layer instantiates the execution of those tasks at the reactive layer. Essentially, it mediates between the symbolic task descriptions, at the deliberative layer, and primitive actions at the reactive layer. For behavior-based systems, the middle layer often enables and disables behaviors in the reactive layer, and it may compute their relative weights for coordination.

In the field, the middle layer has been given different names, such as the "task sequencer" [BFG+97, Kon97] and the "executive" layer [SA98]. For this thesis, I use the term *task sequencer*. Examples of good task sequencing implementations are Reactive Action Packages (RAPs) [Fir89], Colbert [Kon97], and the Task Description Language (TDL) [SA98]. Each of these provides a language that allows developers to specify the task sequencing functionality. This functionality includes task decomposition, task synchronization, execution monitoring, and exception handling [SA98], as will be described in Section 3.2.

Task sequencers should be able to work within time constraints. Before choosing subtasks, a task sequencer should be able to retrieve estimates of the time required to complete each subtask. That way, it can choose the sequence of subtasks that best satisfy its time constraints. Tsotsos, in his S* Proposal [Tso97], describes one good strategy for selecting sequences of tasks under time constraints.

2.4.2 Examples

One of the first robot control architectures to take the hybrid approach is AuRA [AB97]. A more recent, and widely used, architecture is Saphira [KM96]. One architecture that I have worked closely with is that used in José and Eric [ML98, EHL+02, EHL03], two mobile robots at the University of British Columbia. The following paragraphs describe each of these systems in the order given.

AuRA

AuRA, the Autonomous Robot Architecture [AB97], was developed by Arkin in the 1980s as a hybrid approach to robot navigation. It combines a hierarchical layer of planners with a behavior-based reactive component. The original formulation of AuRA consisted of three planning layers. The top layer managed goals, which are, in this case, the locations to which a robot must move. The middle layer computed sequences of straight path legs that the robot must follow to reach its goal. The bottom layer selected appropriate subsets of behaviors that, together, control the motion of the robot.

AuRA defines different types of behaviors, which execute asynchronously. Most notably, it defines "motor schemas", which are behaviors that control the motors of a robot, given environmental stimuli. More specifically, each motor schema generates a force vector which indicates its desired direction and speed for the robot. For coordination, AuRA uses a weighted sum to combine the outputs of each motor schema into a single force vector. It then uses this vector to set the physical motor velocities. Developed motor schemas include those for moving a robot toward a goal point, keeping a robot on a desired path, and moving a robot away from sensed obstacles.

Saphira

Saphira is an "integrated sensing and control system for robotics applications" [KM96] that was developed in the 1990s at SRI International's Artificial Intelligence Center. It was originally developed for use with Flakey the robot, but is now distributed with several commercial robots. Saphira gave Flakey capabilities such as attending to, and following, humans, taking verbal advice from humans, and performing delivery tasks in an office building.

The central component in the Saphira architecture is a database, called the Local Perceptual Space (LPS), which maintains representations of the environment inhabited by a robot. Its representations have different levels of abstraction, and they include sensor readings, maps, and other useful data. For reasoning, the LPS maintains "artifacts", which are representations of physical, and artificial, entities. The LPS can maintain topological maps, for example, using artifacts that represent doors, corridors, and junctions. A task sequencer, called Colbert [Kon97], controls the execution of reactive behaviors.

Behaviors in Saphira are specified using fuzzy control rules, which map logical combinations of fuzzy variables to control commands. Fuzzy variables have values (between 0 and 1) that, generally, indicate the likelihood of a condition being true. Behaviors compute these values using information in the LPS. Applying a fuzzy control rule yields a value (between 0 and 1) that indicates the preference for the corresponding control command. An example of a fuzzy control rule is the following.

front-right-blocked OR *front-left-blocked* => *slow down*

This rule has fuzzy variables *front-right-blocked* and *front-left-blocked*, and its control command is *slow down*. It states that a robot should slow down if the space in front of it (to both sides) is occupied. Saphira combines the results of all control rules from all behaviors to choose the actual control commands to apply. It uses "context-dependent blending" to weight each behavior based on the current situation.

José and Eric

José and Eric [ML98, EHL+02, EHL03] are modified Real World Interface B14 mobile robots. They are approximately one metre high, without peripherals, and they are cylindrical in shape with 14-inch diameters. They can move forward and backward, and they can rotate left and right, all at various speeds. Each B14 robot is equipped with six bump sensors, distributed around its base, which allow it to detect collisions with solid objects. In addition, each robot is equipped with sonar sensors and infrared distance sensors, which are unused. Each robot houses a standard computer system that runs the Linux operating system and communicates with other computer systems over a wireless network connection.

José and Eric have been used for many types of scenarios, where each scenario required different modifications to the basic B14 robot. In all scenarios, both robots required vision capabilities. In particular, each robot has been configured with different vision sensors, including the Triclops, Digiclops, and Bumblebee stereo vision modules manufactured

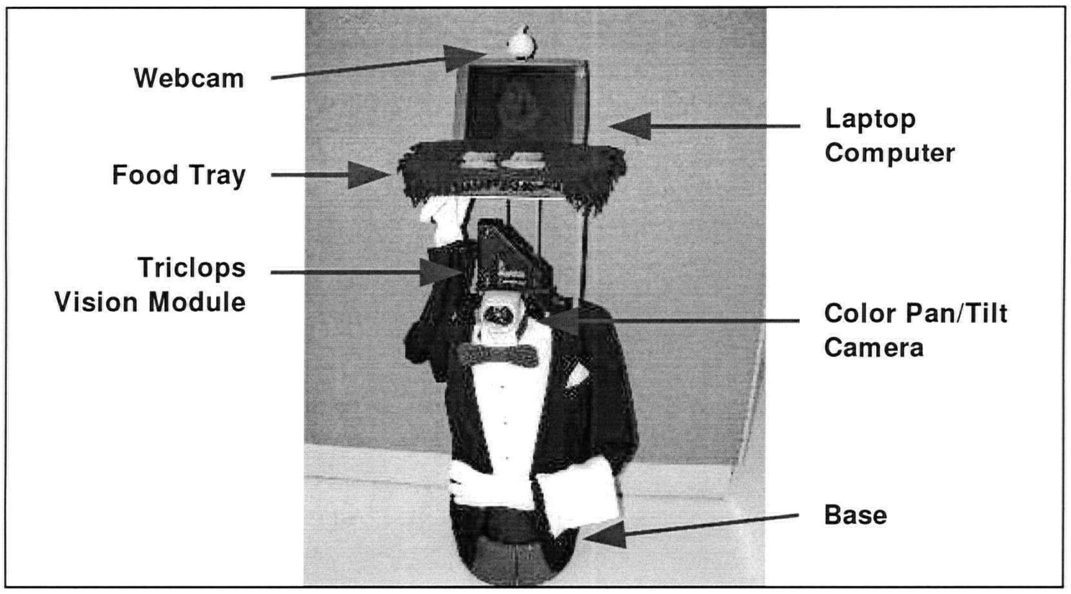

Figure 2.4: José, the Robot Waiter

by Point Grey Research. Probably the most famous incarnation of these robots is *José, the Robot Waiter*, which was UBC's winning entry into the 2001 AAAI *Hors d'Oeuvres Anyone?* competition [EHL+02]. For this competition, José could carry a food tray into a room of people, find groups of people to serve, detect when people have taken food from the tray, and return to its home base when its food tray became empty. Figure 2.4 contains a picture of José in its waiter configuration.

José and Eric have behavior-based control systems. Each behavior (or module) executes as an individual process on either the robot hardware or an external computer system. Each onboard module can read input from the sensors and send commands to the actuators. Communication among different modules may take place through sockets. Modules executing on the robot hardware can, alternately, communicate through a shared memory structure. This shared memory contains abstract representations of sensor readings, such as occupancy maps [ML98] and robot locations.

Control systems for José and Eric generally have a supervisor module that controls the other modules, via sockets, by sending data to them [EHL+02]. Data sent by the supervisor modules includes commands for enabling and disabling the modules. For each module, developers must write code that initializes communication with other modules. This may require code for setting up socket connections and acquiring references to data in shared memory. Communication over sockets requires that modules listen to those sockets for commands and data sent by other modules.

It is possible for different modules to, concurrently, send commands to the same actuator or update the same shared memory entity. Therefore, their control systems require a coordination mechanism. Some investigation had been done, however, into a coordination mechanism where behaviors construct bids for actions [SM94]. For improved efficiency, Eric has recently been equipped with extra processing units [EHL03]. These units allow modules to execute in parallel within the robot hardware, itself. However, Eric requires a control system that takes advantage of these processing units.

2.5 Summary

This chapter has shown that robot control systems, to be successful in real-world environments, must be both *deliberative* and *reactive*. Regardless of their specific architectures, they must coordinate the processing of sensor data, and the controlling of actuators, such that robots can perform useful tasks. Deliberative systems, such as Shakey [Nil84], generally do this by following the sense-model-plan-act framework. Reactive systems, such as those developed using the Subsumption architecture [Bro86], do this by directly mapping environmental stimuli to physical actions. Hybrid systems, such as AuRA [AB97], combine both of these extremes.

Chapter 3

Evaluation Criteria

This chapter presents my criteria for evaluating control systems for autonomous robots. As stated in the introduction, I use these criteria to evaluate my software framework and execution system. It begins, in Section 3.1, by presenting criteria related to the quality of software systems. There, it presents successful coding practices that developers use to create high quality software systems. Since most control systems for autonomous robots are, indeed, software systems, these criteria and coding practices apply. This chapter continues by presenting criteria specific to robot control systems. In particular, Section 3.2 presents criteria that have been established by four significant researchers in the field. Section 3.3 summarizes this chapter.

3.1 General Quality Criteria

Recall that software quality has both external measures and internal measures. High quality software systems satisfy these measures, as a whole. This section, first, presents the measures of external software quality. Then, it presents the measures of internal software quality. I base these measures of software quality on measures defined by McConnell [McC93] After listing the measures of software quality, this section presents the successful coding practices. For examples, it frequently refers to José and Eric [EHL+02], as described in Section 2.4, since I am familiar with their source code.

3.1.1 External Quality Measures

McConnell [McC93] defines eight measures of external software quality. They are *correctness*, *usability*, *efficiency*, *reliability*, *integrity*, *adaptability*, *accuracy*, and *robustness*. The following paragraphs explain each of these measures in the given order.

Correctness

Correctness is a measure of how well a software system adheres to its specification. If the system does not perform its intended tasks correctly, then it is not useful. As a simple example, if a system must add two numbers together and, instead, it multiplies them, then the system is not correct. A system, as a whole, can be either correct or incorrect. Large software systems, however, tend to have small errors that manifest themselves only in rare cases. Therefore, correctness can be measured more as an estimate of how often an error does occur during general usage. In high quality systems, errors either do not occur, or their occurrences are extremely rare. Many software systems, such as the control systems for José and Eric, are not thoroughly tested for correctness. Such systems may contain unidentified errors that can cause problems.

Usability

Usability is a measure of how easily a person can use, and learn to use, a software system. Consider two software applications that provide the same functionality, where the user must issue commands to call on that functionality. The first application requires the user to memorize dozens of text commands. The second application requires the user to select commands from well-organized, hierarchical, menus. When a user is first presented with both applications, that person may prefer to use the menu-based one, as it requires less effort to learn. However, after learning some text commands, the user may begin to prefer the application with text commands, as typing in commands may be quicker than navigating menus. For this example, perhaps the ideal software application would mix both options. In any case, high quality software systems are easy to learn and use.

José and Eric have, in some respects, good usability. Different people have developed different graphical user interfaces for controlling these robots. These graphical user interfaces allow users to, besides issue motor commands, visualize internal state representations such as occupancy grid maps and abstractions of sensor data. The usability for José and Eric is probably weakest in the startup sequence. It is weak here because users, when starting each control system, must start each module separately, and in a particular order.

Efficiency

Efficiency is a measure of the resources that a software system requires to complete its tasks. An efficient system minimizes the amount of time, memory, and hardware resources required to perform these tasks. For example, a system that takes one second to complete a task is much more efficient than a system that takes ten minutes, when both tasks yield identical results and require identical memory and hardware resources. Often, developers must sacrifice efficiency to create a system that is satisfactory with respect to the other measures. For example, to achieve numerical results with greater accuracy, a system may require more computation time.

The control systems in José and Eric support high efficiency. Each module, itself, can be implemented in a suitable language for maximum efficiency. Shared memory and hard-coded socket protocols allow for highly-efficient communications between different modules.

Reliability

Reliability is a measure of the ability of a software system to perform its tasks for long periods of time without failure. A system is more reliable than another if, when under expected conditions, it fails less often. For example, a system that is proven to fail, on average, once a year, is more reliable than a system that is proven to fail once a week.

José and Eric, depending on the application, can have low reliability. For example, José, when acting as a robot waiter, frequently suffered from broken socket connections over its wireless network. These situations generally caused its entire control system to fail. Thus, the reliability of José depended on the reliability of its wireless network, which was often unreliable.

Integrity

Integrity is a measure of how well a software system ensures that its data remains consistent, valid, and correct. A system with high integrity utilizes mechanisms to prevent unauthorized access to its data. These mechanisms include encrypting data files, such that their data cannot be decoded outside the software system, and requiring that users specify valid passwords.

A system with high integrity ensures that if pieces of data depend on one another, then a change in one piece will result in the appropriate change in the other. For example, consider a system that maintains a Boolean field that is *true* if and only if a particular integer field has a positive value. The system must update the Boolean field appropriately whenever the integer value changes between negative and positive. A system with high integrity ensures that its data values have the appropriate format and are within the required range. For example, if an integer field must contain a positive value, then the system should not allow it to take a negative value.

The control systems in José and Eric support high integrity in different ways. For example, developers can code system parts such that each executing module, itself, has high integrity. In addition, whenever a module writes to a block of shared memory, it can temporarily lock that memory. That way, other modules, assuming that they use the same locking mechanism, cannot concurrently write to the shared memory.

Although José and Eric have mechanisms for maintaining high integrity, their control systems can be further improved in terms of integrity. For example, their shared memory systems do not fully protect against misuse. Any module, if it has the correct numerical key, can access the shared memory and damage the data within it. In addition, modules that communicate through sockets do not authenticate one another. They require mechanisms to ensure that they are communicating with the expected modules.

Adaptability

Adaptability is a measure of how well a software system can be used in other, unintended, configurations and environments. If a system can be used, without modification, in new configurations and environments, then it has high adaptability. For example, a highly adaptable system is one that works with new input devices without significant external modification. José and Eric have high adaptability in the sense that they can be reconfigured with different camera systems, and other hardware, without significant external modification.

Accuracy

Accuracy is a measure of the closeness of the results generated by a software system to the actual results. Whereas for correctness, results are either be right or wrong, the accuracy measure applies to results that are generally correct, within a certain range. Accuracy is most often a factor when software systems use continuously varying quantities. Because computers cannot represent these quantities exactly, results of calculations often differ significantly from the theoretical results. In any case, high quality software systems can normally guarantee the accuracy of their results to within a specific range. The accuracy of a robot control system, such as that used in José and Eric, depends upon the specific application. For a navigation task, an accuracy measure could be the distance from a robot to its goal location, after that task has completed.

Robustness

Robustness is a measure of how well a software system responds to extreme or unexpected conditions, such as the failure of system parts. To be robust, a system must be able to recognize these conditions and handle them gracefully. For example, if the system encounters an error, it should not terminate harshly. In addition, it should not continue to execute if the error causes the system to perform its tasks incorrectly. Instead, the system should either try to recover from the error, or it should terminate in a way that is acceptable to users. Acceptable termination may include printing a message that explains why the execution terminated and providing a way for the user to recover the previous state of the system.

José and Eric have significant weaknesses with respect to robustness. If a module terminates unexpectedly, or a socket connection breaks, their control systems make no attempt to recover the broken parts. To resume normal operation, the user must, tediously, restart these control systems. Recall that José, when acting as a robot waiter, frequently suffered from broken socket connections over its wireless network. Its control system did not attempt to reconnect the disconnected pieces.

3.1.2 Internal Quality Measures

McConnell [McC93] defines seven measures of internal software quality. They are *maintainability*, *flexibility*, *portability*, *reusability*, *readability*, *testability*, and *understandability*. The following paragraphs explain each of these measures in the given order.

Maintainability

Maintainability is a measure of how easily a software system can be improved, modified, and extended. If a system is not easily maintainable, then its developers will require more time and energy to do their work. The source code of a system should be written in a way that helps to make errors easy to find. In addition, the source code should be written in a way that allows parts of it to be easily changed, and it should allow new parts to be easily added. Knowledge of good coding practices makes these requirements easier to satisfy.

José and Eric have high maintainability with respect to the modularity of their control systems. Recall that their control systems consist of several modules that communicate with each other. The addition of new modules is relatively simple, despite the fact that their source code must contain special code for connecting to shared memory and establishing socket connections. Besides that initial setup, and the knowledge of communication protocols, developers have no additional restrictions.

José and Eric have low maintainability with respect to the inconsistencies across different module implementations. Over the years, different people have contributed to the development of different parts of their control systems. Each developer used his, or her, own coding style, which made source code increasingly difficult to understand. In addition, developers often created unnecessary coupling between different source code parts in their attempts to give the robots improved capabilities.

Flexibility

Flexibility is a measure of how easily a software system can be changed such that it can be used for different purposes. It is similar to adaptability, but it applies to the source code. A flexible system allows parts of a system to be changed such that the effects on the remainder of the system are minimized. An example of a flexible system is one that has been coded such that it works with different video display configurations.

José and Eric have high flexibility because of the modularity of their control systems. Developers can easily add new modules, or modify the source code for existing ones, to give the robots new capabilities. However, each module is tightly coupled with specific communication media. In particular, each module communicates directly through sockets and shared memory. Because of this tight coupling, any changes to the communication media would require changes to the modules, themselves. Flexibility can be further improved by reducing this coupling.

Portability

Portability is a measure of how easily a software system can be modified such that it can operate in completely different execution environments. A portable system has the potential for more widespread usage. An example of a portable system is one that can execute properly on Linux systems and Microsoft Windows systems, where both versions share the same source code. Portability is achieved by avoiding the usage of system-dependent source code and libraries. When system dependence is required, the system-dependent source code should be separated from the non system-dependent source code.

Many of the modules within the control systems of José and Eric are not widely portable. This is mainly because they use a shared memory implementation that is supported only by specific operating systems. Porting their control systems from Linux to Microsoft Windows, for example, would require a complete rewriting of any source code based on shared memory. Luckily, the parts that use shared memory execute only on the robot hardware, and would never have any need for porting. Most parts that can execute outside the robot hardware can, without significant modification, be ported to different operating systems.

Reusability

Reusability is a measure of the usefulness of parts of a software system in different configurations. A high quality system consists of components that can be reused elsewhere in the system, or in different systems. For example, if a system has source code for creating and maintaining a linked list data structure, then many parts of that same system can make use of that code whenever they need linked list functionality. In addition, other systems can make use of that same source code. Most software compilers are distributed with libraries that provide useful functionality that developers can use in their source code. These libraries have been thoroughly tested for errors, and are guaranteed to be correct. By calling on prewritten routines, developers can avoid the errors that they can create when they write the routines themselves.

The individual modules within the control systems of José and Eric have high reusability. This is because each module operates independently, and each module can be inserted wherever its communication requirements are satisfied. If one module uses shared memory, then it can work with any shared memory implementation that has the same interface. Likewise, if it must establish a socket connection with a specific type of module, then a module of that type must be ready for connection and must support the same protocol. Modules with increased numbers of connections with other modules generally have decreased reusability.

Readability

Readability is a measure of how easily the source code of a software system can be understood by other developers, at the level of individual statements. Each identifier should have a self-descriptive name, and each statement should be formatted with appropriate spacing. Groups of statements should be accompanied by comments when those statements require further clarification. If a software system has good readability, then less time is wasted by developers that must learn how the source code works. The source code for José and Eric has high readability in some parts and low readability in others. This is reflected by the fact that many people contributed to writing the source code.

Testability

Testability is a measure of how easily a software system can be tested for errors. Developers can make a system more testable by breaking it up into small routines that have clear input and output specifications. They can test these routines using *unit testing* techniques. That is, for each routine, they can invoke the routine with inputs that exploit identified test cases. With each routine invocation, they can verify that the routine returns the correct value or performs the correct action. The source code for José and Eric has high testability in some parts and low testability in others. Differences in programming style among the different developers contributes to this variation.

Understandability

Understandability is a measure of how easily the overall structure of a software system can be understood through its source code. This measure is similar to readability, except that it applies on a more abstract, structural, level. If a software system has good understandability, then less time is wasted by coders that must learn the overall structure of the software system. Coders can make systems more understandable by dividing their source code into separate source files, modules, and routines. Each structural part of the source code should be accompanied by comments that give an overview of that part. While proper design documentation helps coders to understand the structure of software systems, appropriate coding practices help coders to create a close mapping between the source code and its documentation.

The source code for José and Eric has high understandability in terms of its modularity. The division of system parts into independent modules is a large contributing factor for this understandability. The quality of internal documentation, however, varies from module to module. Thus, some modules have better understandability than others.

3.1.3 Successful Coding Practices

Software engineers have identified successful coding practices that allow developers to create high quality software systems [McC93]. First, they have established the fact that software systems can be divided into smaller parts, where each part shares common functionality or data. These parts can be further broken down until a level is reached where all parts are simple routines. In each system, relationships and dependencies exist within parts and among different parts. Proper management of these relationships and dependencies is crucial to the development of high quality software systems.

Cohesion and *coupling* are two relevant terms in the management of relationships and dependencies among parts of a software system. The following paragraphs show how these terms apply. They also show how *object-oriented programming* helps developers to create systems that have strong cohesion and loose coupling. Such systems are generally high quality systems, with respect to the internal quality measures. Design patterns and software frameworks, which are also described in the following paragraphs, help to improve the quality of object-oriented software systems.

Cohesion and Coupling

Cohesion is the measure of the strength of relationships within individual parts of a software system. Recall that high quality systems have "strong" cohesion. This means that each part of the system consists only of subparts that are closely related. The relationships between subparts can be functional, where their routines have similar functionality. For example, math libraries consists of routines that perform mathematical functions. Similarly, relationships can be data-based, where all all routines act on the same data. For example, a part of a system can consist of a collection of routines for managing a linked list data structure.

Coupling is the measure of the number of dependencies among different parts of a software system. Recall that high quality systems have "loose" coupling. This means that dependencies among different system parts are minimized. As with the relationships within a system part, these dependencies can be functional, where one part calls routines that are defined in another part. Similarly, the dependencies can be data-based, where one part references variables defined in another part. It is important that systems are structured such that most dependencies are kept within the same parts of the system.

Figure 3.1 depicts a system that has strong cohesion and loose coupling. Each larger rectangle represents a cohesive part of the system, which consists of subparts (the smaller rectangles) that are closely related. An arrow between two rectangles indicates that one subpart depends upon the other subpart. The minimal number of arrows between the larger system parts indicates a low number of associations and, hence, loose coupling.

Strong cohesion and loose coupling result in high maintainability because systems with these properties localize functionality and data. If a system has an error that is related to specific functionality or data, the coder is likely to find the error in the part of the system that defines that functionality and data. If a part of a loosely coupled system is changed, that that change breaks a minimal number of dependencies. Loosely coupled systems result in high flexibility because they are structured such that if one part of a system must change, the other parts are affected minimally.

Systems that have strong cohesion and loose coupling have high portability when

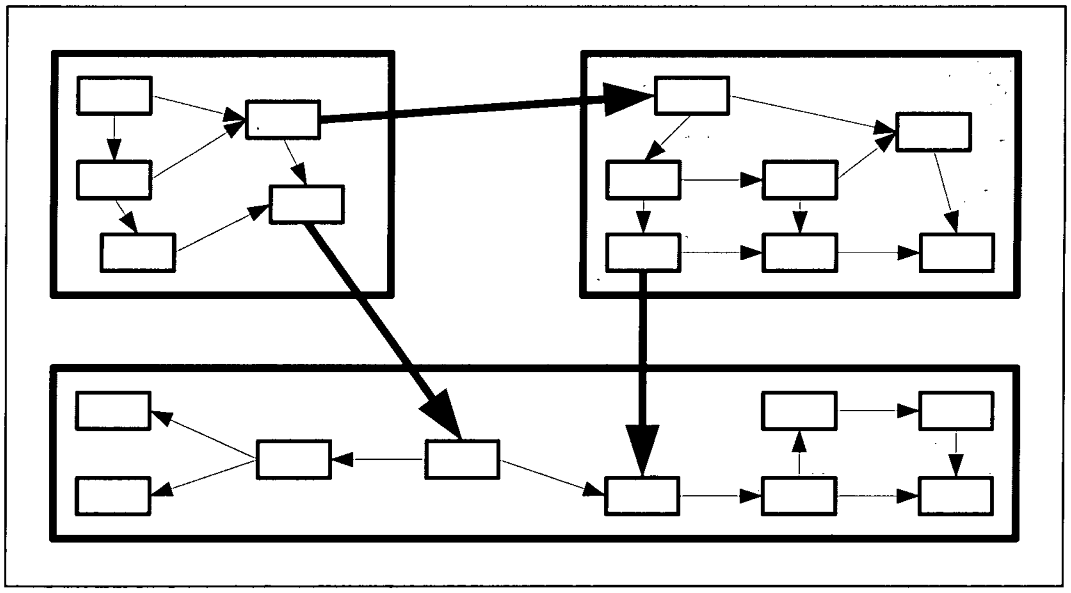

Figure 3.1: Strong Cohesion and Loose Coupling

they decouple system-dependent code from non system-dependent code. They have high reusability because, except for a minimal number of dependencies, large parts of the systems can be used in other parts, and within other systems, without change. They have high testability because they can be divided into mostly-independent parts that are easily testable. They have high understandability because a coder can understand each part of each system by looking at a minimal number of depended-upon parts.

Object-oriented Programming

Object-oriented programming is a common approach to developing high quality software systems [BD00]. It allows representations of complex entities to be packaged into cohesive data structures. *Object-oriented software systems* are viewed as collections of *objects*, where objects are these cohesive data structures. Each object encapsulates a collection of member variables, which define the state of the entity it represents. In addition, each object provides a collection of *methods*, which are routines that act on the object state.

Each object in an object-oriented software system is an *instance* of a *class*. Figure

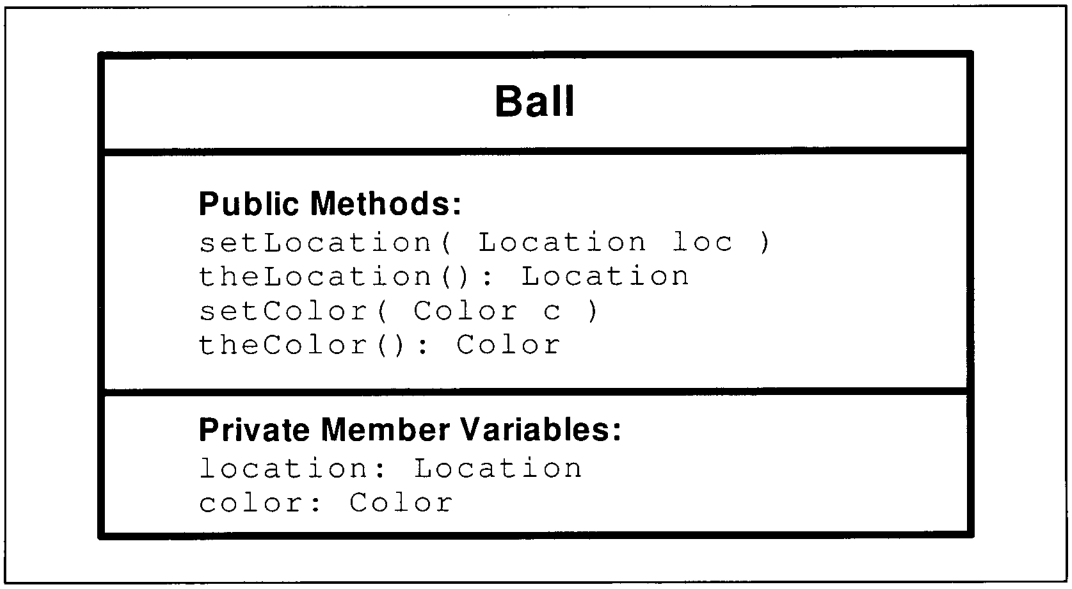

Figure 3.2: A Ball Class

3.2 depicts a class that is used to define ball objects. Each class defines the member variables and methods that comprise its instances. In addition, each class specifies which member variables and methods may be accessed from source code outside the class. In the figure, the ball class defines private member variables that store the location and color of a ball. The ball class also defines public methods that are used for setting and retrieving the information about the ball. Classes often define *constructors*, which are special methods that are invoked upon object creation to initialize the objects.

The object-oriented programming paradigm facilitates reusability because it allows each class to be used to create any number of objects. It also facilitates reusability by allowing developers to define new classes that are extensions of existing classes. By extending classes, developers form class hierarchies, where the *superclass* is the original class, and the *subclass* is the new class formed by extending the superclass. Figure 3.3 depicts a sample hierarchy, where class (`Ball`) is a subclass of class `Item`. Class `Ball` has two subclasses — `GolfBall` and `TennisBall` — which are two different kinds of balls. One advantage of extending a class is that instances of subclasses can be used wherever instances of

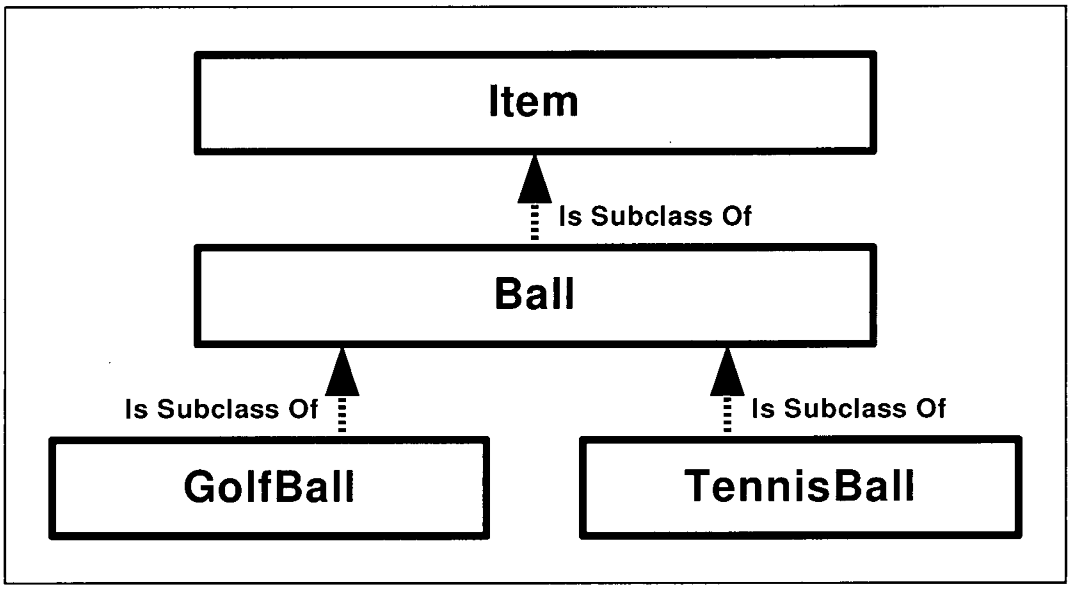

Figure 3.3: A Sample Class Hierarchy

their superclasses are required. Instances of class `GolfBall`, for example, can be used wherever an instance of class `Ball` is required. This allows the same source code to be used to interact with all subclasses of the required class.

Through class hierarchies, the object-oriented programming paradigm supports the declaration of *interfaces*, which provide common protocols for interacting with parts of a system. If two parts share a common interface, then one part can be substituted with the other part without affecting the remainder of the system implementation. In a class hierarchy, each superclass can act as a common interface for instances of all its subclasses. The subclass instances can be assigned to any variable that expects an instance of the superclass. Thus, parts of a software system that use these subclass instances, through superclass variables, are decoupled from specific subclass implementations. Some programming languages, like Java, provide direct support for interfaces by allowing developers to declare them as special types of superclasses.

Objects support strong cohesion, since they can represent single entities within a software system. For strong cohesion, objects should maintain an internal state that consists of data directly related to the represented entity. In Figure 3.2, for example, the color and location properties are directly related to the concept of a ball. For strong cohesion, each method should act on the internal state of the object in ways that are directly related to the represented entity. In the figure, method `setLocation`, which places the ball at a given location, is appropriate for the ball object.

Objects support loose coupling, since they can encapsulate data and prevent instances of other classes from directly accessing this data. Instead of providing direct access to member variables, an object can provide methods that return or modify the values stored in these member variables. This alternative yields developers the flexibility to change the representation of the stored data without requiring changes in parts of software systems that use its class. In the ball example, the location of the ball can easily be given a new data type, or a new name, if other parts of the system do not have direct access to that data.

Design Patterns

Design patterns are templates for class hierarchies and dependencies that provide reusable solutions for common design problems [GHJV95]. Thus, they help to improve the quality of object-oriented software systems. The two design patterns that are most significant to my thesis project are the *proxy* pattern and the *observer* pattern.

In the *proxy* pattern, as depicted in Figure 3.4, a special object acts as an intermediate layer between a method caller and the object that provides the required method. This special object, called the proxy object, provides methods with the same declarations as the public methods provided by the target object. The method caller holds a reference to the proxy object that, to the method caller, is identical to the target object. Rather than call methods directly on the target object, the method caller calls methods on the proxy object, which delegates the method call to the target object. Interface superclasses generally facilitate this substitution, which decouples the method caller from the specific target object.

The proxy pattern has several applications, which depend on how the proxy object handles its method calls. For my thesis, the most significant application is remote method

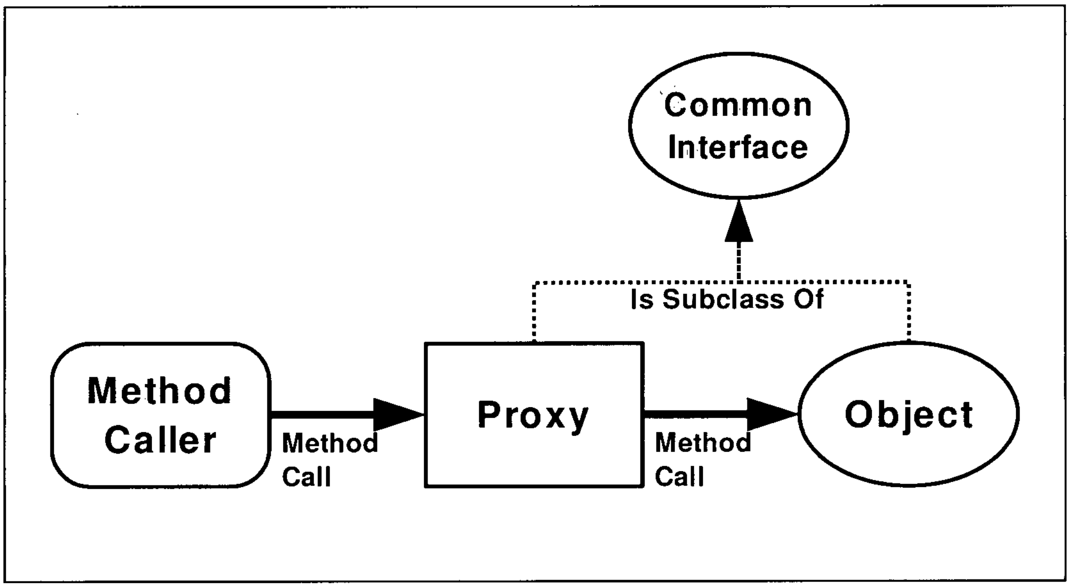

Figure 3.4: The Proxy Pattern

invocation, which allows methods to be called on objects that may exist on other computer systems. In this case, the proxy object has the ability to, through a medium unknown to the caller (such as sockets), initiate the invocation of the remote method. Generally, a collaborating object, on the remote side, must process the transferred data and make the actual method call. Each proxy method, then, forwards its argument values to the corresponding target method and returns the value that has been returned by that target method. Other applications of the proxy pattern include encryption of argument data, caching of method call results, and access restriction.

In the *observer* pattern, one or more objects register themselves as observers of a source object. When a specific event occurs in the source object, that object notifies all of its observers so that they may act appropriately. Figure 3.5 depicts the observer pattern. In particular, it shows an object notifying three of its observers.

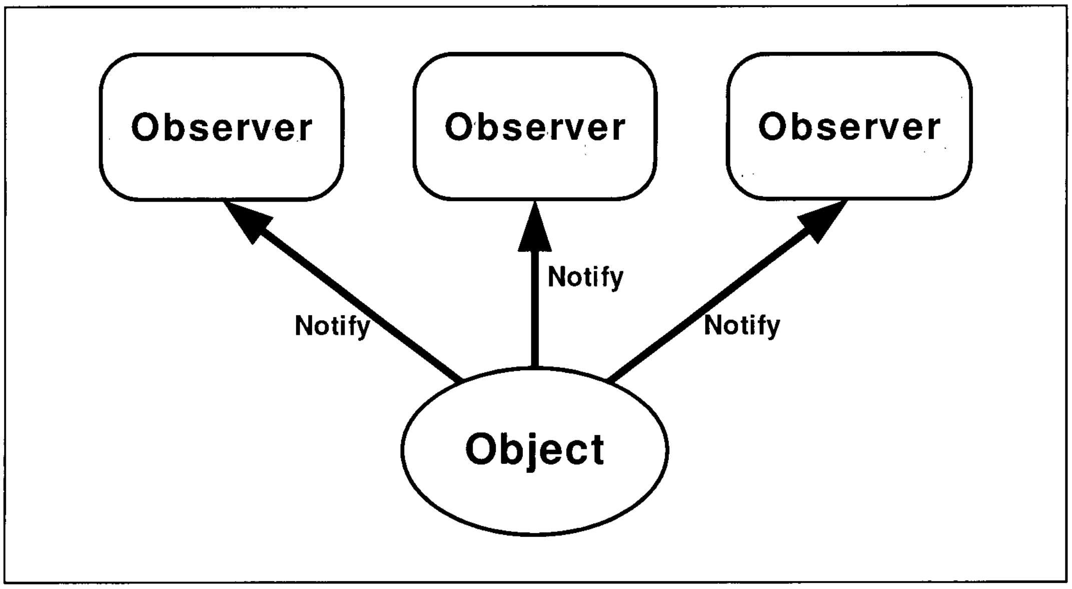

Figure 3.5: The Observer Pattern

Software Frameworks

A software framework is a reusable collection of related classes and class hierarchies. By extending the framework classes, developers can define subclasses that have a common structure. For example, these new subclasses may share common method declarations. Because of this common structure, systems that know about the framework can work with these subclasses, and their instances, in a consistent manner. A software framework may be accompanied by additional classes, such as classes that define useful data types and classes for organizing instances of the new subclasses. Software frameworks can be used to implement generic code for design patterns.

3.2 Robot-specific Criteria

This section presents evaluation criteria that have been established, separately, by Ronald Arkin, Rodney Brooks, Kurt Konolige, and Reid Simmons. It presents them in that order, referring back to the systems described in Chapter 2 for examples.

3.2.1 Arkin's Criteria

Arkin [Ark98] gives eight criteria for evaluating architectures that are used to develop robot control systems. Although he focuses on behavior-based architectures, his criteria can be used to evaluate any autonomous robot control system. His eight criteria are *support for parallelism*, *hardware targetability*, *niche targetability*, *support for modularity*, *robustness*, *timeliness in development*, *run time flexibility*, and *performance effectiveness*. Good systems generally satisfy most, if not all, of them. The following paragraphs discuss, in order, each of these eight criteria.

Support for Parallelism

An architecture supports parallelism if systems developed using that architecture can be easily divided into components that can execute in parallel. Such systems can complete more work, in less time, than systems that must execute entirely on the same processing unit. Consider a robot control system that must perform both vision processing and motion control. If a separate component can be developed for each of these tasks, and each component can execute independently, then the system will be able to do vision processing at the same time that it does motion control. If both components are implemented in software, then the vision processing software can execute on one processor, and the motion control software can execute on a different processor.

Executing different software tasks on different processors can be more efficient, in terms of time, than executing both tasks such that they share the same processor. Behavior-based systems generally have a good support for parallelism, since each behavior can execute asynchronously on a different processor. Thus, architectures like Subsumption [Bro86] and DAMN [Ros95], and the reactive components of AuRA [AB97] and Saphira [KM96], have good support for parallelism. Deliberative architectures, such as that used in Shakey [Nil84], often have less parallelism support.

Hardware Targetability

An architecture has good hardware targetability if it can be mapped easily onto the hardware of a real robot system. An example of an architecture with good hardware targetability is one where software interfaces to sensors and actuators can map directly to physical sensors and actuators. Software systems can interact with the virtual sensors and actuators as if they were the real, physical, ones. Less coding effort is required for such systems, since the mappings between the software and hardware are implicit in the system design. In this sense, reactive systems, which often work directly with sensors and actuators, have better hardware targetability than deliberative systems, which often work with symbolic representations.

An architecture also has good hardware targetability if systems developed using that architecture consist of components that can be implemented in hardware. Because hardware implementations generally execute faster than software implementations, systems with this type of hardware targetability have the potential to be more efficient. Architectures based on simple components, such as Subsumption [Bro86], have good hardware targetability in this sense. However, they lack the level of expressiveness provided by architectures such as Saphira [KM96], which are based on complex control and representation.

Niche Targetability

An ecological niche for a robot is defined by the relationships between a robot and its environment. An architecture has good niche targetability if it has the following two properties. First, the architecture must provide ways to express the required relationships. Second, the architecture must allow developers to, without significant effort, modify systems for operation in different environments. Robots, like animals, must adapt to changes in their environments in order to survive. If a human would rather do a task, rather than let a robot do the task, then the robot does not have good niche targetability with respect to that task.

Behavior-based architectures, such as Subsumption [Bro86], generally have good niche targetability. Their mappings from environmental stimuli to actuator responses clearly reflect the relationships between a robot and its environment. In addition, their modularity helps their configurability for different environments. Deliberative architectures, like NASREM [AML87], have lower niche targetability as they may require different types of representations for different environments. These different representations can require significant changes to system components, such as the global memory representations and world modeling modules in NASREM. For example, a navigator robot that stores two-dimensional occupancy maps of a flat ground plane may work well in an office building. For the robot to work well in the wilderness, full three-dimensional maps may be more useful. However, the robot would require significant changes to any components that work with the maps and derived representations.

Support for Modularity

An architecture supports modularity if systems developed using that architecture provide abstractions for collections of lower level components. Modules are collections of components that allow access to these components at an abstract level. In object-oriented programming, for example, objects are modules that provide methods that can be called, but the implementations of these methods are hidden and irrelevant. A modular architecture "makes a developer's task easier and facilitates software reuse" [Ark98]. If a module is designed appropriately, then a developer can easily reuse the same module within the same system or within a completely different system.

Consider a robot system that contains a motion controller module, where that module calls on a vision module to determine where the robot can safely move. The motion controller module does not need to know the specific implementation of the vision module. Instead, it needs to know, only, that it can call on the vision module to retrieve a description of what is seen by the robot. If the vision module implementation changes, but the interface to that module remains the same, then the motion controller should not require any changes.

Behavior-based architectures generally have good modularity because each behavior is an independent unit with clear input and output requirements. However, specific dependences on other behaviors, as occurs in Subsumption [Bro86] when new layers augment existing layers, reduces this modularity. DAMN [Ros95] has better modularity, in this sense. Deliberative architectures, such as Shakey [Nil84], which have large planners that work with complex world models, have poorer modularity. NASREM [AML87], with its hierarchy of planners, has somewhat better modularity.

Robustness

Robustness was defined in Section 3.1 in the context of software quality. Consider a mobile robot whose vision sensor becomes damaged. This damage prevents the robot from perceiving obstacles in its environment. In a system that is not robust, the robot may continue to move, thinking nothing is in its way, toward imminent collision. The control system for this robot would be more robust if it could recognize that the vision hardware did not work properly. Then, it could compensate for that loss by moving toward its target at a speed that is too slow to cause damage. The system would be even more robust if it could repair, or replace, the damaged component.

Architectures that allow developers to use general purpose programming languages, such as Saphira [Act99], have a greater risk of error introduction. Thus, they may lack the robustness provided by architectures, such as Subsumption [Bro86], that use specialized languages with simpler semantics. Nonetheless, with proper design, such systems can be implemented such that they localize the effects of failures.

Timeliness in Development

An architecture supports timeliness in development if robot systems can be developed quickly and easily using the architecture. The development time for these systems can be reduced if tools are available that simplify the work of the developer. Useful tools include specification languages and graphical editors. Some architectures, such as those modeled after complex biological systems, are philosophical in nature. Such architectures cannot be easily implemented on real systems, so they do not support timeliness in development.

Architectures based on networks of components, such as Subsumption [Bro86], have good support for timeliness in development. These architectures can allow developers to create graphical tools for connecting the different components. The timeliness in development for individual components often depends on the specification semantics. Architectures, such as Saphira [Act99], that are based on common programming languages can rely on available tools for those languages, including editors and debuggers. Simpler languages allow such tools to be developed more quickly. Architectures with good support for modularity facilitate code reuse and, therefore, timeliness in development.

Run Time Flexibility

An architecture supports run time flexibility if systems developed using that architecture can be easily reconfigured during execution. A robot-related example is a system that has a component for planning an optimal path to a given destination. That system has good run time flexibility if it allows the path planning component to be reconfigured, at run time, such that it uses a different algorithm or different input parameters. It can have better run time flexibility if it can reconfigure itself. The robot system can apply learning algorithms such that it can learn the best ways to reconfigure itself for specific situations.

Saphira [KM96], with its context-dependent blending, has good support for run time flexibility. It allows its behaviors to be weighted differently depending on the current situation. DAMN [Ros95] has similar support for run time flexibility. Systems developed using hard-wired networks, such as those developed using the Subsumption architecture [Bro86], have poorer run time flexibility.

Performance Effectiveness

Performance effectiveness is a measure of how well a robot performs the tasks it was developed for. How well a robot performs a given task can be measured in different ways, which depend on the particular task. Consider a robot that must move to a given goal location. For this robot, useful performance measures are the elapsed time, the path smoothness, the energy consumed, and the clearance from obstacles. A robot with optimal performance effectiveness would reach its goal location quickly, along a smooth path, while minimizing energy consumption and avoiding obstacles.

The performance effectiveness of a robot control architecture depends on the specific implementation and the specific task. Recall that deliberative architectures, such as that used for Shakey [Nil84], work best in predictable environments where a robot must perform complex tasks. Reactive architectures, such as Subsumption [Bro86], work best in dynamic, unpredictable, environments where the maintaining of internal representations does not help.

3.2.2 Brooks' Criteria

Brooks [Bro86] gives four criteria should must be satisfied by a control system for an autonomous robot. They are *multiple goals*, *multiple sensors*, *robustness*, and *additivity*. The following paragraphs discuss them in that order. Robustness was discussed previously in this section, so it will not be discussed again here.

Multiple Goals

A robot control system should be able to manage several concurrent goals, which may exist at different levels in a control hierarchy. For example, the goal of moving a robot to a goal location requires the subgoals of moving to waypoints along a path and avoiding obstacles. A control system must maintain a balance among the different goals it attempts to achieve at any time. It must eventually satisfy its high-level goals, but certain situations may require it to temporarily drop those goals to satisfy lower level goals. For example, if a large object, like a vehicle, quickly approaches the robot, then the robot should favor moving out of the path of the object.

Brooks claims that his Subsumption architecture [Bro86] has good support for multiple goals. In particular, each layer can work independently on its own goals. Architectures with hierarchical components, such as NASREM [AML87] and AuRA [AB97], have good support for managing goals at different levels. Architectures with behavior-based components, such as DAMN [Ros95] and AuRA [AB97], have good suppport for managing concurrent goals at the same level.

Multiple Sensors

Autonomous robots often have multiple sensors, and these sensors have uncertainties in their readings. A robot control system must work with thes uncertainties so that it can make better decisions. Sensor readings may have particular interpretations only in certain contexts. For example, vision sensors do not provide good distance measurements, using common techniques [OK93], in environments with reflection and insufficient texture. Brooks claims that his Subsumption architecture [Bro86] has good support for multiple sensors. In particular, each layer can process data, in its own way, from any sensor.

Additivity

Additivity refers to the addition of new sensors, actuators, capabilities, and processing units. A robot control system should be designed such that these new parts can be added without significant effort. Brooks claims that his Subsumption architecture [Bro86] has good additivity because different layers can execute on different processors. In addition, it can allow support for more sensors and actuators through additions to existing networks. Generally, systems with good support for modularity have good support for additivity.

3.2.3 Konolige's Criteria

Konolige and Myers [KM96] give three criteria for a mobile robot that must perform tasks in an "open-ended scenario". Their example of an "open-ended scenario" is one where a robot is led through a regular office building, with working people. In that scenario, the robot must learn to identify people and locations in the building, and then perform delivery tasks. Their three criteria, which they refer to as the "three C's", are *coordination*, *coherence*, and *communication*. The following paragraphs discuss, in order, these criteria.

Coordination

Control systems for autonomous robots must deal with functionality on many different levels. The lowest level consists of components that read data directly from sensors and send commands directly to actuators. Higher levels consist of components that plan and sequence tasks so that a robot can achieve its goals. The specific interactions between the different levels depend on the environment inhabited by the robot. In each environment, the robot may have different goals and different tasks for achieving them. Each task may require certain functionality from the low-level components. However, when a robot must perform several tasks at once, these low-level components must be properly coordinated such that, together, they aid in the completion of the tasks.

Coordination, in this context, is different from what was described in Section 2.3. Here, it refers to the structuring of a robot control system. Konolige and Myers [KM96] have determined that a "layered abstraction approach" is the best approach for structuring a control system for an autonomous robot. In this approach, high-level functionality is achieved through the proper coordination of lower level components. For example, moving a robot to a goal location requires the use of sensing components (for avoiding obstacles) and components that control the primitive motions. Architectures with hierarchical components, such as NASREM [AML87] and AuRA [AB97], have the this type of structure.

Coherence

As stated in Section 2.1, robot systems must combine deliberation and reactivity so that they can achieve complex goals while remaining reactive to unexpected events. For robots to survive in complex environments, they must model their environments as accurately as possible. Accurate environment representations allow robot control systems to make more informed choices to better achieve their goals. Thus, robot control systems must maintain a strong coherence between their internal representations and their physical environments.

While accurate, complex, representations allow robot systems to complete complex tasks in complex environments, the maintenance of these representations has negative impacts on a robot. In particular, the time required to construct a complete environment representation is not adequate when the robot must be able to immediately react to unexpected events. For example, if a dangerous object quickly approaches the robot, and the robot uses its complex representations to detect oncoming objects, then the robot may not react in time to avoid collision.

Following the above, robot control systems must maintain, along with their complex environment models, simple models based on immediately sensed data. These simple models allow for immediate reaction to sensed events. The proper balance of complex and simple environment models results in the required balance of deliberation and reactivity. Saphira [KM96], with its LPS, maintains environment representations at different levels of abstraction. NASREM [AML87], with its hierarchical sensory and world modeling modules, also allows for such representation.

Communication

Robots that operate in environments inhabited by humans should be able to communicate with those humans. In particular, they should be able to receive, and understand, commands and information given by humans. They should update their internal models based on the information given by humans, and they should use the commands given by humans to form goals that they must achieve. Besides understanding humans, a robot should be able to give information back to humans. This information includes the knowledge acquired by the robot, along with its current goals.

Complex speech recognition and synthesis systems provide the ideal level of communication between robots and humans. However, simpler communication protocols, such as gestures and typed text, are often adequate. For robots to communicate effectively with humans, they must be preprogrammed with basic concepts and language primitives. Each robot should, for example, have an understanding of its own identity, relative to people and other objects. Saphira [KM96] was developed with such communication as a requirement. Speech recognition and synthesis require a level of processing provided by common programming languages. This level of processing would be difficult, if not impossible, to achieve with architectures like Subsumption [Bro86], which use simpler semantics.

3.2.4 Simmons' Criteria

Simmons' criteria relate to the middle layer in a three-layered hybrid system. To reiterate, this middle layer takes sequences of tasks, from the deliberative layer, and instantiates the execution of those tasks at the reactive layer. To do this, task sequencers must interpret the state of a robot and its environment, and they must make decisions based on that state.

Simmons identifies four main areas of functionality that a task sequencer must provide [SA98]. These are *task decomposition*, *task synchronization*, *execution monitoring*, and *exception handling*. The following paragraphs describe these four criteria in detail, referring to the three task sequencer implementations noted in Section 2.4. Once again, the task sequencing mechanisms are RAPs [Fir89], Colbert [Kon97], and TDL [SA98].

Task Decomposition

Task decomposition allows a task sequencer to further break down sequences of tasks it retrieves from a planner. This division is important because tasks specified by the planner are often too abstract to map directly to low-level actions. Suppose, for example, that a task sequencer receives the task of moving the robot ahead by a particular distance. That task sequencer must break that task down into sequences of velocity commands for the robot. In a behavior-based system, the task sequencer would enable the behaviors required to control the robot motors.

For task decomposition, RAPs uses a Lisp-like language that specifies different alternatives for each task. Each alternative has a defined context under which it is applied, and it has constructs for starting sequences of subtasks. Colbert and TDL both use C-like languages, where developers write a different function to handle each task. These functions use conditional constructs ("if" statements) to decide which subtasks to start.

Task Synchronization

Task synchronization allows a task sequencer to execute subtasks and and low-level actions in series and in parallel. It is important because complex tasks often require the coordination of concurrently-executing low-level processes. In behavior-based systems, parallel execution allows different subsets of behaviors to execute for different tasks. RAPs, Colbert, and TDL all have constructs for starting subtasks in series and in parallel. More complex task synchronization is possible, which allows the task sequencer to apply time constraints to the execution of low-level processes. TDL provides a rich set of constructs based on time, such as those for terminating task execution after a specified time.

Execution Monitoring

Execution monitoring allows a task sequencer to await a specific condition before taking particular actions. For example, if the robot must wait for its batteries to be charged before moving, a task sequencer can wait for that condition to become true. In addition, if the robot must move to a specific location, a task sequencer can detect when that destination has been reached. RAPs have support for monitoring subtasks. Colbert and TDL allow developers to write specialized monitor functions, which are called repeatedly.

Exception Handling

Exception handling allows a task sequencer to quickly recover at higher levels, when lower level tasks fail. In object-oriented programming languages, such as Java, a method "throws" an exception when it encounters a situation that it cannot handle appropriately. When a method throws an exception, control flow returns up the method call chain to an ancestor that can "catch" the exception. Upon catching the exception, the catcher can take appropriate action, which depends on the specific application. Any time a task sequencer detects that it cannot perform a task, given its constraints, it should throw an exception.

An example of an exceptional situation is depicted in Figure 3.6, where a robot has a long wall between itself and its nearby goal location. This goal location is provided

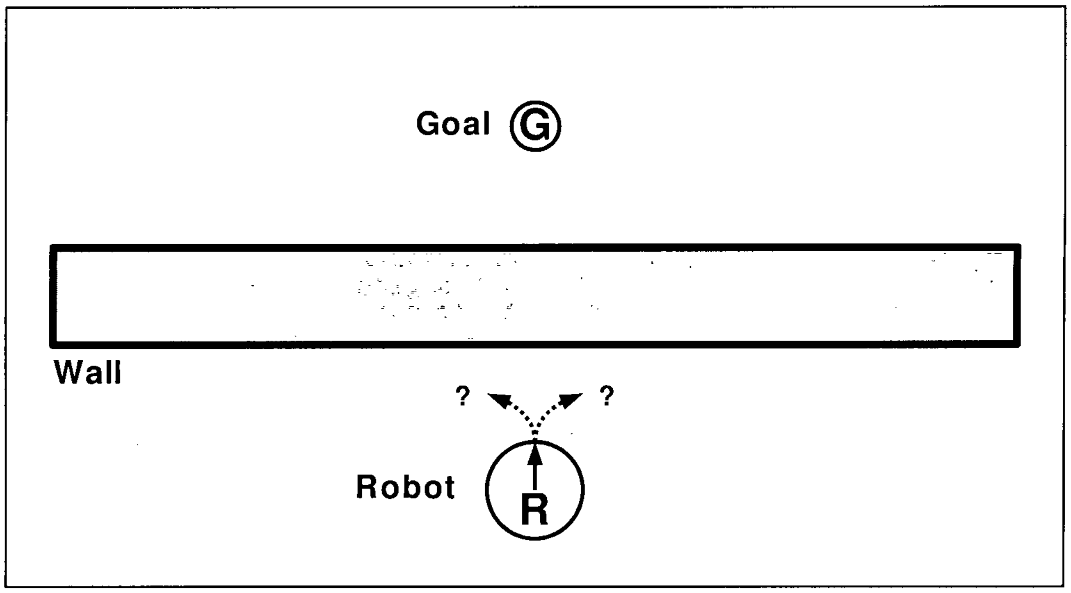

Figure 3.6: A Robot Caught in a Local Minimum

by a path planner, which computes sequences of waypoints a robot must pass to reach its ultimate goal. Suppose the robot has a reactive behavior that translates the goal location and nearby obstacles into attractive and repulsive forces, respectively. The exceptional situation arises from the robot being unable to make forward progress as a result of the attractive forces cancelling out the repulsive forces. In that case, the behavior can request that the path planner generates a new sequence of waypoints that can be reached.

RAPs provides exception handling in the sense that if a task fails, the RAP interpreter chooses another alternative for handling the task. Colbert does not provide explicit support for exception handling. TDL provides support for exception handling in a way similar to that provided by the Java programming language.

3.3 Summary

This chapter presented my criteria for evaluating robot control systems, which are based on criteria specified by other researchers and professionals. The chapter started with the general quality criteria and established object-oriented programming as a mechanism for developing high quality software systems. It continued with the robot-specific criteria, which indicate the types of structure and functionality my software framework and execution system should support.

Chapter 4

Framework Derivation

This chapter derives my software framework by resolving the specific issues that I believed were most important, based on the criteria outlined in Chapter 3. Recall that my software framework allows developers to specify robot control systems as networks of software components. This chapter, through its derivation, introduces the types of software components that are part of my framework. Figure 4.1 depicts a network that can be created with these types of components, where the arrows indicate the predominant direction of data flow between components. As stated in the introduction, this chapter derives my software framework at only a conceptual level, without giving specific programming language details.

In this chapter, Section 4.1 resolves the first issue, which is representing a robot and its environment. It introduces *state objects*, which facilitate the required representations. Section 4.2 resolves the issue of providing proxies for communicating with state objects over a computer network. It introduces *state object interfaces*, which are specialized interfaces for this purpose. Section 4.3 resolves the issue of providing access to physical sensors and actuators. It describes how state objects can provide this access.

Section 4.4 resolves the issue of providing reactivity. For this purpose, just as different architectures define behaviors differently, my software framework defines its own type of behavior. Not surprisingly, I refer to these behaviors as *behaviors*. Section 4.5 resolves the issue of updating internal representations. It describes how behaviors can be

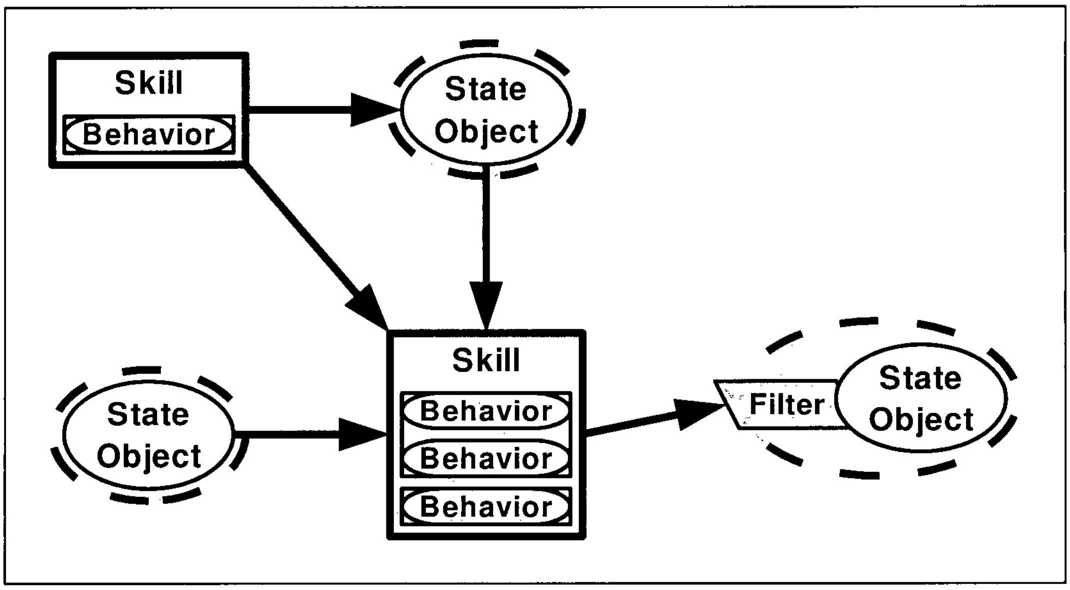

Figure 4.1: A Network of Framework Components

used to update the representations stored within state objects. Section 4.6 resolves the issue of coordinating competing modifications to state objects. It introduces *filters*, which are components that facilitate this coordination.

Section 4.7 resolves the issue of providing deliberation. It describes how behaviors can be given capabilities for planning and task sequencing. Section 4.8 presents the issue of decoupling task sequencers from specific target behaviors. This decoupling allows one or more target behaviors to replace an existing behavior without affecting its task sequencers. That section introduces *skills*, which force this decoupling by acting as abstractions to groups of behaviors. Skills, in this context, are not to be confused with skills as used in 3T [BFG+97] and other architectures. Section 4.9 summarizes this chapter.

4.1 Representing a Robot and its Environment: State Objects

A robot must maintain accurate representations of its own state and the state of its environment. As stated in Chapter 2, these accurate representations allow a robot to construct informed plans to achieve long term goals. Different alternatives exist for representing, and providing access to, the state of a robot and its environment. Saphira [KM96], as described in Section 2.4, uses its LPS to maintain environment representations at different levels of abstraction. José and Eric store occupancy maps, and other abstractions of sensor readings, within their shared memory systems. As explained in Section 3.1, object-oriented programming provides a good representation strategy.

Following the successful object-oriented approach to software development, my software framework uses objects to represent a robot and its environment. In particular, it provides *state objects*, which are components that store, and provide access to, the required data. In object-oriented programming terms, each state object is an instance of a class, where all state object classes are derived from a common class that is exclusive to state objects.

State objects, like other objects in object-oriented programming, provide access to their data through methods that other components can call. To prevent synchronization issues, a state object, using a locking mechanism, handles only one method call at a time. For increased integrity and looser coupling, they do not allow other components to directly access their member variables. State objects are passive components, which means that they do not initiate interactions with any other components. Instead, other components may freely initiate interactions with state objects, if they are connected within the component network, by calling their methods. Thus, state objects are self-contained components that do not depend, directly, on any other components.

Figure 4.2 depicts a sample state object, whose name is `object1`. The core part of the state object is represented by the inner ellipse, and it is an instance of class `C_`-

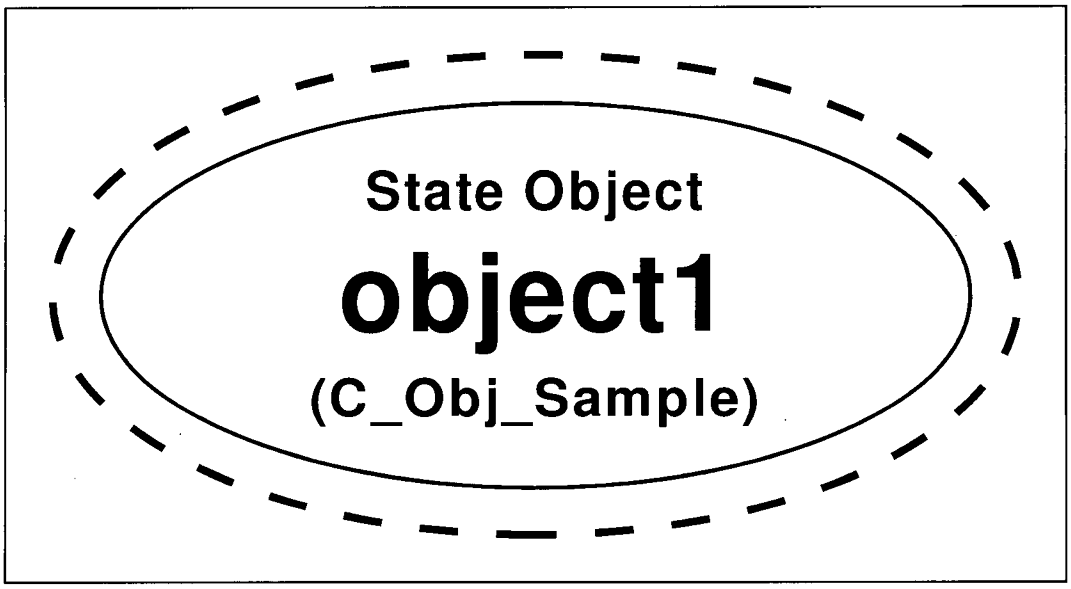

Figure 4.2: A Sample State Object

Obj_Sample. The outer ellipse, with the dashed boundary, indicates that the state object exists within a protective shell. Suppose that this state object holds a single integer value. Also, suppose that it provides two methods: one method for changing the value (called setValue) and one method for retrieving it (called theValue). This state object, with its two methods, will be referenced in the remaining sections of this chapter.

Because state objects are ordinary objects, from object-oriented programming, they inherit all the quality benefits described in Section 3.1. Because state objects are self contained, they have extremely high reusability. Their implementations can be reused within the same system, and within other systems, without dependencies on any other components. In addition, the fact that they do not initiate interactions with other components helps to simplify the possible communication pathways within a robot control system.

4.2 Providing Proxies for Communicating with State Objects: State Object Interfaces

Section 3.1 introduced the proxy pattern, which is useful for remote method invocation. Because state objects are ordinary objects, and they may exist on different computer systems than the components that call their methods, they can benefit greatly from the proxy pattern. To use the proxy pattern, state objects require superclasses that declare the common methods provided by the state object and its proxy. In my software framework, *state object interfaces* take on this role.

With my software framework, different components may call different subsets of the methods provided by a state object. I believe that, to minimize coupling and simplify usage, components should have access to only the state object methods they need. For example, if a state object stores the robot location and heading, then a component should not have access to the heading if only the robot location is relevant. Following this argument, my software framework allows each state object to have more than one interface. Each state object interface, then, declares a different subset of the methods provided by its corresponding state objects.

In a network of components, such as one defined using my software framework, developers may find it useful to specify, and visualize, the communication pathways between components. In particular, they may wish to visualize, for each component, which components it reads data from and to which components it writes data. For example, in reactive robot control systems, behaviors read input data from particular sensors and send commands to particular actuators. In that case, a developer may wish to visualize, using directed edges, the flow of data from sensor, to behavior, to actuator.

For better visualization and understandability, my software framework defines two main types of state object interfaces. These are the *read* interfaces and the *write* interfaces. *Read* interfaces declare methods that retrieve data from state objects. These methods, which facilitate the flow of data *out of* a state object, should not modify the state objects in any

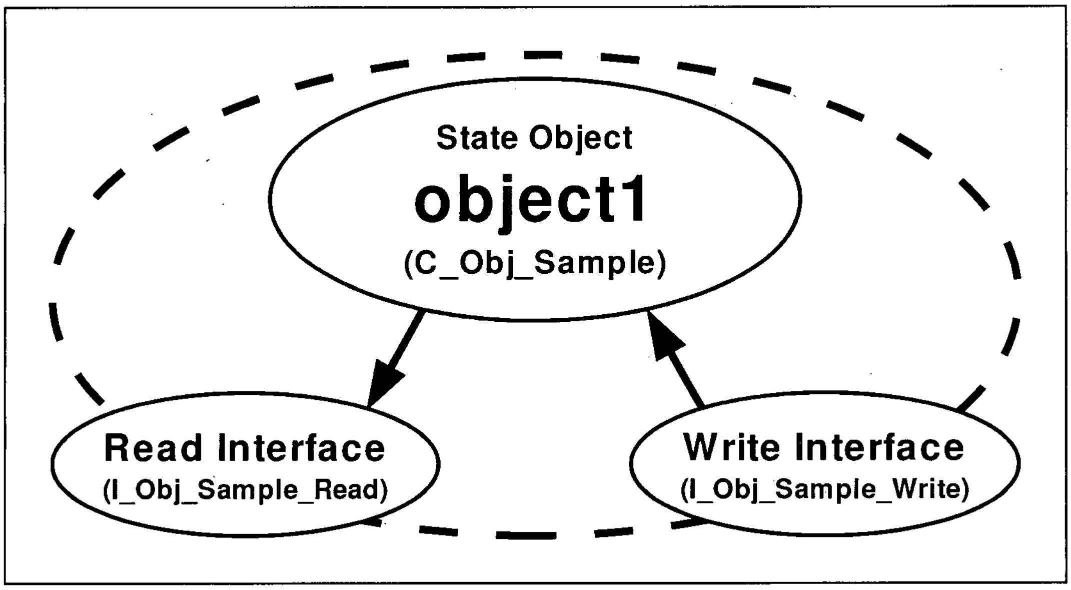

Figure 4.3: A State Object with Two Interfaces

way. *Write* interfaces declare the methods that may modify state objects. They, generally, facilitate the flow of data *into* a state object. Each state object may have associated interfaces of both types.

Figure 4.3 depicts state object `object1`, as described in Section 4.1. Here, this state object has two associated interfaces, where one is a read interface (`I_Obj_Sample_Read`) and the other is a write interface (`I_Obj_Sample_Write`). The arrow from the state object to the read interface indicates data flow out of the state object, and the arrow from the write interface to the state object indicates data flow into the state object. Suppose that the read interface declares method `theValue` and the write interface declares method `setValue`. This indicates that some components will be responsible for setting the stored value, and other components will only retrieve the value.

4.3 Providing Access to Physical Sensors and Actuators: State Objects

Access to physical sensors and actuators is crucial because my software framework is intended for controlling a physical robot. State objects can provide this access through methods that communicate with these sensors and actuators. Specifically, state objects for sensors can provide methods that return abstract representations of the data retrieved from sensors. For example, a state object could provide methods for reading images from a vision sensor. Likewise, state objects for actuators can provide methods that control actuators based on their arguments. For example, a state object could provide methods for setting the velocities for a robot.

State objects for sensors and actuators must have particular media and protocols for communicating with hardware components. These communication media and protocols generally depend on the specific robot configuration. State objects, with their interfaces, provide flexible mechanisms for adapting to configuration changes. Because state objects can take on any implementation that adheres to the same interfaces, appropriate changes can be made without affecting other parts of a robot control system.

4.4 Providing Reactivity: Behaviors

Robot control systems, to work effectively in unpredictable environments, require components that provide reactivity. In particular, these components, as stated in Section 2.3, must send appropriate commands to actuators, given sensor readings. Behaviors are commonly used for this functionality, where each robot control system defines behaviors in a different way. For example, Subsumption [Bro86], as stated in Section 2.4, defines behaviors as finite state machines. Saphira [KM96], in contrast, uses fuzzy control rules.

Recall that my software framework provides its own definition for behaviors and that I refer to these components as *behaviors*. For reactivity, behaviors must have the ability to read data from sensors and control actuators. Section 4.3 described how state objects can provide access to physical sensors and actuators. Thus, behaviors in my software framework communicate with sensors and actuators by calling methods on state objects. Like state objects, behaviors are ordinary objects, as defined in object-oriented programming, and each behavior is an instance of a class.

To respond quickly to environmental stimuli, behaviors in my software framework must frequently poll sensors for new data and compute new commands for actuators. To facilitate this repeated cycle, behaviors must provide a method that can be called repeatedly to perform this reactive functionality. In my software framework, this method is referred to as the *reaction method*. To communicate with sensors and actuators, the reaction method must take, as arguments, interfaces for the corresponding state objects. Generally, it must take read interfaces for sensor state objects, and it must take write interfaces for actuator state objects. In reality, behaviors take proxy objects as arguments, but to each behavior, each argument is an interface to a state object.

4.5 Updating Internal Representations: Behaviors

The representations of a robot and its environment are useless if no system components update them. Luckily, behaviors can take on this updating role. Section 4.4 showed how behaviors can call methods on state objects for accessing sensors and actuators. However, these state objects need not be those that provide methods for interacting with sensors and actuators. In fact, my software framework allows any behavior to take any kind of state object as a reaction method argument. Different behaviors may have different requirements with respect to how often their reaction methods must be called. For example, a path planning behavior may not need to execute as often as a behavior that sets robot velocities. Thus, my software framework allows for different call rates.

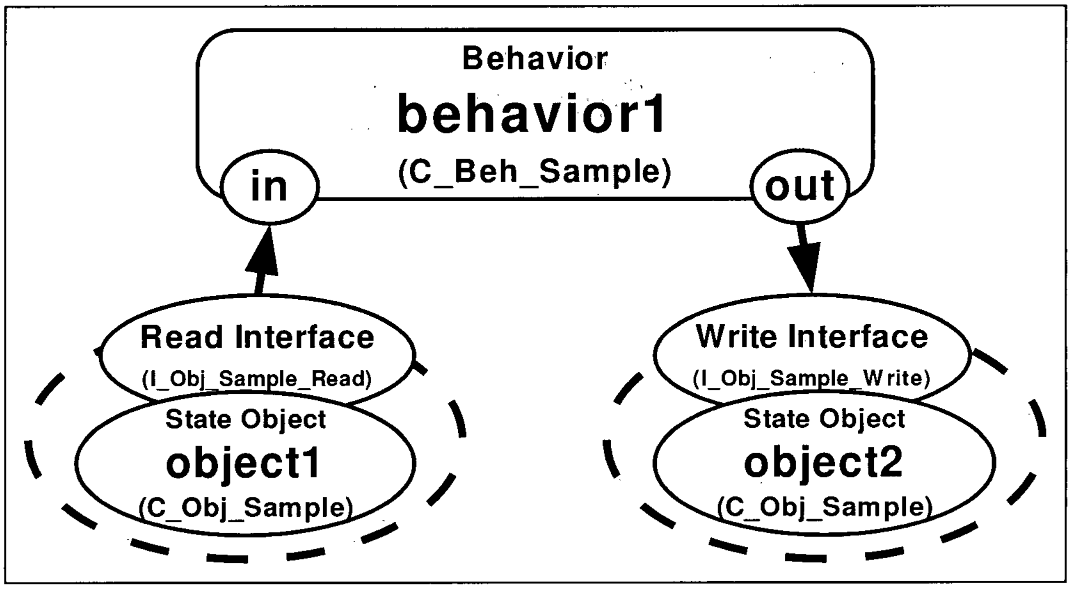

Figure 4.4: A Sample Behavior with Two State Objects

Figure 4.4 depicts a sample behavior, named `behavior1`, that is an instance of class `C_Beh_Sample`. This behavior has a reaction method that takes, through parameter `in`, a read interface to `object1`. The reaction method also takes, through parameter `out`, a write interface to equivalently-defined `object2`. The exact identities of `object1` and `object2`, which could be the same state object, are irrelevant to the behavior.

4.6 Coordinating Competing Modifications to State Objects: Filters

Because my software framework allows different behaviors to concurrently modify the same state object, the affected state objects require a coordination mechanism. Chapter 2 described coordination mechanisms that have been used in successful robot control systems. To reiterate, Subsumption [Bro86] uses signal overriding, DAMN [Ros95] uses vote-based arbitration, AuRA [AB97] uses vector summation, and Saphira [KM96] uses weighted blending. A robot control system should use a coordination mechanism that is suitable for its architecture and task requirements.

State objects can provide their own coordination functionality. However, because that would couple each state object with one specific coordination algorithm, it decreases the flexibility of that state object. In addition, it could, undesirably, require that state objects have the ability to distinguish among calling behaviors. A more flexible approach, for each state object, is to define a separate component that provides the required functionality. The state object, and the behaviors that interact with it, should have no dependence upon the existence of this component. Because communication with state objects is method call-based, this component must, then, intercept method calls and interact with the state objects accordingly.

In my software framework, *filters* are the components that implement coordination mechanisms. Filters are objects that, essentially, "filter" the arguments passed into a state object method. In the object-oriented programming sense, filters contain the same method declarations as their corresponding state objects. When a behavior attempts to call a method on an object that has an attached filter, the behavior actually calls the corresponding filter method. That filter method, if it desires, calls the underlying state object method with, possibly, modified arguments. When the method call is complete, the filter returns control to the calling behavior. Filtering of arguments, then, occurs in the sense that the filter checks each argument, does some processing, and calls the target method with new argument values.

To facilitate coordination, each behavior can have attached properties that correspond to each of its reaction method arguments. If a state object has a filter, then that filter can, for each method call, retrieve the property values for the calling behavior. Examples of useful properties are the behavior identifiers, the method call times, and the relative weights of the behaviors. The required selection of properties depends on the specific configuration of components. In any case, filters allow for the implementation of different coordination mechanisms, such as priority-based selection and weighted averaging.

Figure 4.5 depicts a state object (`object1`, as defined previously), whose methods can be called by two different behaviors (`behavior1` and `behavior2`). Both behaviors,

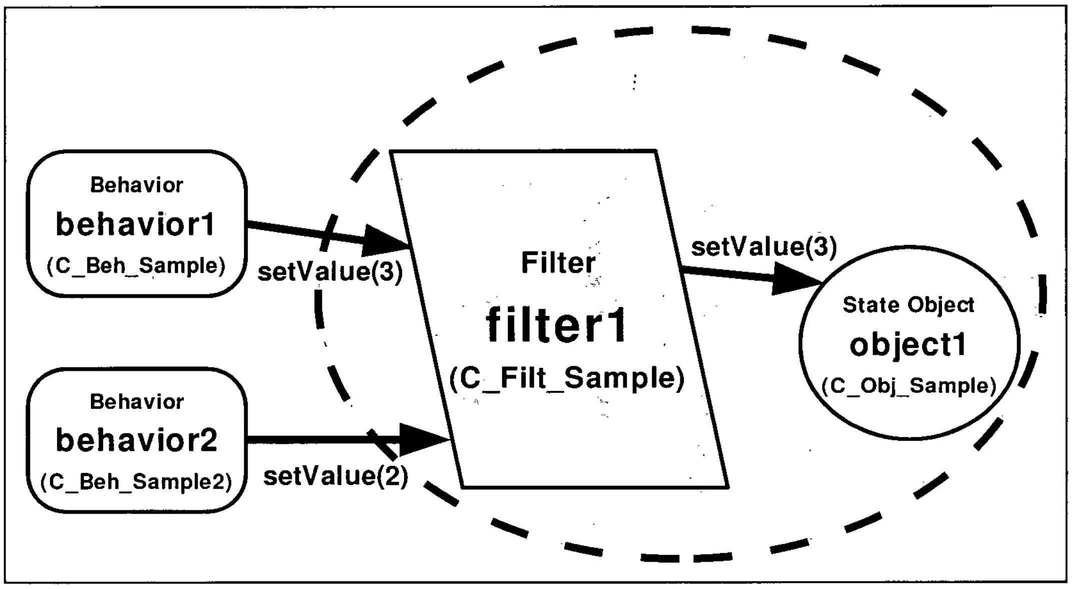

Figure 4.5: A Sample Filter in Action

through their write interface arguments (which have been omitted for simplicity), attempt to set the value stored by the state object. In particular, `behavior1` attempts to set the value to 3, while `behavior2` attempts to set the value to 2. They set these values by calling method `setValue`, where calls to that method are routed through filter `filter1`, an instance of class `C_Filt_Sample`. Although the diagram, for simplicity, indicates simultaneous calling of method `setValue` by both behaviors, the filter and state object handle one method call at a time.

Suppose that `filter1`, in Figure 4.5, uses behavior priority as its coordination mechanism. In addition, suppose that `behavior1` has priority over `behavior2` such that if `behavior1` had recently called `setValue`, the filter will block `behavior2` calls from reaching the state object. If `behavior1` calls `setValue` immediately before `behavior2`, then the filter prevents the state object from receiving the second method call. Hence, the state object receives only one method call (from `behavior1`), and it receives only one value (3).

For simplicity and efficiency, my software framework utilizes filters only for method calls through write interfaces. Read interfaces, which generally declare methods that do not modify state objects, can benefit from the reduced method call overhead. Besides coordination, filters can be used for shielding state objects from extraneous argument values. For example, consider a state object that allows behaviors to control the speed of a robot. If the behavior attempts to set the speed to an unsafe value, then a filter can recognize that and pass a safer value to the state object. By providing this protection, filters facilitate robustness.

4.7 Providing Deliberation: Task Sequencing Behaviors

At this point in the derivation, my software framework supports reactivity through the interactions of behaviors and state objects. Simple, reactive, tasks can be performed by collections of behaviors executing concurrently. The derivation did not, however, describe how my software framework can support deliberation, which allows a robot to complete tasks that are more complex. To support deliberation, my software framework must allow developers to define components that do high level planning. These components must interact with the state objects to access representations of a robot and its environment. More importantly, my software framework must allow developers to define task sequencing components that control behavior execution.

My software framework facilitates deliberation through the collaboration of behaviors and state objects. Certain behaviors can take low level data representations and convert them to representations that are suitable for high level planning. Other behaviors can break tasks down into sequences of subtasks that can be performed by lower level behaviors. At the lowest level, behaviors can convert subtasks into commands for controlling actuators. In essence, behaviors and state objects work together to complete the SMPA cycle, as described in Section 2.2.

Behaviors can construct plans to achieve goals by computing sequences of subtasks and updating state objects that store the current sequences of subtasks. Behaviors can do task sequencing by reading these subtasks and deciding which reactive behaviors they require. Task sequencing behaviors can control the execution of other behaviors in two main ways. Both ways, which are based on the information stored in polled state objects, have severe drawbacks.

The first way a task sequencing behavior can control another behavior is by updating a state object that identifies the current task. The target behavior can repeatedly poll this state object for new task requests and act accordingly. The drawback with this option is that it requires each target behavior to be encoded, at design time, with knowledge of all the possible tasks. In addition, for behaviors to support new tasks, developers must change the behavior implementations. To decrease unnecessary coupling, a different component should control the response of a behavior to each task request.

The second way a task sequencing behavior can control another behavior is by deciding, themselves, exactly which behaviors must execute for each task. The task sequencing behavior, then, updates state objects that indicate exactly which behaviors should execute. For flexibility, my software framework must allow developers to easily add new behaviors to a robot control system. Thus, it is impossible for a task sequencing behavior to anticipate, in its implementation, the exact behavior implementations that it requires. Nonetheless, task sequencing behaviors require enough information about their target behaviors so that they can make decisions about them. In addition, they must enable and disable target behaviors in a way that does not couple them with specific behaviors.

Unlike reactive behaviors, certain behaviors may not require repeated execution to complete their tasks. Instead, each time a behavior is enabled by a task sequencer, it may require only a single execution of its reaction method. For example, a behavior that makes a robot emit a sound need only execute when it receives that task request. Once it has emitted the sound once, it need not repeat. Similarly, a reaction method could, possibly, need recalling until a particular condition becomes true. My software framework allows behaviors to decide, within each reaction method invocation, whether their execution will repeat. If a behavior decides not to repeat, it signals that it has completed its task. Task sequencing behaviors should have the ability to recognize this completion.

4.8 Decoupling Task Sequencers from Specific Targets: Skills

The previous section noted that behaviors can act as task sequencers, but it presented solutions that had low quality. These solutions resulted in unnecessary coupling between task sequencing behaviors and their target behaviors. An additional option is to use interfaces for behaviors, but that does not allow a task sequencer to reference multiple behaviors as if they were a single behavior. This section introduces *skills*, which are components that help to reduce the unnecessary coupling. After introducing skills, this section describes how robot control systems use them for task sequencing.

4.8.1 Overview of Skills

A *skill*, in essence, is a general notion of a robot ability. This ability is described by the types of data required for input and the types of generated output. For example, a skill for a vision-based mobile robot may take vision data as input and compute motion commands for output. A skill does not describe, in any way, *how* it processes the data. It may, for example, use the vision data for sensing, and moving away from, obstacles. Instead, the skill may ignore the vision data and attempt to keep the robot moving forward. In my software framework, behaviors provide the specific functionality for a skill.

Each skill contains a collection of functionally-similar behaviors. The behaviors in each skill share a common superset of state objects that they interact with for input and output. My software framework enforces that each behavior instance is assigned to a skill, where that skill allows the state object accesses required by the behavior. That is, the state objects connected to the skill provide the interfaces required by the behavior. Each behavior defines its required state object accesses with the parameters of its reaction method.

Consider a reaction method for an obstacle avoidance behavior, where this method declares two parameters. The first parameter for this method requires a read interface for a state object that provides access to a vision sensor. The second parameter requires a write interface for a state object that provides access to an actuator for controlling motion. The behavior must be added to a skill that allows input from a vision sensor, through the same read interface, and, likewise, allows output to a motion actuator.

The general ability of each skill is defined with a collection of state object interfaces. In particular, each skill has ports, where each port specifies a specific state object interface. Within each skill, each reaction method parameter, from each behavior, maps to a particular port. In addition, each port connects to a state object that supports the interface specified by the port. Thus, all connections between behaviors and state objects are routed through specific skill ports.

Figure 4.6 depicts a sample skill. This skill defines two ports, `in` and `out`, where port `in` takes read interface `I_Obj_Sample_Read` and port `out` takes write interface `I_Obj_Sample_Write`. The skill also defines two behaviors, `behavior1` and `be-havior2`, where each of their reaction method parameters map to a specific port. These mappings are depicted by arrows, and their directions depend on whether the given port takes a read interface or a write interface. Parameter `in` in `behavior1`, for example, maps to port `in` for reading. Parameter `obj` in `behavior2` maps to port `out` for writing.

By restricting the behavior inputs and outputs to those allowed by the enclosing skill, my software framework minimizes the associations between behaviors and state objects. Thus, it helps to minimize coupling between behaviors and state objects. Without these restrictions, behaviors would be able to take, into their reaction methods, interfaces for any state object. This freedom to choose any state object could overwhelm the developer who implements the behaviors. Along with the restrictions on reaction method parameters, the routing of connections through skill ports also help to minimize the coupling between behaviors and state objects. Skills help to promote cohesion by grouping behaviors into functionally-similar units. Thus, skills act as abstractions to groups of behaviors.

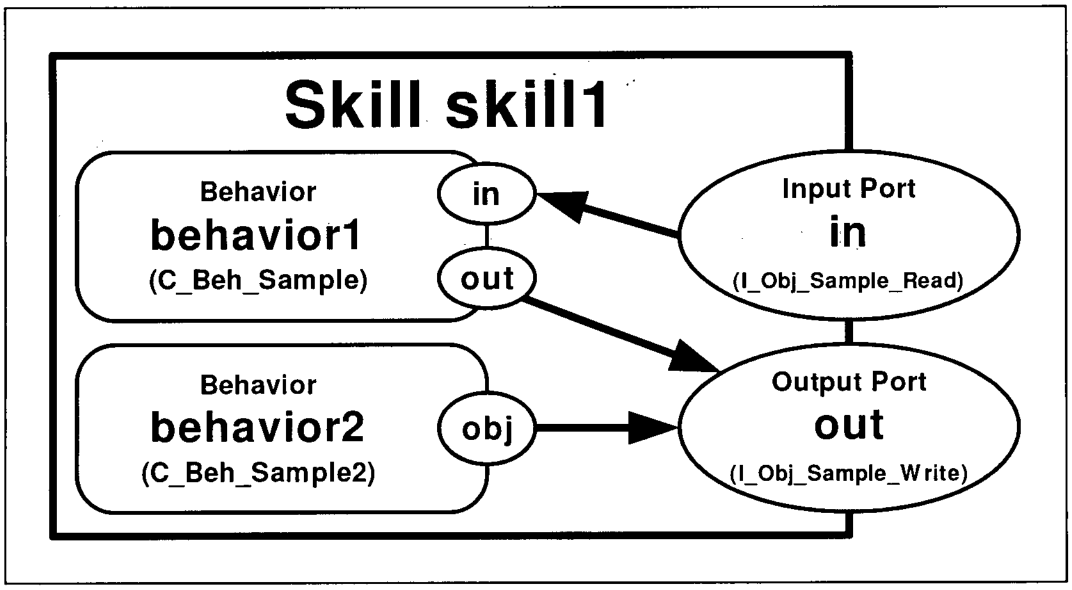

Figure 4.6: A Sample Skill

4.8.2 Task Sequencing with Skills

Within a skill, subsets of behaviors can execute concurrently to perform tasks. For example, a skill may have both a goal-seeking behavior and an obstacle avoidance behavior that, together, move a robot to its goal location while avoiding obstacles. However, the skills require a task sequencing component that selects individual behaviors for execution. Section 4.7 described how robot control systems, developed using my software framework, can use behaviors for task sequencing. However, it acknowledged that task sequencing behaviors should not, themselves, be coupled with individual target behaviors. Instead, my software framework couples task sequencing behaviors with skills, which are coupled with the target behaviors.

In my software framework, each behavior in each skill is assigned a specific task under which it will execute. Each behavior may have a different task, and a skill may have several behaviors assigned to the same task. Task sequencing behaviors, instead of interacting, directly, with their target behaviors, send task requests to skills. In particular, each task sequencing behavior specifies, to its target skill, which task that skill should perform. When a skill accepts a task request, it enables the behaviors assigned to the specified task and disables the remaining behaviors. The introduction of skills to my software framework yields increased flexibility because the number of skills is less than the number of behaviors, and the available tasks are less likely to be changed.

For simplicity, task sequencing behaviors interact with individual skills in the same way that they interact with state objects. That is, task sequencing behaviors call methods through skill interfaces, and their reaction methods take, as arguments, these interfaces. My software framework does not make any explicit distinction between task sequencing behaviors, which interact with skills, and behaviors that do not interact with skills. Both types of behaviors can take regular state object interfaces as their reaction method arguments. In addition, both types can be assigned to a single skill.

All skill interfaces share the same method declarations, and all skills share the same implementation. Therefore, developers do not supply special skill implementations. Through a skill interface, task sequencing behaviors can query the set of tasks that can be performed by the skill. They can also query whether the skill can perform a specified task, which is indicated by the presence of behaviors assigned to the task. Task sequencing behaviors can specify which task a skill will execute, and they can retrieve the identity of the task that is currently executing. In addition, they can query whether the currently executing task has completed its work. Task completion is signalled by the completion of all behaviors assigned to the executing task.

Figure 4.7 depicts a behavior setting a task for a skill. In particular, the behavior (`taskSequencer`) attempts to set the current task in skill `skill2` to the task named `"T1"`. Skill `skill2` defines two tasks: `"T1"` and `"T2"`. Task `"T1"` has two associated behaviors, `behavior11` and `behavior12`, which are enabled whenever the current task is set to `"T1"`. Likewise, task `"T2"` has one associated behavior, `behavior21`, which is enabled whenever the current task is set to `"T2"`.

A realistic robot control system could consist of dozens of skills, from those responsible for high-level planning, down to those responsible for low-level reactivity. In such a

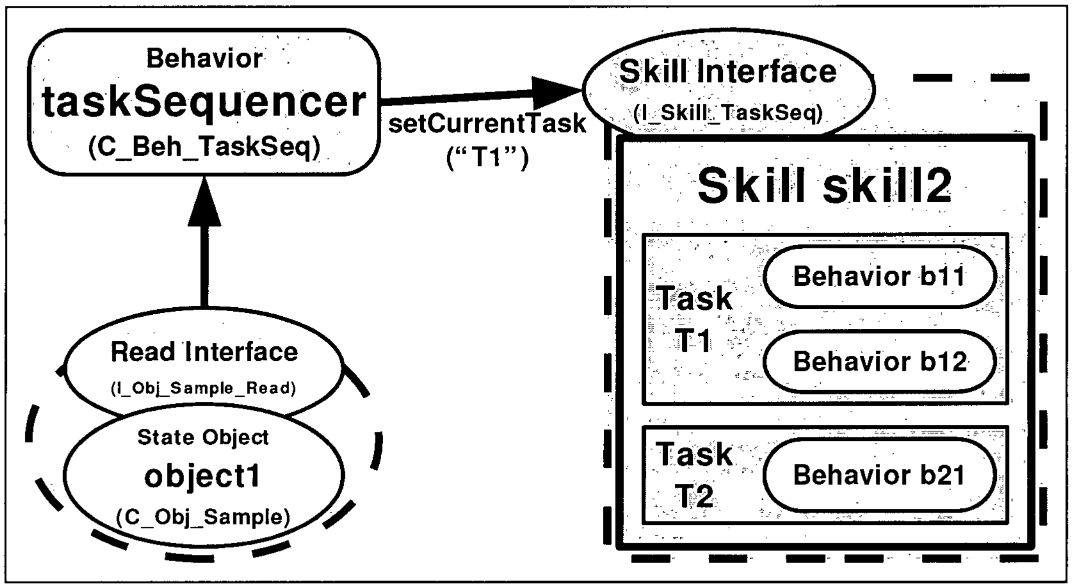

Figure 4.7: A Behavior Setting a Task for a Skill

system, a single task sequencer that controls all skills would be quite complex. Thus, task sequencing activities should be divided amongst several different task sequencing behaviors. With my software framework, developers can construct hierarchies of task sequencing behaviors. In these hierarchies, higher level task sequencers set tasks that result in the enabling of lower level task sequencers.

4.9 Summary

This chapter described my software framework, which allows developers to define robot control systems as networks of components. Each network can be represented as a directed graph, where the vertices correspond to the components, and the edges correspond to connections between components. The vertices can be either *state objects* or *skills*. *Filters* and *behaviors* augment these vertices, respectively. Communication between components takes place across edges and is based on method calls. *State object interfaces* define the allowable method calls.

State objects store, and provide access to, data within a robot control system. They may also provide access to external entities such as sensors and actuators. Skills are collections of behaviors, where all behaviors call methods on the same subset of state objects through the same interfaces. Method calls through *read* interfaces, which generally return data to the calling behavior, are represented by edges from the state object to the enclosing skill. Similarly, method calls through *write* interfaces, which generally modify state objects, are represented by edges from the skill to the state object.

Within a skill, each behavior is assigned to a particular task. Through skill interfaces, behaviors can sequence the tasks to be executed by other skills. This task sequencing control is represented by an edge between the task sequencing skill, which contains the task sequencing behavior, and the target skill. Each state object may have a filter, which intercepts method calls to the state object and modifies its arguments. Filters facilitate the coordination of competing modifications to state objects.

Chapter 5

Implementation Details

The development of a robot control system, using my software framework, requires the assignment of people to different roles. These roles are the *coder*, the *designer*, and the *user*. The *coder* uses a programming language to implement the functionality provided by the individual components, such as state objects and behaviors. The *designer* creates the model specification, which defines the robot control system in terms of its components and their interconnections. The *user*, through an execution system, starts and monitors the execution of the robot control system.

My software framework has two main parts, which, as stated in the introduction, I implemented using the Java programming language. First, it contains a collection of Java classes that coders extend to implement classes for individual components. Second, it contains a collection of Java classes for creating model specifications. These model specifications use the component classes to fully specify components and their interconnections. My execution system, also implemented in Java, has the ability to execute robot control systems defined using these model specification classes.

This chapter describes, in detail, the implementation of my software framework and execution system. Section 5.1, to provide context for the following sections, gives an overview of my execution system. Section 5.2 explains why I chose to use Java as the implementation language, and it describes the most useful features of Java. Section 5.3

explains how coders use my software framework to define components.

Section 5.4 describes the classes for creating model specifications, and it explains how designers use these classes to specify complete robot control systems. Section 5.5 describes the low-level details of my execution system, which includes intercomponent communication. Section 5.6 describes the high-level details of my execution system, which includes the main parts and user interaction. Section 5.7 summarizes this chapter.

5.1 Execution System Overview

Recall that the purpose of my execution system is to execute robot control systems that are defined in model specifications. The execution system consists of a collection of execution nodes, called *executors*, and a central controller, called the *execution manager*. Executors hold executing components, and the execution manager controls the execution of components in the executors. Figure 5.1 depicts a simple execution system. Note that in this implementation, skills execute separately from their enclosed behaviors, and they may each execute within a different executor. An arrow from a skill to a behavior denotes skill containment. Also note that, for efficiency, behaviors communicate directly with state objects, rather than through their parent skills.

All executors run in their own, separate, processes, which may exist on different computer systems. Likewise, the execution manager runs in a different process, which may exist on its own computer system. The execution manager provides the main point of contact for users of the execution system. Through the execution manager, users can load new model specifications, control the execution of the specified systems, and retrieve information about executing systems. Generally, users communicate with the execution manager through client applications, which may exist on different computer systems.

The execution manager and its executors are Java applications. Thus, users start them just like any other Java application. However, users must start the execution manager before any of its associated executors. After starting the execution manager, users can start executor processes, which register themselves with the execution manager. Users supply

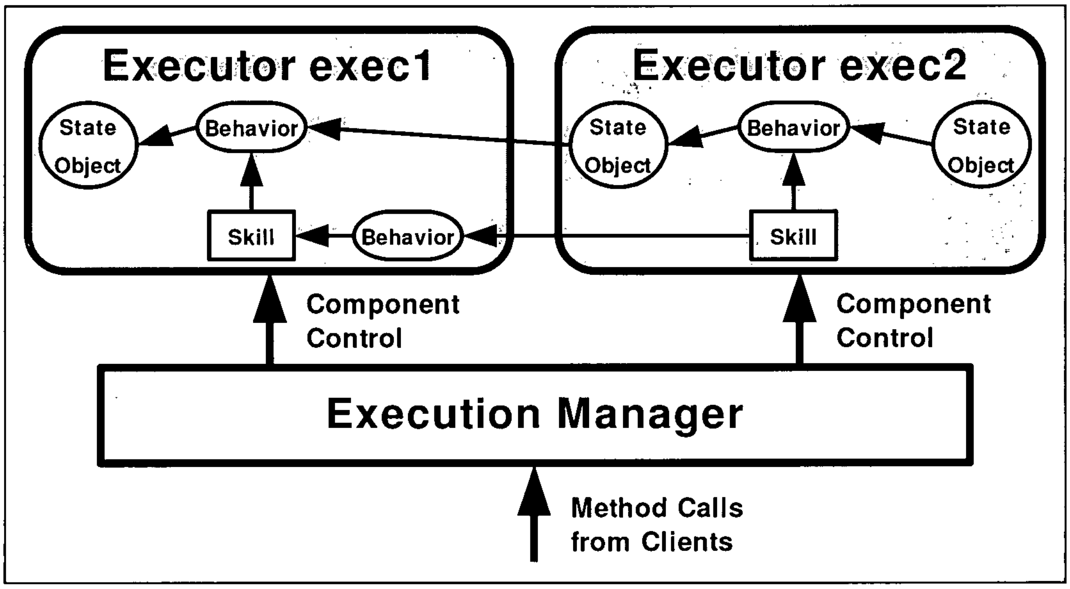

Figure 5.1: Execution System Components

each executor with a unique name that allows the execution manager to identify it.

When a user loads a new model specification into the execution system, the execution manager instructs the executors to create the required components. Each executor holds its own disjoint subset of these components. The execution manager distributes components to different executors based on preferences indicated in the model specification. After completing component creation, the execution manager connects the components, as indicated. After the execution manager connects the components, the specified robot control system is ready for execution.

5.2 Implementation Language: Java

I chose Java as my implementation language because I knew it well and because it has features that simplify the development of my software framework and execution system. This section describes the features of Java that I found most useful and are referenced in later sections of this chapter. It starts with the more basic features and continues with Remote Method Invocation (RMI) and Reflection.

5.2.1 Basic Features

Probably the most significant feature of Java that makes it suitable for implementing my software framework and execution system is that it is an object-oriented programming language. For my software framework, this feature allows coders to define behaviors, state objects, and filters, using Java classes. Within an executor, then, component instances are instances of these classes. Java allows coders to define special classes, called *interfaces*, where each Java interface formally declares the public methods for all of its subclasses. Each subclass of a Java interface is said to "implement" that interface. My software framework uses Java interfaces to define state object interfaces.

Java source code compiles into "byte code", which is executed using a Java Virtual Machine (JVM). Java byte code is stored within a Java *class file* (a file with a ".class" extension), where each class file contains all the information about a single Java class. In general, the JVM is a regular software application that interprets the byte code defined in class files. Because byte code is interpreted, the execution of a Java application is generally slower than the execution of an equivalent implementation that has been compiled into native code. However, with few exceptions, this slowness is outweighed by the fact that a single implementation of any Java application will execute equally on any computer system that provides a JVM. A significant exception to the portability of Java byte code arises with threading, which is partially handled by the underlying operating system.

Threading allows my execution system to execute multiple components concurrently, without requiring that the user start a separate Java application for each component. Most importantly, Java allows my execution system to start new threads, and it allows my execution system to schedule methods for execution at a fixed time rate. However, Java provides no accepted way to stop an executing thread. Thus, threads must be coded carefully such that they do not steal excessive processor time from other threads. If they cannot perform their functions quickly, they should be coded such that they frequently relinquish control to other threads.

Java, like many other programming languages, has a standard collection of classes that is available with every compiler. Specifically, each Java Development Kit (JDK) is distributed with the Java API (Application Programmers' Interface), which provides useful classes and methods. Different vendors, such as Sun Microsystems, release their own implementations of the JDK and API. My execution system implementation uses the Java API extensively. Within the API, collections of classes are arranged into *packages*, which allow classes to be organized by common function and help to prevent class naming conflicts.

Although my software framework and execution system are Java-based, the coder is not required to code exclusively in Java. The Java Native Interface (JNI) allows a Java application to invoke routines that have been written in other programming languages. Thus, the coder can make use of routines written in other languages without needing to rewrite them in Java. Also, the native code runs more efficiently than Java byte code because it does not require interpreting by a virtual machine. However, if the routines require the passing of anything more complex than primitive types, the code on the native side must do extra work to put the data into a usable form. This extra processing can often outweigh the speed benefits.

5.2.2 Remote Method Invocation

Remote method invocation in Java allows methods in one virtual machine to invoke methods on objects that may exist in another virtual machine. Such functionality is required in my execution system because it allows components to exist within different executors and still be able to communicate with each other. Java's RMI system provides an easy way to facilitate this functionality.

RMI is made possible through the *proxy* pattern, which was described in Section 3.1. When the method caller invokes a remote method, it really invokes a method defined in a proxy object. This proxy object is known as the *stub*, and it exists within the same virtual machine as the method caller. The stub, via a socket connection, sends the required information, such as argument values, to a receiving object in the target virtual machine. This receiving object is known as the skeleton, and it makes the actual method call on the

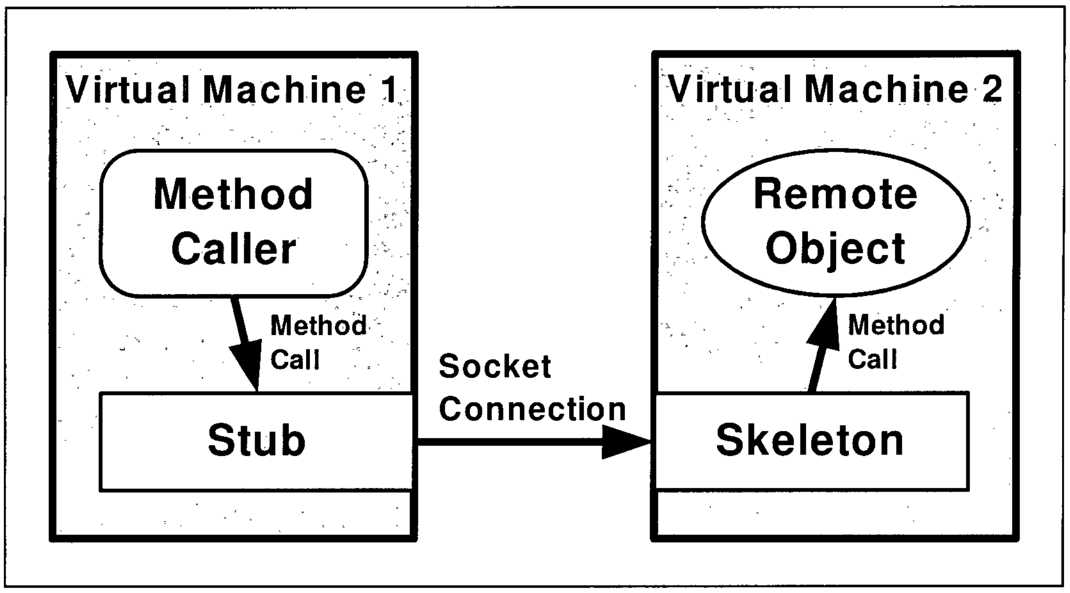

Figure 5.2: Remote Method Invocation in Java

target object. When the method call is complete, control returns to the method caller, with any return values. Figure 5.2 depicts the execution of a remote method call in RMI.

For an object to have its methods accessed remotely, the coder must define its class in a certain way. In particular, using classes defined in package `java.rmi`, the coder must write the class for the remote object, plus the interface that it shares with its stub object. In addition, coders must use a separate compiler (`rmic`) to generate classes for the stub and skeleton objects. In an executing system, a method caller must retrieve the stub required to call its desired remote method. In general, the process that creates the remote object registers the stub with a naming service. However, that process can, instead, retrieve the stub directly and send it to the method caller. The stub has all the information required to locate the skeleton, which listens for socket connections on a network port.

5.2.3 Reflection

Reflection allows a Java application to gather information about the classes and interfaces that it uses. This information is accessible through an object of class `Class` (in package java.lang), and it includes a listing of the methods and variables that are members of the class or interface. In addition, it includes the access specifiers (public, private, etc.) and the superclass information, among other class properties. For each method, Reflection allows an application to retrieve information about the parameter types and the return type. It also allows an application to retrieve information such as exception types, the access specifier, and other modifiers. For each member variable, Reflection allows an application to retrieve information such as the variable type and access specifier.

In general, Reflection is useful because it allows an application to create, and use, an object of a particular class without the name, and implementation, being known at compile time. An application can, then, use the Reflection facilities to create an instance of that given class and then assign that object to a variable of a superclass type. This allows the application to interact with the object normally, as an instance of the superclass. If necessary, an application can use the Reflection facilities to call particular methods on objects. My software framework uses Reflection to create objects and call methods, as described, as well as to generate specialized classes that hide communication details from coder-supplied classes.

5.3 Component Coding

Coders define state objects, filters, and behaviors, by writing Java classes that extend classes from Package RobotControl.Framework. Designers can define several components, in model specifications, using the same class, where each component has its own instance of the class. Package RobotControl.Framework also contains interfaces for creating state object interfaces, and it contains an interface that allows behaviors, and system users, to control skills. This section describes how coders use the classes and interfaces in that package to define the software components in a robot control system.

5.3.1 State Objects

In my software framework, coders implement state objects using Java classes that extend class `C_AbstractObject`. Besides that distinction, state objects must satisfy two other requirements. First, all non-primitive parameter types and return types in all public methods must be descendants of class `Serializable` (in package `java.io`). This ensures that their values can be properly passed between a state object and the calling behaviors. Second, the no-argument constructor of the class must be publicly accessible so that the execution system can create an instance of the defined class. This requires that the coder writes no constructors at all, such that the no-argument constructor becomes implicit, or that the coder explicitly provides that constructor.

Figure 5.3 gives source code for a sample state object implementation. This code defines a class, called `C_Obj_Sample`, for a state object that stores a value. This class provides method `setValue`, which changes the stored value, and method `theValue`, which returns the stored value. Note that this code, except for its extension of `C_Abstract-Object`, gives no hint that it will be used to define remote objects. Also note that the defined class does not implement any interfaces that identify the publicly-accessible methods. This exclusion does not break the proxy pattern, assuming that each proxy object is coupled, in some way, with a specific state object class. Specific interfaces are assigned in the model specification, which makes it easier for designers to add new interfaces to a state object.

5.3.2 State Object Interfaces

In my software framework, coders can implement three types of interfaces for state objects. Two types of interfaces, the *read* interfaces and the *write* interfaces, were described in Section 4.2. The third type of interface, called the *user* interface, declares methods that are available to external client applications so that users can inspect and modify state objects. Read interfaces must extend interface `I_Object_Read`, write interfaces must extend interface `I_Object_Write`, and user interfaces must extend interface `I_Object_User`.

```
import RobotControl.Framework.C_AbstractObject;

public class C_Obj_Sample extends C_AbstractObject {
    public void setValue( int value ) {
        _value = value;
    }
    public int theValue() {
        return _value;
    }

    private int _value = 0;
}
```

Figure 5.3: A Sample State Object Implementation

```
import RobotControl.Framework.I_Object_Read;

public interface I_Obj_Sample_Read
                            extends I_Object_Read {
    public int theValue();
}
```

Figure 5.4: A Sample Read Interface Implementation

Figures 5.4, 5.5, and 5.6 give source code for state object interface implementations. Each of the defined interfaces may be used to communicate with the sample state object defined in Figure 5.3. Figure 5.4 gives a read interface implementation that declares method `theValue`. Figure 5.5 gives a write interface implementation that declares method `setValue`. Figure 5.6 gives a user interface implementation that declares methods `setValue` and `theValue`.

```
import RobotControl.Framework.I_Object_Write;

public interface I_Obj_Sample_Write
                                extends I_Object_Write {
    public void setValue( int value );
}
```

Figure 5.5: A Sample Write Interface Implementation

```
import RobotControl.Framework.I_Object_User;

public interface I_Obj_Sample_User
                                extends I_Object_User {
    public void setValue( int value );
    public int theValue();
}
```

Figure 5.6: A Sample User Interface Implementation

5.3.3 Filters

Coders implement filters using Java classes that extend class `C_AbstractFilter`. Filter classes do not have any other restrictions, except that their no-argument constructors must be publicly accessible. During a method call to a state object, through a write interface, the execution system will call the corresponding filter method, if it exists. This filter method must have the following format.

```
public <return_type> <method_name>( <parameters>,
                <method_call_info>, <state_object> )
```

Each filter method corresponds to a state object method with the same name, where this name is given by `<method_name>`. The filter method must have the same return type as its corresponding state object method (`<return_type>`). In addition, the filter

```
import RobotControl.Framework.C_MethodCallInfo;

public class C_CallInfo_Sample
                              extends C_MethodCallInfo {
    public void setMultiplierValue( int multiplier ) {
        _multiplier = multiplier;
    }
    public int theMultiplierValue() {
        return _multiplier;
    }

    private int _multiplier = 1;
}
```

Figure 5.7: A Sample Method Call Information Class

method must declare the same parameters (`<parameters>`) at the same locations in its parameter list. Besides these corresponding parameters, the filter method must declare two extra parameters. The first parameter (`<method_call_info>`) requires an object that contains information about the method caller. The second parameter (`<state_object>`) requires the state object that is the target for the method call.

The object that contains information about the method caller allows a filter to handle method calls differently for each caller. That object is an instance of a class that extends class `C_MethodCallInfo`. Its class has no other restrictions, but it should declare methods for retrieving relevant information about each method caller, which may include identifiers and weights. A filter can use this information to aid in coordination. Figure 5.7 gives source code for a method call information class. This class (`C_CallInfo_Sample`) is for an object that stores a multiplier value. It is intended that filter methods multiply this value against their argument values to produce new arguments for the state object method.

Figure 5.8 gives source code for a sample filter implementation. Instances of the defined class (`C_Filt_Sample`) may be used to create filters for the sample state object defined in Figure 5.3. In particular, the filter class contains a method that is called in place

```
import RobotControl.Framework.C_AbstractFilter;

public class C_Filt_Sample extends C_AbstractFilter {
    public void setValue( int value,
                          C_CallInfo_Sample callInfo,
                          C_Obj_Sample object ) {
        int newValue = value *
                       callInfo.theMultiplierValue();
        object.setValue( newValue );
    }
}
```

Figure 5.8: A Sample Filter Implementation

of state object method `setValue`. This filter method applies the multiplier value, as described in the previous paragraph, to the input argument value, and it calls the state object method with the new value.

5.3.4 Behaviors

Coders implement behaviors using Java classes that extend class `C_AbstractBehavior`. As with state objects and filters, each behavior class must have a publicly-accessible no-argument constructor. In addition, each behavior must provide a properly-defined reaction method, called `doReaction`, which is called by the execution system. Optionally, a behavior class may override methods `onEnable` and `onDisable`, where both methods take no arguments and return no value. These optional methods allow behaviors to do any initialization and cleanup, respectively, when they are enabled and disabled.

Each reaction method declares a list of parameters, where each parameter may take a state object interface or a skill interface. State object interface arguments must be either read interfaces or write interfaces. Skill interfaces are have type `I_Skill_TaskSeq`, and they declare methods that allow a behavior to control the corresponding skill. Each reaction method returns a `boolean` value that indicates whether the execution system

```
import RobotControl.Framework.C_AbstractBehavior;

public class C_Beh_Sample extends C_AbstractBehavior {
    public boolean doReaction(
                        I_Obj_Sample_Read objIn,
                        I_Obj_Sample_Write objOut ) {
        objOut.setValue( 2 * objIn.theValue() );
        return true;
    }
}
```

Figure 5.9: A Sample Behavior Implementation

should continue to call it for the current task. If this method returns `false`, the execution system will not call it again until its task is reset. This return value accounts for the fact that some tasks may consist of a single action that occurs once and stops, and other tasks consist of an action that is continuously repeated.

Figure 5.9 gives source code for a sample behavior implementation. The behavior class, `C_Beh_Sample`, declares a reaction method with two parameters. The first parameter, `objIn`, takes an interface as defined in Figure 5.4. The second parameter, `objOut`, takes an interface as defined in Figure 5.5. The reaction method sets its output value (`objOut`) to be twice its input value (`objIn`). It always returns `true`, which indicates that the execution system can continue to call it for the current task.

Figure 5.10 gives source code that shows the usefulness of overriding method `onEnable`. In this example, class `C_Beh_Sample2` declares a reaction method with one parameter. This parameter, `objOut`, takes an interface as defined in Figure 5.5. The reaction method updates `objOut` with its current counter value (in `_counter`) and then increments that counter value. It returns `true` until the counter value reaches 100. Method `onEnable` is responsible for resetting the counter value to 0 at the beginning of a new task. The defined behavior, essentially, counts from 0 to 99 for its task.

It is important to note that behavior implementations do not require any special net-

```
import RobotControl.Framework.C_AbstractBehavior;

public class C_Beh_Sample2 extends C_AbstractBehavior {
    public void onEnable() {
        _counter = 0;
    }

    public boolean doReaction(
                         I_Obj_Sample_Write objOut ) {
        objOut.setValue( _counter );
        ++_counter;
        return ( _counter < 100 );
    }

    private int _counter;
}
```

Figure 5.10: Another Sample Behavior Implementation

working code to communicate with state objects. As with state objects, remote method call functionality is handled by the execution system. This allows behavior implementations to focus on core functionality. How the method call reaches the target state object is irrelevant to the behavior. However, if the target state object does not exist, or cannot be contacted, my execution system throws an exception of type `C_ObjectAccessException`, which can be caught by the behavior.

Inevitably, behavior code will be executed within a thread in a particular virtual machine. When the execution system disables a behavior, it allows the reaction method to return normally, if it is executing, rather than stop its execution in the middle of an operation. Thus, to not starve other threads, and to allow prompt thread stoppage, reaction methods should not execute for significant periods of time without returning. If a reaction method takes too long to execute, the coder should change its implementation such that it divides its functionality over successive reaction method invocations.

5.3.5 Skill Interfaces

Recall that behaviors can take skill interfaces as arguments, where these interfaces have type `I_Skill_TaskSeq` and declare methods for controlling a skill. This interface type is also available to client applications so that users can monitor and control task execution. Registration of behaviors with tasks is specified in the model specification and cannot be controlled through these skill interfaces. Each task has a name, which is a `String` that uniquely identifies the task within its skill. The following paragraphs describe the different methods that behaviors and system users can call through skill interfaces.

`String[] theTaskNames()`

> Returns the names of all defined tasks, where each task has at least one behavior registered with it. This method is useful if the task sequencer does not know exactly which tasks are available for execution. It is especially useful for system users, if they wish to control the skill, since it gives them a list of tasks to select from.

`void setCurrentTask( String taskName )`

> Sets the current task to that specified by the given name. If the specified task does not exist, or does not have any behaviors registered with it, then any current task will be deactivated. A task name that is `null` also results in task deactivation.

`String theCurrentTask()`

> Returns the name of the task that is currently set for execution, regardless of whether the task is complete. If no task has been set, this method returns `null`.

`boolean isTaskComplete()`

> Returns `true` if and only if the currently set task is complete. A task is complete if all its behaviors have completed their work, which is indicated by their reaction methods returning `false`. In addition, if no current task is set, task completion is assumed to be `true`. Task completion is a good indication, to the task sequencer, that it should set a new current task, or take a particular action.

`boolean canDoTask( String taskName )`

Returns `true` if and only if the specified task can be completed, given the current skill configuration. This method helps the task sequencer to decide which task it will set as the current task. If the skill cannot perform the given task (it does not have any assigned behaviors), the task sequencer can, instead, consider other tasks.

Figure 5.11 gives source code for a task sequencing behavior class named (`C_Beh_TaskSeq`). Its reaction method takes interface `objIn` as input and uses it to help choose a new task for `skill`. It makes its decision according to the following conditions, which it applies in the order given. If the stored value is greater than or equal to 0, it sets the task to `"T1"`. If the current task is `"T1"`, it sets the task to `"T2"`. If the skill can do task `"T3"`, the reaction method sets the task to `"T3"`. Finally, if the current task is complete, it sets the task to `"T1"`.

```
import RobotControl.Framework.C_AbstractBehavior;
import RobotControl.Framework.I_Skill_TaskSeq;

public class C_Beh_TaskSeq extends C_AbstractBehavior {
    public boolean doReaction( I_Obj_Sample_Read objIn,
                               I_Skill_TaskSeq skill ) {
        int value = objIn.theValue();
        if ( value >= 0 ) {
            skill.setCurrentTask( "T1" );
        }
        else if ( skill.
                    theCurrentTask().equals("T1") ) {
            skill.setCurrentTask( "T2" );
        }
        else if ( skill.canDoTask("T3") ) {
            skill.setCurrentTask( "T3" );
        }
        else if ( skill.isTaskComplete() ) {
            skill.setCurrentTask( "T1" );
        }
        return true;
    }
}
```

Figure 5.11: A Task Sequencing Behavior Implementation

5.4 Model Specification

I chose to define the model specification format using a hierarchy of composition, where specification layers are composed of other specification layers. Although many alternatives existed, I found that this particular representation, when implemented in Java, allowed the execution manager to process model specifications with greatest efficiency. Designers must define model specifications using Java source code, where this code references classes defined in package RobotControl.Model. A model specification, as accepted by the execution manager, contains collections of state object specifications, skill specifications, and connection specifications. The following paragraphs describe the model specification

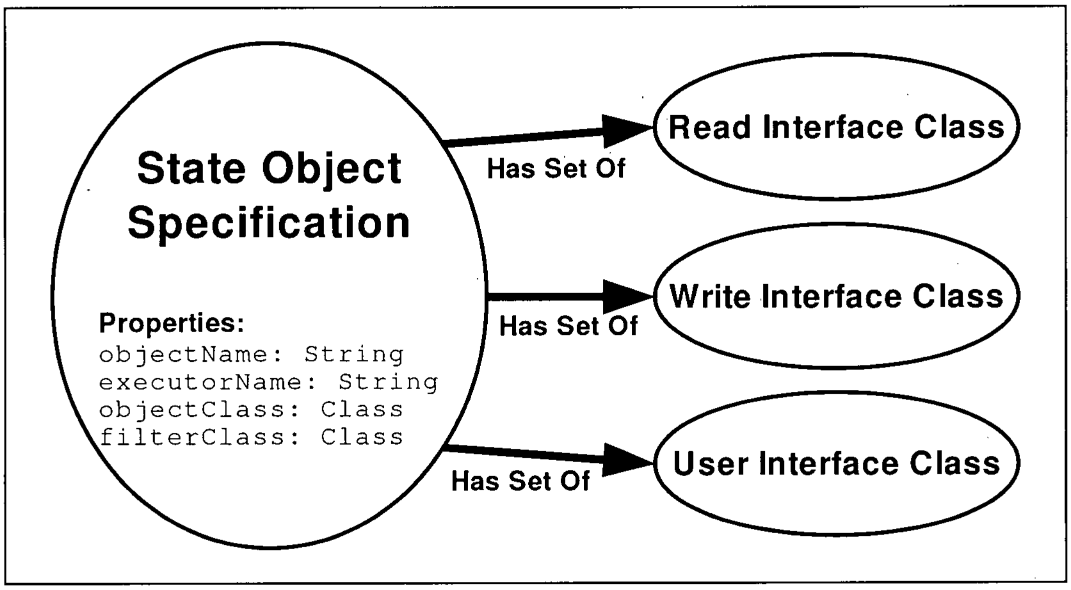

Figure 5.12: The State Object Specification

format in more detail, starting with the descriptions of its subparts, and ending with the model specification as a whole.

5.4.1 State Object Specification

Figure 5.12 depicts the composition of a state object specification. Each state object specification defines a single state object, with its interfaces and optional filter. More specifically, each state object specification has the following four properties, which must be configured appropriately. First, it has a unique name (`objectName`) that allows the specified state object to be distinguished from other state objects. Second, it has the name of the preferred executor (`executorName`) that will create and hold the state object. Third, it has the state object class (`objectClass`), which provides the coder-supplied implementation of the state object. Finally, it has the filter class (`filterClass`), which may be `null` if the state object does not have a filter. Along with these properties, it defines the read, write, and user, interfaces that allow communication with the defined state object.

Each state object specification, in Java source code, is an instance of class `C_`-

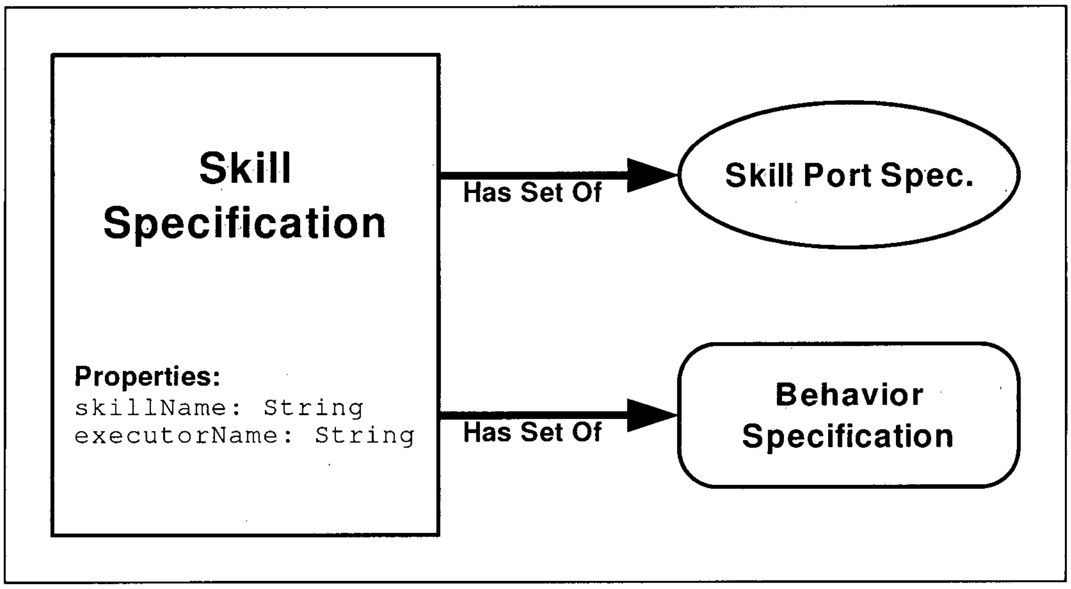

Figure 5.13: The Skill Specification

`ObjectSpecification`. This class provides the methods for configuring, and inspecting, the different parts of a state object specification. For organization purposes, the constructor provides the only way to set the state object name.

5.4.2 Skill Specification

Figure 5.13 depicts the composition of a skill specification. Each skill specification defines a single skill, and it consists of four main parts. First, it has a unique name (`skillName`) that allows the specified skill to be distinguished from other skills. Second, it has the name of the preferred executor (`executorName`) that will create and hold the skill. Third, it has a set of port specifications, which defines the possible connections with other components. Finally, it has a set of behavior specifications, where each specified behavior is part of the specified skill.

Each skill specification, in Java source code, is an instance of class `C_Skill-Specification`. This class provides the methods for configuring, and inspecting, the different parts of a skill specification. Similar to state object specification class, and all other named specification classes, the constructor provides the only way to set the skill name. The following paragraphs describe the port specification, and the behavior specification, in more detail.

Port Specification

Each port specification has a name (`portName`) that is unique in relation to the other port specifications from the same skill. Each port specification also has an interface class, which defines either a read interface, a write interface, or a skill (task sequencing) interface. Each port specification, in Java source code, is an instance of class `C_SkillPort-Specification`, and this class defines methods for setting these two properties.

Behavior Specification

Figure 5.14 depicts the composition of a behavior specification. Each behavior specification has the following five basic properties. First, it has a name (`behaviorName`) that is unique in relation to the other behavior specifications from the same skill. Second, it has the name of the preferred executor (`executorName`) that will create and hold the behavior. Third, it has the name of the task (`taskName`) under which the behavior will execute. Fourth, it has the behavior class (`behaviorClass`), which defines the reaction method for the specified behavior. Fifth, it has an integer (`invokePeriod`) that represents the desired number of milliseconds between consecutive reaction method invocations.

Along with the five basic properties, each behavior specification has two parallel arrays that correspond to the reaction method parameters. The first array is an array of port names, where each reaction method parameter maps to a specific port in its parent skill. Each reaction method parameter must have an interface type that is compatible with the type defined in its corresponding port specification. The second array is an array of method call information objects. It defines the information that is passed with method calls through write interfaces and is processed by compatible filter methods. Array entries should be left `null` for parameters that that do not take write interfaces. Each behavior specification, in

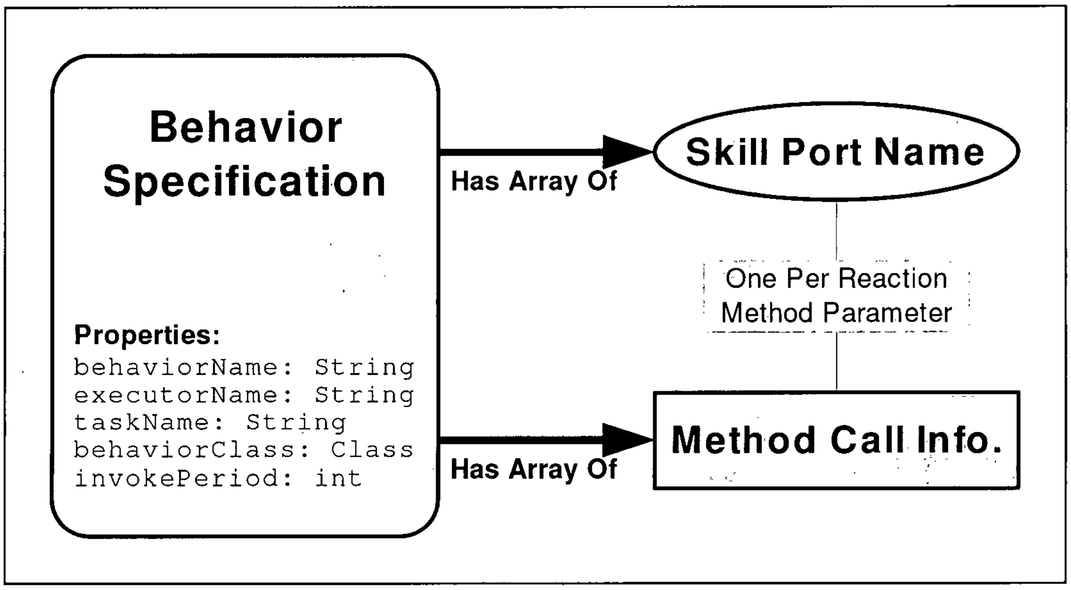

Figure 5.14: The Behavior Specification

Java source code, is an instance of class `C_BehaviorSpecification`, and this class defines methods for setting the forementioned properties.

5.4.3 Connection Specification

Each connection specification specifies a connection between a skill (the source) and either a state object or another skill (the target). In these connections, the source skill is the active component, and its behaviors call methods on the target component (which is passive) through a specific port. The connection is defined by the name of the source component, the name of the target component, and the name of the port. For each connection specification, the specified port must be defined for the source skill, and it must have an interface for communicating with the target component. Each connection specification, in Java source code, is an instance of class `C_SkillConnectSpec`. This class takes property values through its constructor, and it provides methods for retrieving these values.

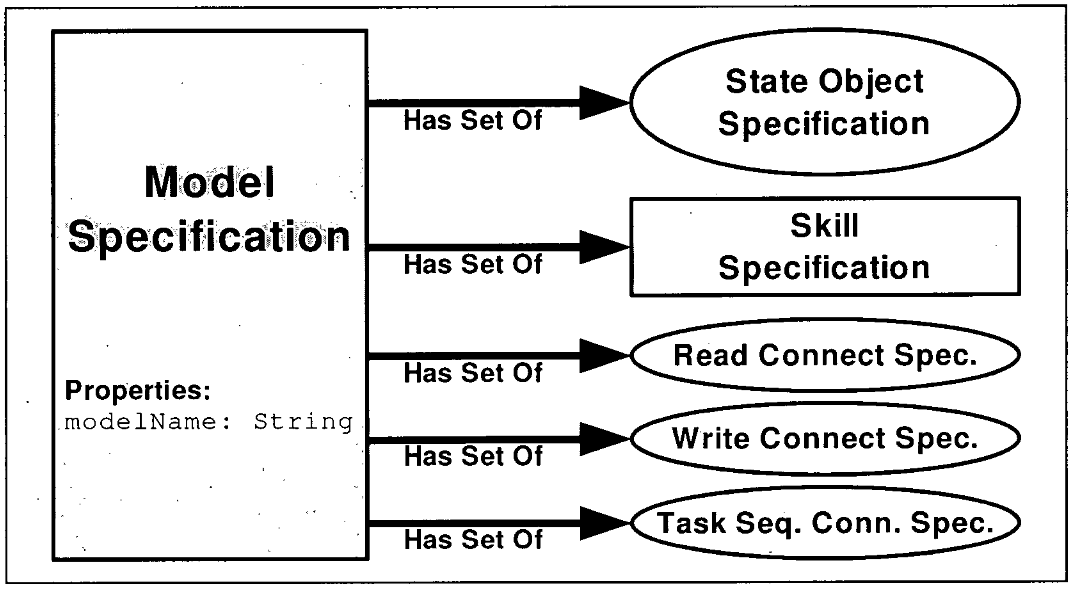

Figure 5.15: The Model Specification

5.4.4 Model Specification

Figure 5.15 depicts the composition of a model specification. Each model specification has a name, which allows the execution manager, and system users, to identify it. As stated previously, it has collections of state object specifications and skill specifications. These collections are sets, and each set must not have more than one state object specification, or skill specification, with the same name.

Each model specification has three sets of connection specifications, which link skills to state objects and other skills. The first set is for read connections between skills and state objects, where the behaviors in each skill communicate with the connected state object using its read interface. The second set is for write connections between skills and state objects, where the behaviors in each skill communicate with the connected state object using its write interface. The third set is for task sequencing connections, where behaviors in the source skill, through a skill interface, set tasks for the target skill.

Each model specification, in Java source code, is an instance of class `C_Robot-ControlModel`. This class provides the methods for configuring, and inspecting, the different parts of a model specification. For organization purposes, the constructor provides the only way to set the model specification name.

5.5 Low-level Details

Recall that a major goal in developing my software framework and execution system is ensuring that the coder is shielded from network communication details. For example, behaviors should be able to call state object methods remotely without any network communication details in their coder-supplied implementations. Section 5.3 showed how coders write component implementations that do not have such details. The execution system must provide these details, and my execution system implementation is successful at doing that. This section first describes how my execution system handles interactions with state objects, followed by a description of how filters fit in. After that, it describes how my execution system handles calling reaction methods for behaviors, considering that each reaction method declares different parameters.

5.5.1 Interactions with State Objects

As stated in Section 5.2, Java provides RMI and Reflection, which allow my software framework and execution system to hide networking details from coders. RMI, on its own, does some of the work by creating socket connections and sending data over them. However, RMI requires the remote object implementation to include special code for extending classes and throwing exceptions. Likewise, it requires the method caller to explicitly catch the thrown exceptions. All this extra code is RMI-specific and hinders the flexibility of developed systems. Reflection allows my execution system to regain that flexibility.

Figure 5.16 depicts a behavior calling a method on a state object. It suggests how the RMI-specific code can be hidden from coder-supplied code through the application of a new proxy layer on top of the RMI proxy. This proxy layer defines two components: the *proxy* and the *wrapper*. The *proxy* is an ordinary proxy object that mirrors the target state

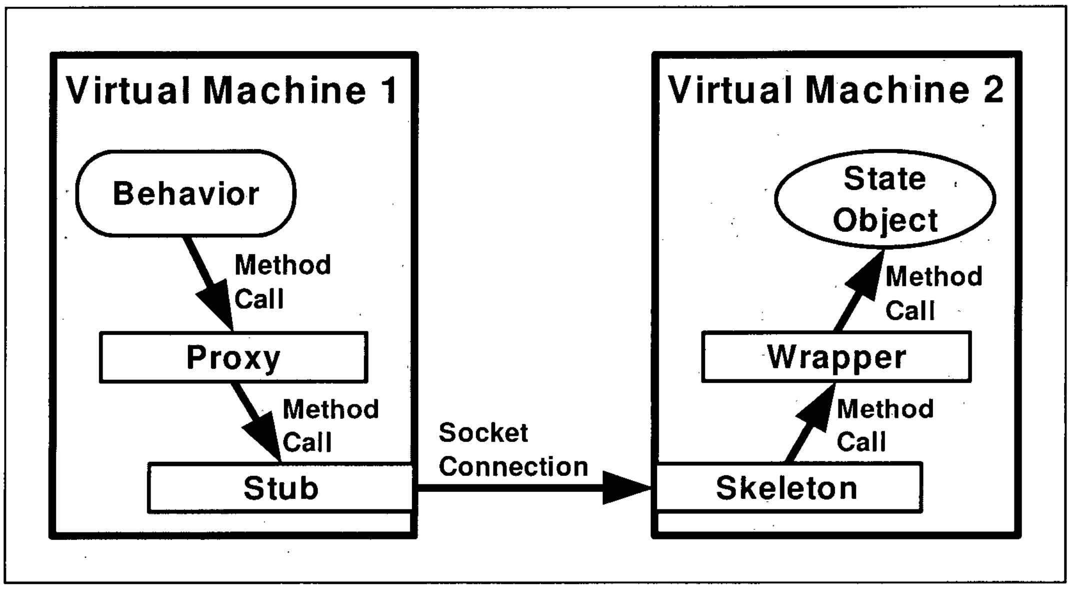

Figure 5.16: State Object Method Invocation

object. The *wrapper* is an RMI remote object that encloses, or “wraps”, the state object. It is these two objects that take care of the RMI details. In the depicted scenario, the behavior first calls the proxy method, which matches, in declaration, the target state object method. The proxy method, using RMI, calls the corresponding wrapper method, which directly calls the target method.

For the new proxy layer to work, my execution system must have the ability to generate classes for the proxy objects and wrapper objects. Because behaviors communicate with state objects through their interfaces, each interface requires a proxy object that implements it. These proxy objects would, then, be passed into behavior reaction methods where they require state object interfaces. My execution system can generate the required classes using Reflection, where each state object has a different proxy-wrapper pair for each of its interfaces. For proxy classes, it uses Reflection to analyze each state object interface and then uses that information to generate the source code for the proxy class. The same proxy class implementation can be shared by all state objects that require it, since the proxy object is not directly coupled with concrete state object implementations.

Generating classes for wrapper objects requires more work. First, because the wrapper objects are RMI remote objects, the execution system must generate an appropriate remote interface, as required by RMI. The wrapper class must implement this interface, which is used by the corresponding proxy object to call wrapper methods. Second, the execution system must generate the wrapper class. As with the proxy classes, the execution system generates the wrapper classes, and interfaces, using the state object interfaces. Proper wrapper class generation requires the name of the target state object so that the wrapper object can call its methods. Thus, the same wrapper interface can be shared by all state objects that need it, but a different wrapper class must be generated for each state object class.

The preceding paragraphs described how my execution system can generate source code for the required classes. It writes this source code to files that must be compiled and loaded. The execution system compiles the source code by issuing operating system commands. Specifically, it invokes the `javac` compiler for all source code, and it invokes the `rmic` compiler for RMI classes. Both compilers must be accessible from the command line, like through the system path or a symbolic link. Reflection allows the execution system to load the generated classes and create instances of them.

5.5.2 Filter Method Invocation

Recall that each state object can have a filter, which intercepts method calls through write interfaces. To facilitate filters, the wrapper object for each write interface must have the ability to call the correct filter method. Recall that my software framework does not require filters to implement *all* the methods provided by their corresponding state objects. Because wrapper implementations cannot anticipate, at compile time, the concrete filter class, the wrapper objects cannot call filter methods directly. While the wrappers can make method calls through Reflection, that would be several times slower than a direct method call.

My execution system overcomes the problem of calling filter methods by generating *filter wrappers*, which are customized for each triplet of write interface, filter class, and state object class. Each filter wrapper, generated using Reflection, provides all the methods

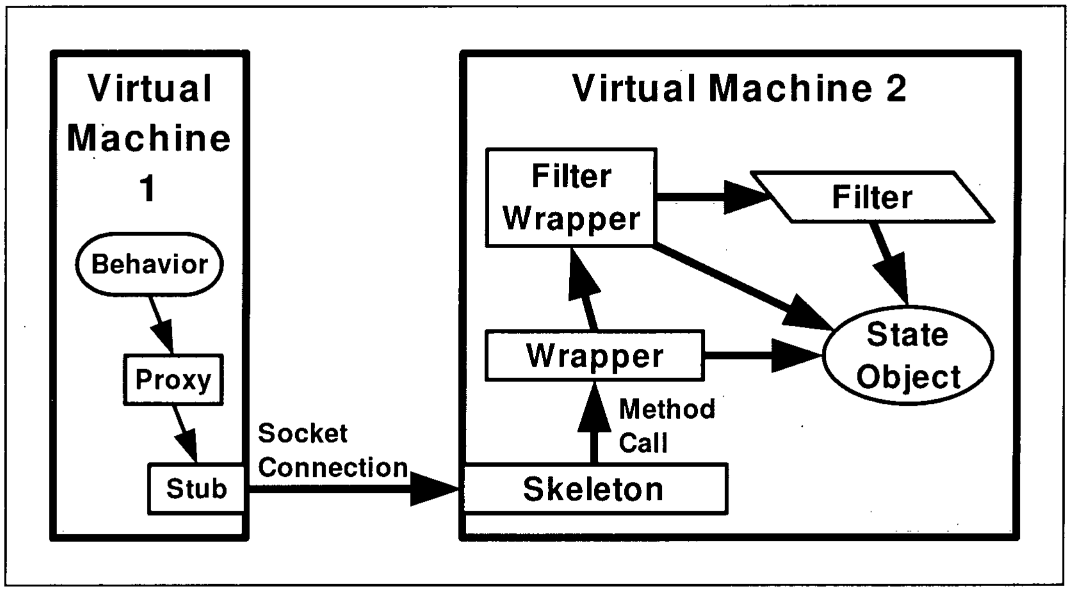

Figure 5.17: Filter Method Invocation

declared by the write interface. For each filter wrapper method, if the filter provides a compatible method, the filter wrapper calls that filter method. Otherwise, the filter wrapper bypasses the filter and calls the corresponding state object method directly. If no filter exists at all, the state object wrapper bypasses the filter wrapper and calls the state object method directly. Figure 5.17 depicts the method call situation.

Recall, from Section 5.3, that each filter method must take a method call information object. In addition, for a filter method to be called, that object must be assignable to the type required by the filter method. Because each filter method receives a different object for each calling behavior, this object should originate on the behavior side of a method call. To facilitate this, my execution system generates each wrapper method such that it takes an extra argument, which is the method call information object (of type `C_Method-CallInfo`). Each proxy method, then, passes this object when calling the corresponding wrapper method. Each behavior, then, receives its own instance of the required proxy class, and each instance contains the respective method call information object.

5.5.3 Reaction Method Invocation

My execution system must be able to repeatedly call the reaction method of each behavior with the proper arguments. The required number of arguments, and the type of each argument, differs with each behavior. Because the execution system cannot anticipate, at compile time, the concrete behavior class, it cannot call reaction methods directly. As with filter implementations, using Reflection to make method calls is too slow. However, using Reflection to generate custom classes results in faster reaction method calls. For each behavior class, my execution system generates a class (called the *behavior method caller*) that provides a single method. This method, which is provided by all such classes, takes an array of arguments, and it makes a direct call to the corresponding reaction method using the argument values.

5.6 High-level Details

This section describes the implementation of my execution system at a higher level than that described in Section 5.5. Specifically, it describes the individual processes that work together to execute robot control systems. It first describes the implementation of executors, followed by the implementation of the execution manager.

5.6.1 Executor Implementation

Each executor is a Java RMI remote object, and it provides methods for controlling the execution of state objects, behaviors, and skills. Its remote interface, `I_Executor`, provides four methods for each component type, along with a method for retrieving the executor name. The first method, for each component type, creates an instance of the specified component and returns a remote interface for controlling the created component. I refer to these interfaces as the *state object control*, *behavior control*, and *skill control*, respectively. Each creation method takes an object of type `C_ComponentLocator`, which is used to identify, and locate, the specified component within a model specification.

The second method, provided by the executor for each component type, returns a list of component locators that correspond to created components of the respective type. The third method returns the appropriate control interface, given the component locator. The fourth method destroys a component instance, given its component locator. The following paragraphs give more details about component creation and control, starting with state objects, and continuing with behaviors and skills.

State Object Creation and Control

State object creation requires the specification of a state object class and a filter class. The state object control interface, which has type `I_ObjectControl`, provides methods related to wrapper objects. Most importantly, for each interface type (read, write, and user), it provides methods for creating a wrapper object, given the wrapper class. Write wrapper creation, along with the wrapper class, also requires the filter wrapper class, which may be `null`. The state object control interface also provides methods for retrieving a list of wrapper names, retrieving a specific wrapper interface, and destroying a specific wrapper.

Behavior Creation and Control

Behavior creation requires the specification of a behavior class and a class, as described in Section 5.5, that is customized to call the corresponding reaction method. The behavior control interface, which has type `I_BehaviorControl`, has two superinterfaces. One (`I_BehControl_ExecMan`) provides methods useful to the execution manager, and the other (`I_BehControl_Skill`) provides methods useful to the parent skill. The following paragraphs describe the methods from both interfaces.

Methods for Execution Manager:

```
int numRequiredArguments()
```

Returns the number of arguments required by the reaction method.

`setBehaviorArgument( int index, I_BehaviorArgument arg )`

Sets the argument value that will be passed into the reaction method at the given parameter index. The argument parameter, essentially, accepts a proxy to a state object or a skill.

`I_BehaviorArgument theBehaviorArgument( int index )`

Returns the argument value, as set by the previous method.

`void setInvokeTimePeriod( int millis )`

Sets the time period (in milliseconds) between reaction method invocations.

`int theInvokeTimePeriod()`

Returns the value as set in the previous method.

`void allowExecution()`

Allows behaviors to execute, such that their reaction methods will be called when they are enabled.

`void blockExecution()`

Blocks behavior execution, overriding the enabled status set by skills. This method complements the previous method.

`boolean isExecutionBlocked()`

Returns `true` if and only if reaction method execution is blocked.

`boolean isReactionExecuting()`

Returns `true` if and only if the reaction method is executing. This allows the execution system to wait for the reaction method to stop executing.

Methods for Skills:

`void enableBehavior()`

Enables a behavior such that, if execution is not blocked, the execution system will call its reaction method. This sets the behavior completion status to *incomplete.*

`void disableBehavior()`

Disables a behavior such that the execution system will stop calling its reaction method until it is re-enabled. This sets the behavior completion status to *complete.*

`boolean isBehaviorEnabled()`

Returns `true` if and only if the behavior is enabled.

`boolean isExecutionComplete()`

Returns `true` if and only if behavior execution has completed. Completion is signaled by the reaction method returning `false` since the behavior was last enabled.

Skill Creation and Control

Skill creation does not require the specification of any implementation classes, since the execution system provides its own skill implementation. Thus, the skill creation method takes only a skill locator. The skill control interface, which has type `I_SkillControl`, provides two methods, where each method returns a different interface for interacting with the skill. The first method returns an interface that is used by a skill proxy to communicate with the skill. That interface provides methods that correspond to the task sequencing methods described in Section 5.3. Figure 5.18 clarifies the skill communication situation.

The second method provided by the skill control interface returns an interface that is used by the execution manager to configure a skill. That returned interface provides the following methods.

`String[] theTaskNames()`

Returns an array containing the names of all tasks defined for the skill, where each task has at least one behavior registered with it.

```
void addBehaviorToTask( String taskName,
                        C_ComponentLocator locator,
                        I_BehControl_Skill behCont )
```

Registers the given behavior with the specified task such that it is enabled whenever

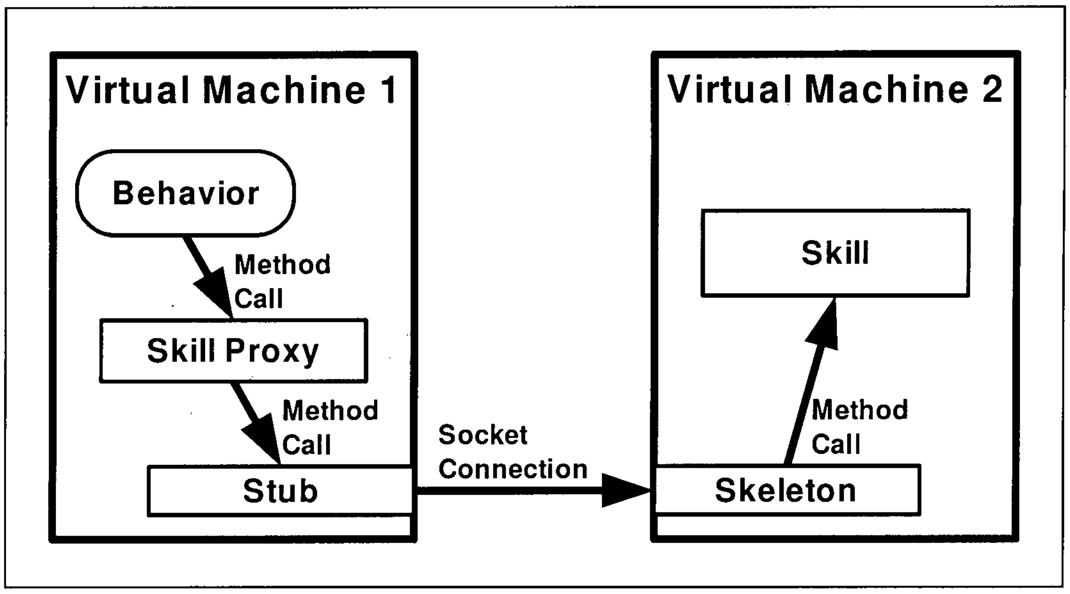

Figure 5.18: Skill Method Invocation

the current task is set to be that specified task. The locator is required for identifying the behavior, and the behavior is given by its control interface. Adding a behavior to a non-existant task results in the creation of a new task.

`C_ComponentLocator[] theBehaviorLocators( String taskName )`

Returns an array containing the locators for the behaviors that have been registered with the specified task.

```
I_BehControl_Skill theBehaviorControl(
           String taskName, C_ComponentLocator locator )
```

Returns the behavior control interface for the specified behavior, registered with the specified task. It returns `null` if no behavior with the given locator has been registered with the task.

```
void removeBehaviorFromTask(
           String taskName, C_ComponentLocator locator )
```

Removes the specified behavior from the specified task. It removes the task if that task would have no remaining behaviors.

5.6.2 Execution Manager Implementation

The execution manager is an RMI remote object that is registered with a naming service such that executors and client applications can access its functionality. For executors, it has an interface that provides a single method. This method allows an executor process to register its executor stub with the execution manager so that the execution manager can communicate with the executor. If the new executor has the same name as a previous executor, it will replace that previous executor. For users (through client applications), the execution manager provides several methods, which are described in the following paragraphs. These method descriptions are followed by a description of the steps required to load a model specification.

Execution Manager Methods

`void loadNewModel( I_RobotControlModel_Read model )`

Loads the given model specification into the execution system. This stops execution of any existing model and unloads it.

`boolean isModelLoaded()`

Returns `true` if and only if a model is currently loaded into the execution system.

`I_RobotControlModel_Read theLoadedModel()`

Returns a copy of the model that is currently loaded. It returns `null` if no model is loaded.

`void unloadCurrentModel()`

Unloads any model that is currently loaded.

`void startModelExecution()`

Starts the execution of the loaded model by allowing all of its behaviors to execute (through `allowExecution`). This method does nothing if no model is loaded.

```
boolean isModelExecuting()
```

Returns `true` if the loaded model is executing and `false` if no model is loaded.

```
void stopModelExecution()
```

Stops the execution of the loaded model by blocking the execution of all its behaviors (through `blockExecution`). This method does nothing if no model is loaded.

```
I_Object_User theUserInterface( C_ComponentLocator locator,
                                Class interfClass )
```

Returns the user interface for the state object identified by the given locator. The specific interface is identified by the given `Class` object.

```
I_Skill_TaskSeq theSkillInterface(
                      C_ComponentLocator locator )
```

Returns the skill (task sequencing) interface for the specified skill, if it is part of the loaded model. It returns `null` if the skill does not exist. This method allows users (through client applications) to control the execution of a skill. For any behaviors to execute, client applications must control at least one skill in a given robot control system. These skills can, then, have behaviors that set tasks for other skills.

Model Specification Loading

The loading of a model specification involves four main steps. First, the execution manager must generate all the classes required to fill the communication gaps, as described in Section 5.5. Second, it must create all state objects, and through their control interfaces, it must create the required wrapper objects. Third, it must create all skills, and for each skill, it must create the required behaviors and register them with the specified tasks. Finally, it must configure the reaction method parameters for each behavior by creating the required proxy objects (for state objects and skills) and passing them into the behavior control interface.

When assigning components to executors, the execution manager uses the executor designated in the model specification. However, for the cases when that executor is not accessible, the execution manager provides its own executor, within the same virtual machine, for component execution. Thus, a robot control system can always execute despite missing executors. The default executor is useful during system development, when the user does not have access to all machines (like the physical robot) or does not want to start up all required executors.

5.7 Summary

This chapter described how developers create robot control systems using my software framework, how they execute these systems with my execution system, and how that execution system works. In particular, it described how coders write Java classes for state objects, filters, and behaviors, and it described how designers compose them into model specification objects. It also described how my execution system consists of a central execution manager, which controls the execution of components that reside within executor processes. Communication among components is provided by the execution system and it is facilitated by classes generated at runtime. These generated classes handle network communication details to promote system flexibility and to allow coder-supplied implementations to focus on core functionality.

Chapter 6

Example System

Chapter 5 demonstrated that my software framework and execution system can actually be implemented in software. However, that chapter did not demonstrate how well my execution system works, if it works at all. In addition, it did not demonstrate how well robot control systems work when developed using my software framework. Thus, I performed tests to prove my claims that my software framework and execution system facilitate the execution of robot control systems. As stated in the introduction, I developed a robot control system that performs navigation tasks, where the robot must move to a given goal location.

In this chapter, Section 6.1 gives an overview of the navigation system I developed to test my software framework and execution system. Section 6.2 describes this navigation system at a greater level of detail, focussing on its state objects and skills. Section 6.3 discusses the results I obtained from testing my navigation system. Section 6.4 summarizes this chapter.

6.1 System Overview

A robot control system that demonstrates the usefulness of my software framework and execution system must exploit the various features they provide. To show reactivity, the robot control system must use state objects to provide access to sensors and actuators, and it must use behaviors to interact with those state objects. To show deliberation, the system must use state objects to store environment representations, and it must use behaviors for planning and task sequencing. To show task sequencing with skills, the system must have behaviors that control skills, and different skills must contain different combinations of behaviors and tasks. To show coordination, the system must use filters to manage method calls originating from different behaviors.

Recall that the robot control system that I chose for a demonstration is one that performs navigation tasks, where the robot must move to a given goal location. As in the motivational example in the thesis introduction, the robot must move quickly and smoothly while avoiding collisions with people and other obstacles. The target robots for this demonstration are José and Eric, which were described in Section 2.4. Different researchers have attempted to implement, and improve, navigation systems for these robots [ML98, Pou98, Car01]. The remainder of this section describes existing implementations for José and Eric, which have influenced my example system, followed by an overview of my own implementation.

6.1.1 Existing Navigation Systems for José and Eric

Each navigation system implementation uses the same basic architecture described in Section 2.4, and each implementation uses the same data structures in shared memory. Most notably, they have access to an occupancy map [ML98], which is a two-dimensional grid that represents the space along the horizontal plane of robot motion. In addition, they have access to a radial map, which is a robot-centered representation of the space around the robot. Both the occupancy map and radial map are constructed using data from vision sensors, and the radial map, which represents the most recent data, is used to update the occupancy map. Besides the occupancy map and radial map, each implementation has access to path plans, which are sequences of waypoints that the robot must follow to reach its goal location. Figure 6.1 depicts a path plan for a robot, where the robot must pass three waypoints on its way to the goal. A path planning module computes these path plans using data in the occupancy map.

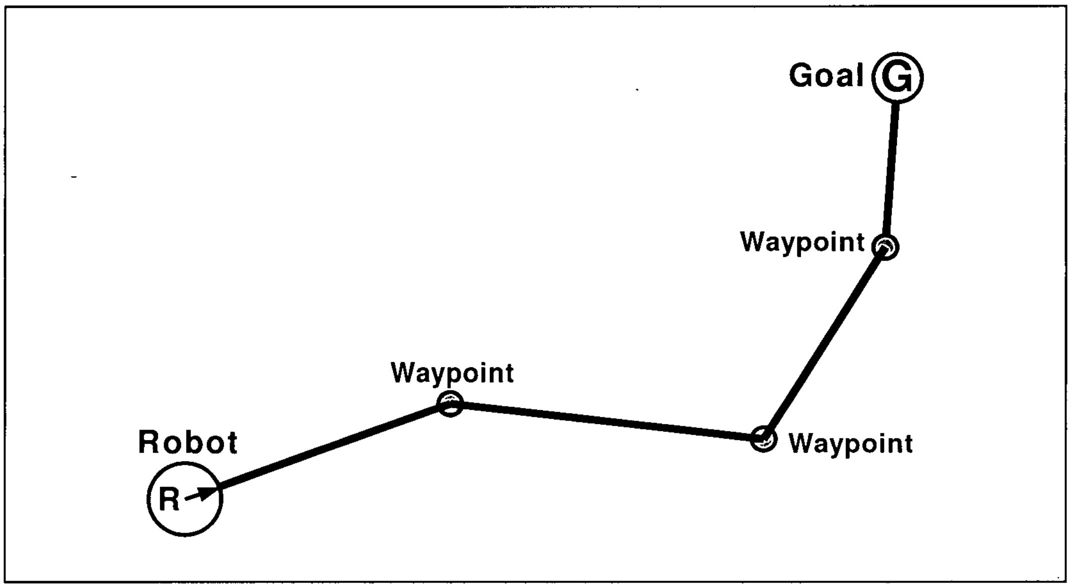

Figure 6.1: A Robot Following a Path

Each navigation system implementation makes the robot follow the sequences of waypoints computed by the path planning module. They differ in *how* they make the robot follow these paths. Murray's implementation [ML98] makes the robot follow the given path exactly, using straight line segments between waypoints, and assuming the environment is static. Poupart's implementation [Pou98], using data from sonar sensors, attempts to make the robot follow a smoother path, while being reactive to dynamic changes in its environment. Carbonetto's implementation [Car01] provides a similar smooth path follower, but it uses data from vision sensors rather than from sonar sensors.

Poupart and Carbonetto both based their implementations on the Dynamic Window technique proposed by Fox *et al.* [FBT97]. Poupart uses that technique in conjunction with the Pure Pursuit technique proposed by Coulter [Cou92]. The following paragraphs describe both of these techniques, as they relate to the two implementations.

The Dynamic Window Technique

The Dynamic Window technique [FBT97] works for robots with non-holonomic motion, such as José and Eric, that can move only in the direction that they face. These robots have a translational velocity and a rotational velocity that can be set only at fixed time rates. Thus, within each time step, the robot motion can be estimated to be a circular arc (or curvature) that is defined by the velocity pair.

The dynamic window algorithm, for each time step, finds the optimal velocity pair that allows the robot to make progress toward a goal location while avoiding obstacles. The "dynamic window", itself, refers to the fact that within each time step, the robot can safely reach velocities within a given range from its current velocities. Thus, to simplify the search space, the algorithm evaluates only velocities within the allowable range, or dynamic window.

The dynamic window algorithm evaluates a fixed number of velocity pairs within the dynamic window. The velocity pair it applies to the robot motors is the one that receives the highest score from an objective function. This objective function has the following form [FBT97], as demonstrated in Figure 6.2.

$$G(v, \omega) = \sigma(\alpha \cdot heading(v, \omega) + \beta \cdot dist(v, \omega) + \gamma \cdot velocity(v, \omega))$$

This objective function computes, and smooths, the weighted sum of three terms, where each term depends on the translational velocity (v) and the rotational velocity (ω). The first term is for keeping the robot headed toward its goal location. The function *heading*, then, returns a measure of the closeness of the robot heading with respect to the angle to the goal location. It is evaluated at the extrapolated robot heading, after the robot has moved at the evaluated velocities for a given time interval.

The second term in the objective function is for maximizing the clearance from obstacles. The function *dist*, then, returns a measure of the closeness to obstacles along the curvature defined by the given velocity pair. The third term, through function *velocity*, attempts to maximize the translational velocity so that the robot reaches its goal location as quickly as possible. To avoid near misses, function σ smooths the weighted sum of the three

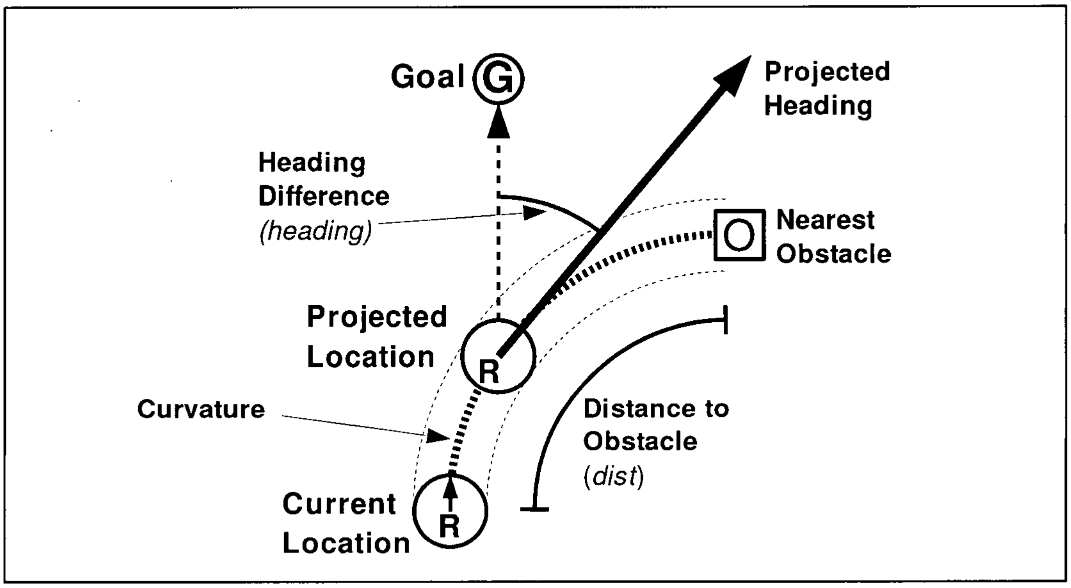

Figure 6.2: The Dynamic Window Objective Function

terms. Poupart and Carbonetto use their own variations of the objective function provided with the dynamic window algorithm.

The dynamic window algorithm is intended for a robot moving toward a single, static, goal point. On its own, it does not work well in cases when the optimal path to the goal location involves the robot turning away from that goal location. Such a situation was depicted in Figure 3.6, in the context of exception handling. Using individual waypoints as goal locations, where the robot must reach one at a time, works better but has significant drawbacks. If the navigation algorithm advances to the next waypoint only when the robot reaches the current waypoint, the robot motion will not be smooth at the waypoints. If the navigation algorithm advances to the next waypoint too early, then the robot may turn too far inside of the corner made by the straight line segments.

The Pure Pursuit Technique

The Pure Pursuit Path Tracking Algorithm [Cou92] provides a local goal location that it modifies as the robot moves. This local goal location is a point, a fixed distance ahead of

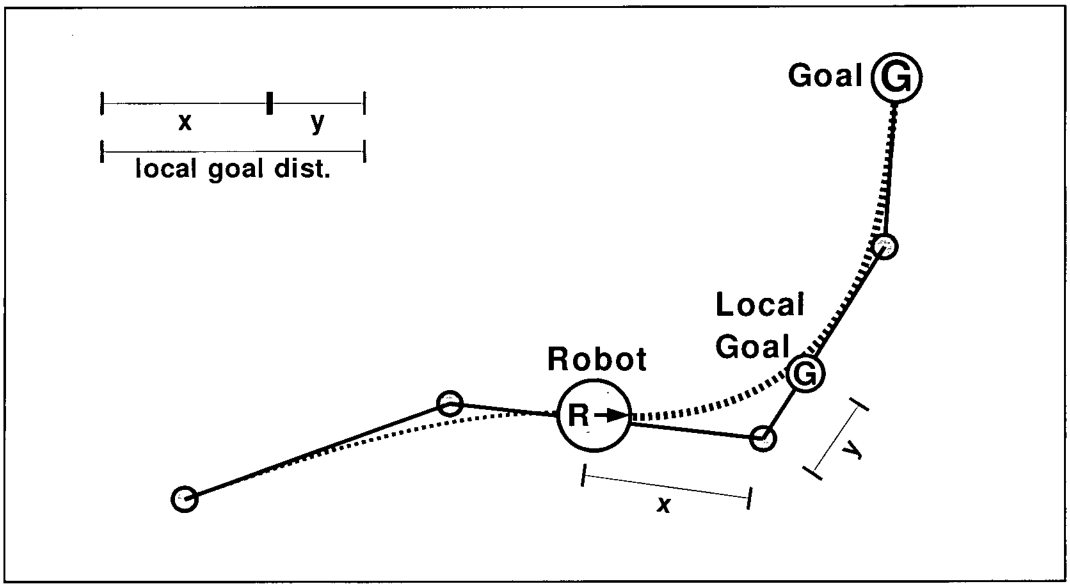

Figure 6.3: The Pure Pursuit Path Tracking Algorithm

the robot, along the path of straight line segments. Choosing a suitable distance allows the robot to pass each waypoint such that it maintains its speed, rotates smoothly, and stays close to the path of straight line segments. Poupart uses this algorithm to compute local goal locations for the dynamic window algorithm. Figure 6.3 demonstrates how the pure pursuit technique works. In this figure, the dashed line indicates the smooth path traversed by the robot, where the robot attempts to reach the constantly-updated local goal.

6.1.2 The Example Navigation System

I based my navigation system design on that provided by Poupart [Pou98], which combines the dynamic window technique [FBT97] with the pure pursuit technique [Cou92]. My design, however, incorporates vision sensors instead of sonar sensors. To demonstrate coordination with filters, my design incorporates a variant of the objective function that is provided with the dynamic window technique. In particular, my design uses a separate behavior — a "curvature evaluator" — for each term of the function. Each behavior takes input from a state object that indicates the curvatures for evaluation (the dynamic window).

This dynamic window is computed by a different behavior: the "dynamic window finder". Each curvature evaluator computes a score for each curvature in the dynamic window and writes that score to a state object that maps curvatures to scores: the "curvature score map".

The curvature score map has a filter that handles the case when different behaviors write different scores for the same curvature. Each curvature evaluator has an associated weight, and a unique identifier, that are defined within its method call information object. The filter uses this information to compute a weighted mean for each curvature, where the resulting value is sent to the underlying state object. A different behavior — the "motion updater" — finds the curvature with the highest score and uses that curvature to control the robot motors.

A task sequencing behavior — the "motion controller" — controls the execution of the forementioned behaviors, which are related to the dynamic window algorithm. Another task sequencing behavior — the "navigation controller" — controls the execution of the motion controller and some higher level behaviors. These higher level behaviors provide path planning and mapping functionality, along with the pure pursuit functionality.

6.2 System Specifics

Figures 6.4 and 6.5 depict my navigation system design. In these figures, ellipses represent state objects, parallelograms represent filters, and rectangles represent skills. Arrows indicate the components with which behaviors in each skill can interact. In particular, solid arrows indicate the flow of data between skills and state objects. Of these, arrows into a skill represent communication through a read interface, and outward arrows represent communication through a write interface. Thicker, dashed, arrows indicate that behaviors in the source skill can set tasks for the target skill.

Part 1 of my navigation system design (in Figure 6.4) contains the higher level skills, with their associated state objects. Because of time constraints, I did not implement this part of the navigation system. Part 2 of my navigation system design (in Figure 6.5) contains the skills associated with the dynamic window algorithm. I implemented most of this part, with exception to subparts that depend on Part 1. With my navigation system implementation, I developed a simple client application that allows users to control system execution and monitor the robot location, heading, and velocities. This section continues by describing the individual state objects, followed by the skills. After that, it suggests further extensions to the design.

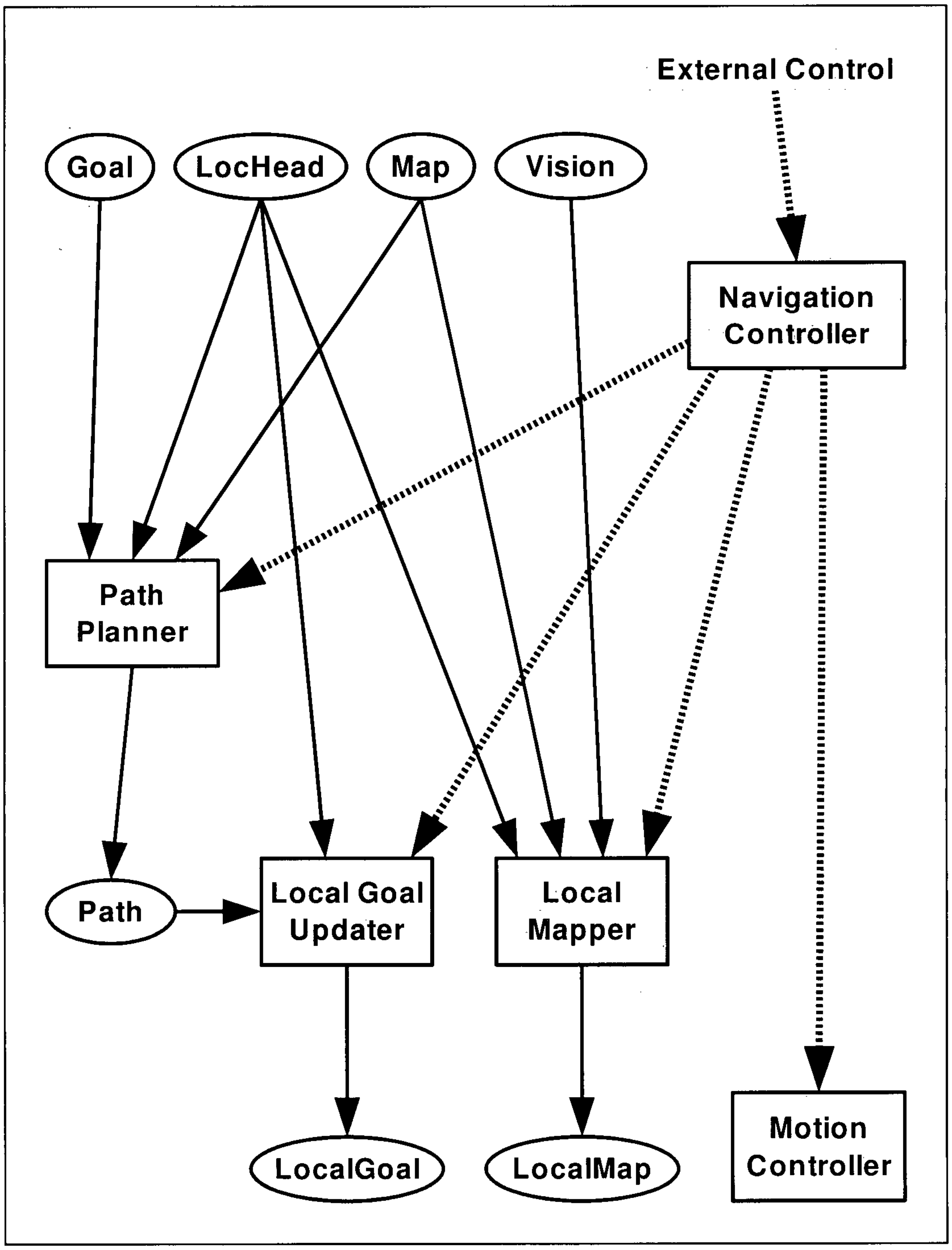

Figure 6.4: The Navigation System Model, Part 1

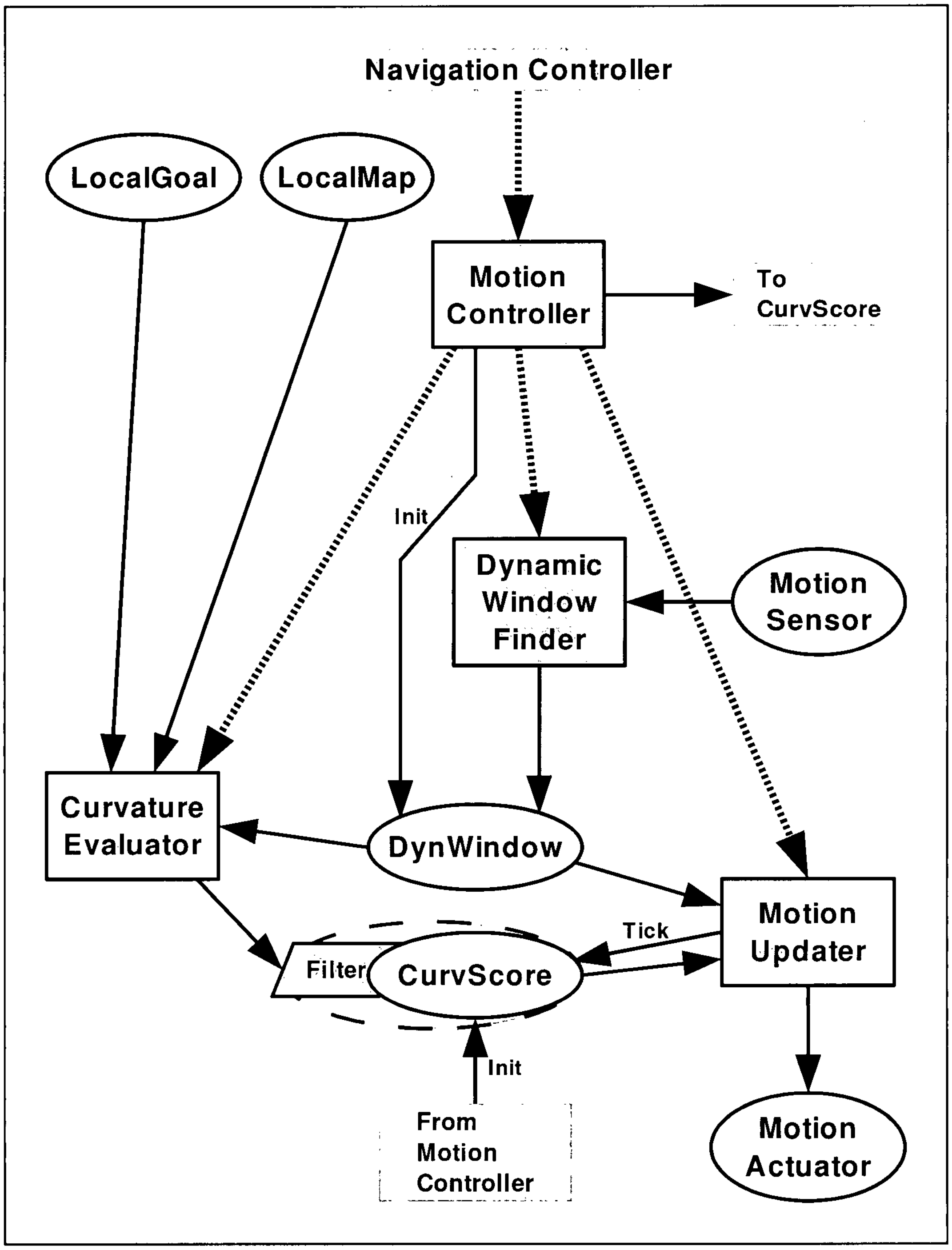

Figure 6.5: The Navigation System Model, Part 2

6.2.1 State Objects

Figures 6.4 and 6.5 depict eleven state objects in total. One additional state object, Robot-Props, was omitted for diagram simplification purposes. The following paragraphs describe these state objects in alphabetical order.

CurvScore

State object CurvScore stores the curvature score map, which provides a mapping from curvatures (velocity pairs) to their corresponding scores. Curvature scores are values between 0.0 and 1.0, where higher scores indicate better curvatures. By default, curvatures without explicit scores receive a score of 0.0. This state object, which is defined using class C_Obj_CurvatureScoreMap, provides one read interface and three write interfaces. The read interface (I_Obj_CurvScoreMap_Read) is used by the motion updater so that it can read the score associated with each curvature.

The first write interface for the curvature score map (I_Obj_CurvScoreMap_Write) is used by the curvature evaluators to set the score mappings. The second write interface (I_Obj_CurvScoreMap_Tick) is used by the motion updater to signal that it is finished with the current set of scores. This allows the curvature score map to flush, or empty, its data to prevent the motion updater from working with outdated scores. The third write interface (I_Obj_CurvScoreMap_Init) provides a method for initializing the curvature score map to an empty state. The curvature score map has a filter (C_Filt_CurvatureScoreMap), which was briefly described in Section 6.1. The method call information objects for this filter are instances of class C_MethodCallInfo_Curv, and they store the behavior identifiers and weights.

DynWindow

State object DynWindow stores the dynamic window. It provides one read interface and two write interfaces. The read interface (I_Obj_DynamicWindow_Read) provides a method that returns all the candidate curvatures within the dynamic window. One write interface (`I_Obj_DynamicWindow_Write`) allows the dynamic window finder to set the dynamic window. It provides a method that takes the translational velocity range as input and converts the range endpoints into a pair of indices into the static velocity quantization. It also provides an equivalent method for the rotational velocity. The other write interface (`I_Obj_DynamicWindow_Init`) provides methods for configuring the velocity quantization.

The static velocity quantization is necessary not only for simplifying the search space for the curvature evaluators. It also forces the curvature evaluators to work with discrete velocities so that the curvature score map does not need to provide special handling for curvatures that differ by minute fractions. Consider a robot that accelerates very slowly, where its dynamic window quantization shifts with the robot velocity. The continuous nature of this dynamic window could result in a huge growth in the number of curvatures with associated scores. The static quantization provides a manageable upper bound for the total number of curvature score mappings at any time.

Goal

State object `Goal` stores the goal for navigation, with respect to the coordinate system used by the occupancy map (`Map`). It is defined using class `C_Obj_LocationHeading`, which provides methods for storing a location and a heading. The robot must move to the location defined within this state object, and upon reaching that location, the robot must be oriented according to the defined heading. My current implementation, however, ignores the heading request. This state object has a read interface (`I_Obj_LocationHeading_-Read`) for retrieving the goal. The goal can either be set by a system user, through a given user interface (`I_Obj_LocationHeading_User`), or by any robot control system that encloses this navigation system.

LocalGoal

State object `LocalGoal` stores the local goal, which is the goal for the dynamic window algorithm. As with `Goal`, this state object is defined using class `C_Obj_Location-Heading`. It provides a read interface (`I_Obj_LocationHeading_Read`) for retrieving the goal and a write interface (`I_Obj_LocationHeading_Write`) for setting the goal. This local goal is defined in a coordinate system, with the same scale as that used for `Goal`, where the axes coincide with the robot.

LocalMap

State object `LocalMap` provides a representation of the space around the robot in terms of its occupancy. It is similar to the main occupancy map, except that its coordinate system coincides with the robot, it is smaller in size, and it is not represented as a grid. It is based on a fusion of current vision data (from `Vision`) with data from the long term occupancy map (`Map`). Because of time constraints, I did not implement this state object, so no further details exist.

LocHead

State object `LocHead` provides access to the current location, and heading, of the robot. For a real robot, this state object would be implemented using a class that interacts with the low level robot software, which is outside of my execution system. However, because of time constraints, I implemented only a version for a simulated robot. The implemented version uses class `C_Obj_LocationHeading` to define the state object. Nonetheless, that class can be easily replaced by a class that works with a real robot.

Map

State object `Map` provides access to the occupancy map. Recall that the occupancy map is a two-dimensional grid, where each cell indicates the likelihood of that cell being occupied by an obstacle. My navigation system design assumes that the map is generated outside of my execution system. This assumption would be true for José and Eric, which have good existing mapping systems. For these robots, this state object would — possibly through the Java Native Interface (JNI) — read map data as it is stored in shared memory. Because of time constraints, I did not implement this state object.

MotionActuator

State object `MotionActuator` allows behaviors to control the robot motors. Specifically, its write interface (`I_Obj_MotionActuator_Write`) allows behaviors to set the translational and rotational velocities for the robot. For a real robot, this state object, as with `LocHead`, would be implemented using a class that interacts with the low level robot software, which is outside of my execution system. However, because of time constraints, I implemented only a version for a simulated robot. The implemented version uses class `C_Obj_MotionActuator_Sim` to define the state object. In addition, it provides a read interface (`I_Obj_MotionActuator_Read`) that allows the robot simulator to retrieve the velocity settings.

MotionSensor

State object `MotionSensor` allows behaviors to retrieve the translational and rotational velocities for the robot. It provides this access through its read interface (`I_Obj_Motion-Sensor_Read`). As with `LocHead` and `MotionActuator`, I implemented this state object using a class (`C_Obj_MotionSensor_Sim`) that works with only a simulated robot. Using a write interface (`I_Obj_MotionSensor_Write`), the robot simulator can provide the state object with artificial velocity readings.

Path

State object `Path` stores a path from the robot location to the goal location (`Goal`). This path is a sequence of waypoints, where the robot must pass each waypoint, in order, to reach the goal. The path planner determines the exact sequence of waypoints, where the coordinates are relative to the occupancy map (Map). Because of time constraints, I did not implement this state object.

RobotProps

State object RobotProps stores information about the specific robot used. Its read interface (I_Obj_RobotProperties_Read) allows behaviors to retrieve values for these properties. The stored information includes the velocity and acceleration constraints for the robot, along with quantization parameters for the dynamic window. It also includes the time that must elapse between successive updates to the robot velocities. For José and Eric, class C_Obj_RobotProperties_B14 provides the state object implementation. Defined within this class, based on information by Poupart [Pou98], the maximum translational velocity is $0.3m/s$, and the maximum translational acceleration is $0.2m/s^2$. The maximum rotational velocity is $(\pi/6)rad/s$, the maximum rotational acceleration is $(\pi/9)rad/s$, and the time step for velocity updates is 0.1 seconds.

Vision

State object Vision allows behaviors to retrieve information based on vision sensor readings. In particular, it provides occupancy information about the space currently viewed by the vision sensors. For José and Eric, the radial map provides the required information. Because of time constraints, I did not implement this state object.

6.2.2 Skills

Figures 6.4 and 6.5 depict eight skills in total. One additional skill, Simulator, provides a simulated robot to work in place of a physical robot. The following paragraphs describe these skills in alphabetical order, where each skill defines only one task.

CurvatureEvaluator

Skill `CurvatureEvaluator` allows behaviors that act as curvature evaluators. These behaviors retrieve information from different state objects and use it to evaluate each curvature within the dynamic window. The following paragraphs describe the behaviors within my navigation system design, where each behavior has a reaction method that executes repeatedly to compute scores for all possible curvatures. Each behavior takes a write interface to a curvature score map. In addition, each behavior has an associated object of type `C_MethodCallInfo_Curv`, which is passed with method calls to the curvature score map. This object, as described with the curvature score map, stores behavior identifiers and weights. More behaviors can be added to give more complexity to the robot motion.

`AngleMinimizer` (class `C_Beh_AngleMinimizer`):

> This behavior gives high scores to velocity pairs that, within the next time step, turn the robot toward the local goal location. It is similar to the *heading* term defined for the dynamic window algorithm [FBT97]. For input, it takes the local goal, the dynamic window, and the robot properties. The angle score is computed as the difference between the robot heading and the angle to the local goal, and it is normalized against the range of all possible angle differences. It prunes (by giving a score of 0.0) any curvatures that, given the acceleration constraints, would not allow the robot to stop at the desired heading.

`DistanceMinimizer` (class `C_Beh_DistanceMinimizer`):

> This behavior gives high scores to velocity pairs that, within the next time step, move the robot closer to the local goal location. It is similar to the *velocity* term defined for the dynamic window algorithm [FBT97]. As with `AngleMinimizer`, it takes the local goal, the dynamic window, and the robot properties. The distance score is computed as the distance away from the local goal location, and it is normalized against the range of all possible distances. It prunes any curvatures that, given the acceleration constraints, would not allow the robot to stop immediately when it reaches the

goal location.

ObstacleAvoider (class C_Beh_ObstacleAvoider):

This behavior gives high scores to curvatures that allow the robot to travel for the longest time without intersecting an obstacle. It is similar to the *dist* term defined for the dynamic window algorithm [FBT97]. For input, it takes the dynamic window, the local map, and the robot properties. Because I did not implement the local map, I did not fully implement this behavior.

DynamicWindowFinder

Skill DynamicWindowFinder allows behaviors that maintain the velocity ranges for the dynamic window. To compute these ranges, the behaviors can use the current robot velocities (from the motion sensor) and the motion constraints for the the robot (from the robot properties). I implemented the following behavior for this skill.

DynWindowFinder (class C_Beh_DynWindowFinder)

This behavior computes the velocity ranges based on the current robot velocities and the motion constraints. Its reaction method executes repeatedly, computing a new velocity range during each iteration. This range is defined by the velocities the robot can safely reach within the next time step.

LocalGoalUpdater

Skill LocalGoalUpdater allows behaviors that repeatedly update the local goal for the robot. Normally, this skill would have one behavior, which uses the pure pursuit technique to choose a location that is along the path computed by the path planner. However, because I did not implement the path planner, I implemented the following behavior, which could be used in situations when a path is not available.

LocalGoalUpdater (class C_Beh_LocalGoalUpdater)

This behavior, given the global goal and the robot location and heading, outputs the equivalent goal, converted to the local coordinate system. Its reaction method executes repeatedly, computing a new local goal during each iteration.

LocalMapper

Skill `LocalMapper` allows behaviors that update the local map, where these behaviors may take input from the vision sensor and the global occupancy map. I did not implement this skill, which would have a behavior that fuses the current vision sensor data with the long-term occupancy map data.

MotionController

Skill `MotionController` allows behaviors that set tasks for the remaining skills depicted in Figure 6.5. I implemented the following behavior for this skill.

`MotionController` (class `C_Beh_MotionController`)

This behavior, when first enabled, configures the dynamic window quantization and clears the curvature score map. Its reaction method executes repeatedly, where on each invocation, it ensures that each skill is performing its only task.

MotionUpdater

Skill `MotionUpdater` allows behaviors that set the velocities for the motion actuator based on information in the curvature score map. I implemented the following behavior for this skill.

`MotionUpdater` (class `C_Beh_MotionUpdater`)

This behavior reads all the curvature scores from the curvature score map and finds the highest score. It then instructs the motion actuator to set the robot velocities according to the corresponding curvature. Its reaction method executes repeatedly, choosing a new curvature during each iteration.

NavigationController

Skill `NavigationController` allows behaviors that set tasks for the remaining skills depicted in Figure 6.4. I did not implement this skill, which would have a behavior that makes control choices on each reaction method invocation. First, whenever that behavior is enabled, it would assume that the goal is valid and that the robot should move to its location. At that time, the navigation controller should instruct the path planner to start its planning task. Once path planning is complete, the behavior should instruct the remainder of the skills to start their tasks, although these could be started while the path planner is working. When the robot reaches the goal location, the behavior can disable all the skills.

PathPlanner

Skill `PathPlanner` allows behaviors to update the path for the robot to follow, given the goal, the robot location, and the occupancy grid. I did not implement this skill, which would have a behavior that does the required path planning.

Simulator

Skill `Simulator` acts as a simulated robot by reading from the actuator state objects and writing to the sensor state objects. A simulator was necessary because I did not implement enough of my navigation system to make it work with a physical robot. I implemented the following behavior for this skill.

`Simulator` (class `C_Beh_RobotSimulator`)

This behavior simply reads the new velocity settings from the motion actuator and estimates, given the motion constraints, where the robot would move in the respective time step. It then updates the robot location and heading, along with the velocities for the motion sensor. Its reaction method executes repeatedly, updating the robot location, heading, and velocities, during each iteration.

6.2.3 Extensions

One useful extension to my navigation system design would be the addition of a purely reactive behavior to the motion updater skill. This behavior would map immediate vision sensor data to velocities for the motion actuator. It would be especially useful when the behaviors in the other skills fail to operate properly. Another useful extension would be the implementation of the vision processing and global mapping functionality using my software framework.

Recall that my navigation system design defined a single task for each skill. Thus, task switching within a skill, where one set of behaviors must be disabled and another set of behaviors must be enabled, was not an issue. With a separate example system, based on the components defined in Section 5.3, I verified that task switching does work properly when a skill has more than one task.

In a more complex navigation system, the need for task switching could arise. Consider a navigator robot that has two different modes of operation. Its first mode is the normal mode, which works as I have previously defined. Its second mode is a "danger mode", which is set whenever the robot is in a dangerous situation. For navigation, the robot is in a dangerous situation when a collision is likely to occur and when a collision has recently occurred. Thus, when the robot first encounters a dangerous situation, it enters its danger mode, where it acts more cautiously until the situation improves.

To support this danger mode, I can extend specific skills in my navigation system design such that they define a danger task. Skill `CurvatureEvaluator` can, for example, keep the same behavior implementations, but it can use behavior weights that put more emphasis on obstacle avoidance. A new path planner behavior can generate paths that move the robot out of the dangerous situation. The navigation controller, and the motion controller, should set the danger task for skills in the navigation system. Of course, the navigation controller must detect these dangerous situations so it can take appropriate action. It can detect recent collisions through bump sensors, and it can use data from vision sensors to help it detect likely collisions.

6.3 Test Results

The completed part of my navigation system allows a simulated robot to move smoothly to a given goal location, without regarding obstacles. This section reveals the results I obtained from executing this navigation system within my execution system. For my tests, I used my home computer, which has a $700MHz$ Pentium III processor and $256MB$ of RAM. This section begins with discussions on synchronization and efficiency, which both hindered the performance of my navigation system. It then gives the results from successful test runs.

6.3.1 Synchronization Issues

For simplicity, I decided to make each behavior in my navigation system have its reaction method invoked at the same fixed rate. To mimic the motion requirements of José and Eric, I set the time period between successive invocations to 0.1 seconds. However, at this rate, the navigation system seemed to yield random velocity settings for the simulated robot, and the robot never did reach the goal locations I set for it. I determined that this poor performance was partly related to the fact that each behavior executes in a separate thread, which runs asynchronously in relation to the other behaviors. Most significantly, the repeated flushing of the curvature score map, with the asynchronous curvature evaluators and motion updater, caused the motion updater to make bad choices. The paragraphs that follow will elaborate on this.

My initial navigation system design had a separate behavior responsible for initiating the flushing of the curvature score map. In that case, if the motion updater starts executing before the curvature evaluator completes its work, the motion updater may receive scores only for a subset of the curvatures. This unwanted occurrence is worsened by the fact that the curvature score map can be flushed before the motion updater can retrieve all of its scores. Such occurrences result in the motion updater choosing an undesired curvature, since the curvature score map may not have a score for the optimal curvature. During initial testing, I found that this unwanted situation occurred frequently.

My first attempt to overcome the synchronization problems was letting the curvature score map retain, separately, the score mappings that were set during the previous time step. The curvature score map, then, provides the previous score for a given curvature if a newer score does not exist. If the curvature evaluators execute at the same rate as the motion updater (or faster), then either the current mapping or the previous mapping should have a score for each curvature. This claim is not true if the curvature score map is flushed twice between motion updater executions. In addition, this may not be true if, after the curvature score map is flushed, the motion updater executes before the curvature evaluators. I found, during testing, that these unwanted situations occur with this first attempted solution.

A small improvement to the first synchronization attempt, as noted with the curvature score map description, had the motion updater initiate the flushing of the curvature score map. This attempt, while achieving slightly better results, still suffered from the same synchronization problems. Figure 6.6 gives results, edited for clarity, that I obtained from a test run. During this test run, the motion updater and one of the curvature evaluators, at the beginning of their reaction methods, printed the current time in milliseconds. The most important point to note is that the motion updater can execute multiple times between consecutive curvature evaluator executions. This means that the motion updater often gets scores of 0.0 for all curvatures within the dynamic window.

One main reason for my one-step history solution failing is related to the fact that behaviors execute as threads. In a single processing unit, only one thread may execute at a time, and threads will not always execute as scheduled. The Java virtual machine uses a round-robin scheduling algorithm that, depending on the underlying operating system, may use time slicing. Either way, the Java virtual machine cannot reliably maintain the order of behavior execution between consecutive reaction method invocations. One additional point to note from Figure 6.6 is that each behavior appears to execute at a rate that, on average, is approximately four times slower than expected. This indicates that the reaction methods have large execution times in relation to the chosen time step.

Slowing down the execution rate for each behavior improves synchronization, at a cost of real time efficiency. Figure 6.7 shows much better results for a time step of 1.0

```
ENTERED CURVATURE EVALUATOR AT [...]69719
ENTERED MOTION UPDATER AT      [...]69759
ENTERED CURVATURE EVALUATOR AT [...]69885
ENTERED MOTION UPDATER AT      [...]70321
ENTERED CURVATURE EVALUATOR AT [...]70801
ENTERED MOTION UPDATER AT      [...]71147
ENTERED MOTION UPDATER AT      [...]71318
ENTERED MOTION UPDATER AT      [...]71483
ENTERED CURVATURE EVALUATOR AT [...]71561
ENTERED CURVATURE EVALUATOR AT [...]71789
ENTERED CURVATURE EVALUATOR AT [...]72141
ENTERED CURVATURE EVALUATOR AT [...]72422
ENTERED MOTION UPDATER AT      [...]72521
ENTERED MOTION UPDATER AT      [...]72682
ENTERED MOTION UPDATER AT      [...]72861
ENTERED MOTION UPDATER AT      [...]72985
ENTERED MOTION UPDATER AT      [...]73107
ENTERED CURVATURE EVALUATOR AT [...]73277
ENTERED CURVATURE EVALUATOR AT [...]73476
ENTERED CURVATURE EVALUATOR AT [...]73653
```

Figure 6.6: Behavior Synchronization Problems

seconds. It does not show perfect synchronization, however, since the motion updater occasionally executes twice between consecutive curvature evaluator executions. Nonetheless, the one-step history provided by the curvature score map should counteract these occurrences. In addition, if these occurrences are rare, the navigation system can move the robot back on track. With a second curvature evaluator, a time step of 1.5 seconds works well. Still, when the system cannot execute at the desired speed, it should maintain a suitable ordering of behavior executions.

My navigation system can, to some extent, maintain the proper ordering of behavior executions on its own. The motion controller can, for example, start the curvature evaluator skill and repeatedly poll for its completion status. When the curvature evaluator is complete, the motion controller can then start the motion updater. While this sequencing strategy

```
ENTERED CURVATURE EVALUATOR AT [...]31382
ENTERED MOTION UPDATER AT      [...]31394
ENTERED CURVATURE EVALUATOR AT [...]32382
ENTERED MOTION UPDATER AT      [...]32393
ENTERED CURVATURE EVALUATOR AT [...]33382
ENTERED MOTION UPDATER AT      [...]33395
ENTERED CURVATURE EVALUATOR AT [...]34382
ENTERED MOTION UPDATER AT      [...]34392
ENTERED CURVATURE EVALUATOR AT [...]35426
ENTERED MOTION UPDATER AT      [...]35429
ENTERED CURVATURE EVALUATOR AT [...]36382
ENTERED MOTION UPDATER AT      [...]36394
ENTERED CURVATURE EVALUATOR AT [...]37382
ENTERED MOTION UPDATER AT      [...]37394
ENTERED CURVATURE EVALUATOR AT [...]38390
ENTERED MOTION UPDATER AT      [...]38392
ENTERED MOTION UPDATER AT      [...]39382
ENTERED CURVATURE EVALUATOR AT [...]39382
ENTERED CURVATURE EVALUATOR AT [...]40382
ENTERED MOTION UPDATER AT      [...]40393
```

Figure 6.7: Improved Behavior Synchronization

works, the act of polling can steal valuable execution time away from other behaviors. A better approach would be to improve my software framework and execution system such that the motion controller awakens upon the occurrences of specific events. For sequencing of skills, the event of interest would be the completion of reaction method execution.

6.3.2 Efficiency Issues

The investigation on synchronization issues revealed that the behavior reaction methods took excessive amounts of time to execute. Given the speed of my microprocessor (700 MHz) and the amount of computation required for each time step, each reaction method should, in theory, take well under a millisecond to execute. In reality, they often took hundreds of milliseconds to execute. My hypothesis was that the most significant contributing factor to this execution time is the time required for remote method calls to reach the destination objects.

My original navigation system implementation made approximately 180 remote method calls within each time step. Most of these method calls were related to the dynamic window and the curvature score map. My dynamic window used 49 curvatures for evaluation, which includes all combinations of 7 translational velocities with 7 rotational velocities. For each curvature, the two curvature evaluators updated the curvature score map with an individual method call. In addition, the motion updater retrieved each curvature score with an individual method call. Thus, 147 remote method calls, per time step, were related to setting and retrieving curvature score mappings.

I added new methods to my curvature score map implementation so that it would allow all the curvature score mappings to be set and retrieved with a single method call. Then, I changed my curvature evaluators and motion updater such that they took advantage of these new methods. My changes reduced the number of remote method calls, per time step, by 144 (48 curvatures by 3 behaviors). More importantly, it allowed successful execution at a time step of 0.2 seconds, which is several times faster than 1.5 seconds. However, these test results show only a dependence on the number of remote method calls. They do not indicate which parts of the remote method calls are responsible for the time differences.

Recall that when a behavior calls a state object method, the behavior really calls a proxy method that initiates a chain of method calls and network transfers. In particular, a method call must pass through a proxy object, an RMI stub object, a socket connection, an RMI skeleton object, and a wrapper object (with an optional filter). Researchers have shown that the standard RMI implementation provides an extreme bottleneck [PHN00]. This bottleneck is mainly due to the fact that RMI opens socket connections for transferring data and that it uses Reflection to determine how to serialize the method call arguments.

Tests on my home computer, alone, confirmed that the standard RMI implementation was responsible for the slowness of remote method calls within my execution system. Method calls through the forementioned communication chain, for the state object implementation given in Figure 5.3, took an average of 0.9 milliseconds. Method calls through the same chain, where the proxy method made direct calls to the wrapper method — without RMI — took an average of 0.1 microseconds. Thus, RMI increased the method call time by a factor of approximately $9,000$. This huge difference can cause my execution system to have poor real time performance when it must make a large number of remote method calls. I conclude that for optimal performance, behaviors and state objects should be designed such that remote method calls are minimized.

6.3.3 Goal Seeking Results

I ran two major tests to show that my navigation system works. The fact that the robot reaches its goal location quickly and smoothly indicates that all behaviors worked properly. My first test had the robot move toward a goal location that is 5.0 metres straight ahead. My second test had the robot move toward a goal location that is 3.0 metres ahead and 2.0 metres to its left. Figures 6.8 and 6.9 depict the results of these two tests, respectively. In these figures, the thick line indicates the path traversed by the robot, and the vertical bars, where they cross the path line, indicate the location of the robot at each second of simulated time. In both tests, the robot initially faces toward the right.

For the first test, the robot should, in theory, move forward at maximum speed, with no rotation, and stop at the goal location. With minor deviations from the straight line, the robot did exactly that. It successfully increased its translational velocity at the beginning of the trip, maintained a straight path toward the goal location, and decreased its speed to stop at the goal location. For the second test, the robot should, while moving forward, orient itself such that it travels toward the goal location. The robot was successful, as it quickly achieved the proper heading for reaching the goal location. The robot did sway slightly off course at times, but it was always able to recover its heading.

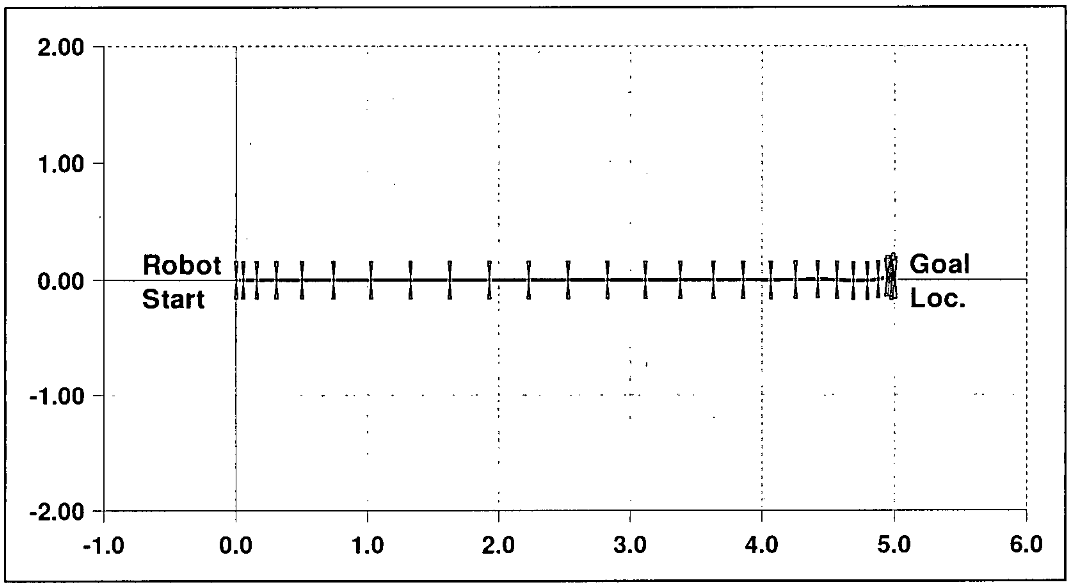

Figure 6.8: A Simulated Robot Moving Straight Ahead

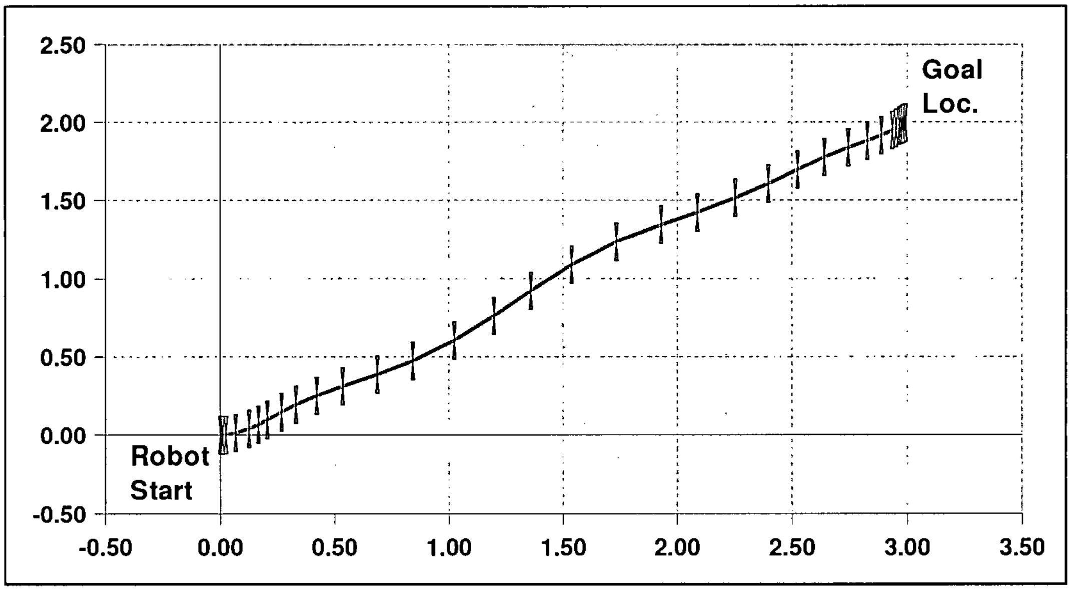

Figure 6.9: A Simulated Robot Turning to Reach a Goal Location

6.4 Summary

This chapter described a small robot control system that I developed using my software framework. This system allows a simulated robot to move quickly and smoothly to a given goal location. Although this system works only for a simulated robot, it demonstrates that my software framework and execution system facilitate the execution of robot control systems. Problems with synchronization and efficiency, however, hinder possibilities for real time execution.

Chapter 7

Evaluation

This chapter evaluates my software framework and execution system using the criteria established in Chapter 3. In particular, Section 7.1, uses the general quality criteria, from Section 3.1, to evaluate them. Section 7.2 uses the robot-specific criteria, from Section 3.2, to evaluate them. Section 7.3 summarizes this chapter.

7.1 General Quality Criteria

As stated in Section 3.1, software quality has both external measures and internal measures. This section first evaluates my software framework and execution system using the external quality measures. It then evaluates them using the internal quality measures.

7.1.1 External Quality Measures

Once again, the measures for external software quality are *correctness*, *usability*, *efficiency*, *reliability*, *integrity*, *adaptability*, *accuracy*, and *robustness*. The following paragraphs investigate these measures in the order given.

Correctness

Correctness, inherently, depends on the programmed functionality. My software framework allows for the development of an unlimited number of different robot control systems. Coders who develop systems using my framework can ensure, themselves, that their systems are correct, according to their specifications. It is impossible for my execution system to ensure that correctness. However, by taking care of networking details, my execution system removes significant sources of error from robot control systems.

Although I cannot predict the correctness of robot control systems developed using my software framework, I can, however, attempt to measure the correctness of my execution system, itself. The correctness of my execution system is crucial to the correctness of complete robot control systems that use it. It is crucial because any error in my execution system can manifest itself as an error in a complete system.

To help ensure the correctness of my execution system, I have applied unit testing. However, this testing is not thorough, as I have sacrificed the testing of some of the trickier test cases for a shorter total development time. In addition, I did not do any unit testing for some of the higher level classes. Instead, I judged the correctness of their implementations by observing the example robot control system in action.

Usability

My software framework, on its own, has good usability. By extending base classes and interfaces, and obeying simple restrictions, coders can easily write classes for behaviors, state objects, interfaces, and filters. The usability of systems developed using my framework is generally irrelevant, as all contact with developed systems goes through my execution system. Nonetheless, designers can add, and extend, user interfaces for state objects so that users of the systems can gain more external control.

My execution systems shows good usability in terms of clients interacting with the execution manager. It shows good usability because the execution manager provides a small number of methods for starting and controlling the execution of a robot control system.

However, these client applications must be appropriately designed such that users can easily interact with them. My software framework lacks usability with respect to designers, who must create model specifications. As stated in Section 5.4, designers must create these model specifications using Java source code. The process of creating model specifications, through source code, is tedious and error-prone. Thus, work must be done to improve the ease in creating model specifications.

Efficiency

The efficiency of systems developed using my software framework depends on both the coder-supplied component implementations and my execution system. With respect to the amount of memory resources used, my execution system has good efficiency. My execution system has no control over the efficiency of the coder-supplied component implementations, so those cannot affect the potential efficiency of developed systems. However, the implemented components communicate with each other through my execution system, which should provide negligible communication overhead. Section 6.3 showed that the communication overhead is not negligible. In fact, the communication overhead generally takes much more processor time than the core functionality. Thus, my execution system, as a whole, has poor efficiency with respect to processing time.

Reliability

I did not do a complete investigation into the reliability of my execution system. In my navigation system, however, my execution system showed poor reliability with respect to synchronization of behavior execution. The reliability of my execution system, alone, is more important than the reliability of robot control systems that use it. This is because my execution system cannot control the latter reliability.

Integrity

My execution system has good integrity with respect to state objects. Each state object is implemented using a Java class that defines methods for accessing its data. State objects, by not allowing direct access to their data (through member variables), protect their data from misuse. The execution manager allows only behaviors to access state objects through their read interfaces and write interfaces. In addition, the execution manager allows only client applications to access state objects through their user interfaces. This constraining of state object use to particular parts of a system helps to boost system integrity.

As with state objects, integrity is similarly good with respect to behaviors, filters, and skills. The reaction method of each behavior assumes that its arguments are interfaces for communicating with valid state objects and skills. Likewise, each filter method assumes that its arguments came from valid behaviors and valid state object wrappers. Each skill assumes that each of its behaviors are valid behaviors that have been appropriately configured.

The preceding arguments are true only under the assumptions that coder-supplied implementations have the required structure, and that the execution system operates as expected. Unfortunately, these assumptions are not generally true, and it is in these assumptions that the integrity of my execution system suffers. It is possible for applications to register different, unintended, executor implementations with an execution manager. The execution manager communicates with these implementations through the same interface implemented by the regular executors. However, these implementations may provide actual functionality that is quite different from what a valid executor provides.

Because the execution manager allows client applications to retrieve user interfaces for specified state objects, it opens up another threat to its integrity. Specifically, users may attempt to damage the data stored within state objects or cause the robot to execute harmful tasks. The execution manager should provide a way to verify, as well as possible, that its users will not attempt to harm the execution of a robot control system.

Adaptability

My execution system has good adaptability in the sense that it allows robot control systems to be changed such that they operate properly in different environments. For instance, my example system can perform navigation tasks regardless of whether it interacts with a real robot or a simulation. To adapt a navigator model such that it works with a simulation, and not a real robot, a designer must simply change the specifications for the state objects that interact with the physical sensors and actuators. The remainder of the system remains the same, including the interfaces for communicating with the state objects.

Accuracy

Accuracy does not apply to my software framework and execution system. It, instead, applies to specific implementations of robot control systems that require accurate results. For my example system, accuracy can be measured in several ways. One such way is through the closeness of the actual paths traversed by the robot to the planned paths.

Robustness

My software framework facilitates robustness in different ways. Most notably, it allows filters to prevent state object methods from receiving harmful argument values. In addition, behaviors can be coded to detect, through exception catching, broken network connections and missing state objects and skills. My execution system is robust in selected areas, such as the execution manager startup phase, where it detects, and handles, illegal user inputs.

The robustness of my execution system, because of my time constraints, leaves much room for improvement. First, any error in coder-supplied implementations, and model specifications, can cause the complete failure of the execution system. For example, a behavior that has a malformed reaction method can cause the failure of its executor, which can cause the failure of the execution manager. System boundaries must be strengthened such that they do not allow invalid data into the system. Various run time errors, such as network connection breakages, illegal memory accesses, and memory exhaustion, can also cause a similar failure to the execution system.

If a component fails to operate properly, or its data becomes corrupted, it affects every component that depends on it. For example, if a state object fails to operate properly, then all behaviors reading from that state object will perform computations with invalid data. This, in turn, negatively affects any state objects that receive any data computed by the affected behavior. If a behavior fails to operate properly, then a similar chaining of invalid state object data can occur. Section 3.1 stated that if a part of a system breaks, the system should either recover from that breakage or notify the user that a problem has occurred. Although component implementations can provide some protection, they cannot, explicitly, verify and restore broken components. My execution system should provide these capabilities, as well as the ability to restore the previous state of the data within a robot control system.

7.1.2 Internal Quality Measures

Once again, the measures for internal software quality are *maintainability*, *flexibility*, *portability*, *reusability*, *readability*, *testability*, and *understandability*. The following paragraphs investigate these measures in the order given.

Maintainability

My software framework and execution system facilitates the development of robot control systems with high maintainability. Designers can introduce new implementations of existing components by making simple changes to model specifications. In particular, a designer can change the class fields within a model specification such that they are assigned the replacement classes. The new classes must, however, be consistent with the remainder of the respective component specification. Any other changes and additions to the model specifications are similarly straightforward. Designers must, simply, update the model specification until the desired robot control system is achieved.

Flexibility

My software framework and execution system have good flexibility in the sense that coders can easily change the implementations of components without affecting the remainder of a robot control system. These changes allow a robot to perform new tasks and to operate under different configurations. The example for adaptability applies here, except that it applies to the coder and not the designer. To change a navigation system such that it works with a simulation, and not a real robot, the coder can modify the source code for the state objects that represent sensors and actuators. The coder would not need to change any other existing parts.

Portability

My software framework has good portability. This is because it can be executed on any computer system that has a supporting execution system. I claim that my current execution system can execute on any operating system that has a Java runtime environment with a version that is at least 1.4. My version requirements exist because my implementation uses code specific to Java 1.4. In addition, my claim is supported by the fact that I did not use any operating system-dependent code. I did not perform thorough tests to determine how my execution system works under different operating systems and Java runtime environments.

Reusability

My software framework has good reusability because software frameworks are, in essence, collections of reusable classes and class hierarchies that new classes can extend. The subclasses, because they use the framework, have a common structure and usage and can be processed by a supporting execution system. My framework can be used to define classes and interfaces, for behaviors, state objects, and other components, that can readily be processed by the execution system.

My model specification structure has advantages with respect to reusability. One major advantage is that the same class or interface can be used throughout the model specification. For example, any number of state objects that define the locations of different obstacles can share the same implementation class. These classes and interfaces can be used within other robot control systems. For example, different robot control systems can use the same location class.

A second advantage of my model specification structure is that, in source code, designers can use of class hierarchies to define extensions of model parts. My navigation system, described in Chapter 6, took advantage of such class hierarchies. In particular, it defined class `C_AbstractNavigatorModel`, which extended framework class `C_RobotControlModel`. The subclass specified an incomplete navigator model that consisted of the main components and connections shared by all navigator models. Class `C_NavigatorModel_Sim` extended the model defined by class `C_AbstractNavigatorModel` to yield complete model specifications for a simulated robot. A different subclass could easily define model specifications for a physical robot.

My software framework can have improved reusability with respect to modularity. With my software framework, designers may define different parts of a robot control system that have the same functionality and structure. Specifically, these different parts may have identically-defined state objects, skills, and connections. For example, if a robot system has two identical arms, its control system may use identical sets of component specifications to define the functionality for both arms. Duplication of these identical specifications is unnecessary. Thus, my software framework requires a way to define sub-networks, such that their specifications can be reused within the same robot control system and within other systems.

Readability

The readability of source code for a particular robot control system depends on the coder who wrote it. My software framework not have any significant impact on the readability of that source code, besides providing a common structure. I claim that my execution system implementation has good readability.

Testability

My software framework allows individual components to be tested outside of the execution system. For each component class, coders can apply unit testing techniques to ensure the correctness and accuracy of its methods. Users of a robot control system can test that system, as a whole, through client applications. These applications, with user interfaces for state objects, can present the users with graphical interfaces that allow them to control and monitor system execution. These applications may also write, to files, records of state object values such that users can analyze the data with other applications.

Understandability

My software framework facilitates the writing of source code that is easy to understand. When a coder writes a class definition for a state object, the general usage of that class is clear. Specifically, each instance will be remotely available for access by behaviors and client applications. In addition, it will provide methods that those behaviors and client applications can call through specific interfaces. Each state object interface clearly designates the general flow of data between a behavior and a state object.

As with state objects and their interfaces, all behaviors and filters have clear class definitions. Each behavior has a reaction method whose parameters are interfaces to state objects and skills. Each filter has methods that, essentially, are called in place of their corresponding state object methods. These filter methods normally process their given parameters and method call information structures and then call the corresponding methods on given state objects.

7.2 Robot-specific Criteria

This section evaluates my software framework and execution system using the criteria established by Ronald Arkin, Rodney Brooks, Kurt Konolige, and Reid Simmons. It investigates these criteria in that order, as given in Section 3.2.

7.2.1 Arkin's Criteria

Once again, the eight criteria given by Arkin [Ark98] are *support for parallelism*, *hardware targetability*, *niche targetability*, *support for modularity*, *robustness*, *timeliness in development*, *run time flexibility*, and *performance effectiveness*. This following paragraphs use these criteria to evaluate my software framework and execution system. They skip *robustness* because Section 7.1 already addressed that quality measure. In addition, they skip *support for modularity* because Section 7.1 addressed reusability, which covers support for modularity.

Support for Parallelism

My software framework and execution system have excellent support for parallelism. My software framework supports parallelism because it allows a robot control system to be defined as a group of components, where each component can execute independently. My execution system can start components within different executors that run in parallel on different computer systems. Each behavior, for example, can execute within a different executor.

Hardware Targetability

My software framework has good hardware targetability in the sense that it can be mapped easily onto the hardware of a real robot system. My navigation system demonstrated how state objects could act as abstractions of physical sensors and actuators. On the other hand, my software framework does not have good hardware targetability in the sense that it cannot be easily mapped onto hardware components. This difficulty in mapping results mainly from the fact that my communication protocol is based on method calls, which are difficult to implement without software. The high level of processing used by my execution system and components is too complex to be easily implemented in physical hardware. I accept this fact because, by sacrificing hardware targetability, I developed an easy to use software framework and execution system.

Niche Targetability

My software framework has good niche targetability in the sense that it provides ways to express the important relationships between a robot and its environment. Coders can express these relationships by implementing suitable state objects. My framework has moderate niche targetability with respect to the ability for robots to adapt to fit their operating environments. Section 7.1 claimed that designers can adapt robot control systems, developed using my framework, such that they can operate in new environments. However, my system does not fully allow robots to adapt themselves.

Timeliness in Development

My software framework has good support for timeliness in development. This is because it allows components to be coded with a common, reusable, structure. Overall, it satisfies the criteria for internal software quality, which lead to shorter development times. The lack of model checking and error reporting, within my execution system, does hinder development times. In addition, the current process for specifying system models — through Java source code — also hinders development times.

Run Time Flexibility

My software framework and execution system can facilitate run time flexibility, in terms of learning, through the clever use of state objects and behaviors. In particular, the robot can learn combinations of behaviors that are useful for each task by maintaining state objects that store the relevant information. Individual behaviors can, from these state objects, retrieve values that indicate whether they should perform their intended tasks. For this functionality, the control system must have behaviors that evaluate, and regulate, different combinations of behaviors in different situations.

Although the forementioned learning technique may work well, it provides unnecessary coupling with the target behaviors. In any case, my software framework does not provide any explicit support for run time evaluation. Although filters may work with useful information, such as behavior weights, my current implementation does not allow this information to be configured at run time.

Performance Effectiveness

The performance effectiveness of my software framework and execution system depends on the complete robot control system. Therefore, this quality cannot be generally measured. Nonetheless, my test results in Section 6.3 showed that my example navigation system has good performance effectiveness, despite problems with synchronization and efficiency. In particular, it made the robot move to its goal locations smoothly and efficiently.

7.2.2 Brooks' Criteria

Once again, the four criteria given by Brooks [Bro86] are *multiple goals*, *multiple sensors*, *robustness*, and *additivity*. The following paragraphs use these criteria to evaluate my software framework and execution system. They skip *robustness* because Section 7.1 already addressed that quality measure.

Multiple Goals

My software framework has excellent support for multiple goals. It allows multiple behaviors to execute concurrently, attempting to achieve their own goals. For example, one behavior may try to achieve the goal of moving a robot to a goal location, while another robot may try to achieve the goal of avoiding collisions. Some behaviors can attempt to achieve high level goals, at the symbolic level, and other behaviors can attempt to achieve lower level goals, at the actuator control level.

Multiple Sensors

My software framework has excellent support for multiple sensors. Each sensor can have a corresponding state object, through which behaviors read sensed data. For example, one state object may provide access to a vision sensor, while another state object may provide access to a sonar sensor. Different state objects may maintain representations of sensed data at different levels of abstraction, and different behaviors may use this data differently.

Additivity

My software framework has good support for additivity. Designers can add new components, such as state objects and skills, to a model specification. In addition, they can specify new executors, which can take part of the processing load. However, my software framework does not allow such additions at run time.

7.2.3 Konolige's Criteria

Once again, the three criteria given by Konolige (and Myers) [KM96] are *coordination*, *coherence*, and *communication*. The following paragraphs use these criteria to evaluate my software framework and execution system.

Coordination

Robot control systems developed using my software framework have the ability to coordinate their activity at different levels. Although the levels of control are not explicit, it is possible to group skills into layers where behaviors have the ability to set tasks for lower level skills. High-level behaviors can perform deliberative tasks, such as constructing plans for achieving goals. In my navigation system, for instance, the motion controller sequences tasks for the other skills. Low-level behaviors can perform reactive tasks, where they interact directly with state objects that communicate with sensors and actuators. In my navigation system, the behavior in the motion updater skill interacts with the motion actuator.

Coherence

Robot control systems developed using my software framework can represent their operating environments at different levels of abstraction. If my navigation system, for instance, was expanded such that it performed lower level vision processing, then it would have state objects for accessing several levels of vision-based data. The lowest level would have the camera images, and the highest level would have the occupancy maps.

Communication

My execution system can allow users to give commands to a robot through interfaces to state objects and skills. However, robots should, preferably, have the ability to autonomously determine their commands and goals. Fortunately, my software framework allows designers and coders to develop networks of state objects and skills that provide this functionality. In addition, it allows them to develop networks that provide robots with the ability to respond to humans. Behaviors and state objects are free to, through their own communication pathways, interact with existing software packages for speech recognition and synthesis.

7.2.4 Simmons' Criteria

Once again, the four criteria given by Simmons [SA98] are *task decomposition*, *task synchronization*, *execution monitoring*, and *exception handling*. The following paragraphs use these criteria to evaluate my software framework and execution system.

Task Decomposition

My software framework supports task decomposition through interactions among behaviors and skills. It allows skills to be, implicitly, arranged into hierarchies where higher level skills have behaviors that decompose tasks into sequences of subtasks for lower level skills. The semantics of the Java programming language allows behaviors to evaluate conditions, based on their reaction method arguments, and to choose the appropriate sequence of subtasks.

Task Synchronization

My software framework supports task synchronization. First, it allows the parallel execution of behaviors that, together, allow a robot to perform specific tasks. These behaviors may be part of the same skill, and they may be distributed amongst different skills. Second, my framework allows task sequencing behaviors, through the Java programming language, to impose timing constraints upon the skills they control. Specifically, the task sequencing behaviors can evaluate conditions with respect to task completion and any other information useful to task sequencing.

Execution Monitoring

My software framework, implicitly, supports execution monitoring. Behaviors, with repeated execution of their reaction methods, can evaluate conditions. Based on the results of those conditions, they can take appropriate action. For example, my navigation system may incorporate a behavior that detects when a robot has reached its goal location. When the robot does reach that location, the behavior can set a task that stops robot motion.

Exception Handling

My software framework supports exception handling in the sense that, through Java exceptions, behaviors can detect if a method call, to a state object or skill, failed. However, beyond that, it does not provide any explicit support for exception handling. Most significantly, it provides no explicit way for behaviors to signal, to their task sequencers, that an exceptional situation has occurred. Behaviors must do this implicitly through appropriately-defined state objects.

7.3 Summary

This section evaluated my software framework and execution system using the criteria established in Chapter 3. It claimed that my software framework and execution system measured high in terms of most of the general quality criteria. Most significantly, they satisfied all of the internal criteria, despite the fact that reusability can be greatly improved. With respect to the external criteria, usability, efficiency, reliability, integrity, and robustness showed need for improvement. My framework and execution system satisfied all of the robot-specific criteria, except for run time flexibility. In addition, they require more work with respect to task sequencing. With more work, my software framework and execution system can better satisfy all the criteria investigated in this chapter.

Chapter 8

Additional Notes

This chapter presents important information that was not presented in the earlier chapters of this thesis. In particular, Section 8.1 suggests specific ways that my software framework and execution system can be improved. Section 8.2 compares my framework and execution system with implementations not yet mentioned in this thesis. Section 8.3 summarizes this chapter.

8.1 Future Work

This section suggests, in three main areas, improvements for my software framework and execution system. The first area, which I believe is most important, is robustness. The second area is general software quality, excluding robustness. The third area is robot-specific and relates to the functionality requirements of robot control systems. Probably the most important work that should be done with my software framework and execution system is testing them with a physical robot that has real sensors and actuators. Such testing may reveal further strengths and weaknesses in my implementation.

8.1.1 Robustness Improvements

My execution system, with more coding effort, can attain a high level of robustness. I have identified three specific areas where robustness can be improved. First, system boundaries must be strengthened such that they do not allow improper usage. Second, my execution system must facilitate recovery from failures. Third, my execution system must allow users to restore the previous internal state of an executing robot control system. The following paragraphs discuss these three areas of improvement in that order.

Strengthening System Boundaries

System boundaries are generally parts of a system that cannot safely assume that they will be used appropriately. For example, publicly-available remote objects may be accessed by different, unknown, processes. These remote objects likely require protective functionality that ensures that they are used correctly and that their inputs are valid. This protective functionality includes argument checking and error reporting. It is important to note that only system boundaries require this protective functionality. All other system parts are inherently protected by their boundaries, and can trust that they will be used appropriately. Therefore, adding protective functionality to non-boundary system parts is generally a waste of programming effort that results in reduced system efficiency.

As stated in Section 5.1, the execution manager provides the main point of contact for users of my execution system. Thus, it is the most important system boundary. Client applications connect to the execution manager, and these applications allow users to load new models of robot control systems and control their execution. These interactions occur through method calls on the execution manager object. Since the execution manager cannot anticipate which client applications will connect to it, and what data they will send to it, it must expect the worst from these client applications.

The execution manager should expect that its client applications may call its methods inappropriately. For example, it should expect that client applications may attempt to pass it an invalid model specification. In fact, it is quite likely for client applications – especially untested ones – to give invalid information to the execution manager. This invalid information may cause the execution manager, or its executors, to perform incorrectly, or terminate abruptly, if it is not handled appropriately. Therefore, the execution manager requires a way to verify that its methods are called appropriately, with valid arguments.

For maximum protection, the execution manager should not attempt to load any model specification that is not completely valid. Section 5.4 described the valid specification format, in terms of Java class definitions. The execution manager should, instead, signal back to client applications that the model specification is invalid. In addition, the client applications should have the ability to retrieve, from the execution manager, information about why a particular model specification was invalid. This allows designers and coders to, more easily, debug their implementations. Preferably, the execution manager, upon recognizing an error in a system model, should throw an exception that contains the information required by client applications.

The interfaces used by the execution manager to interact with executors define system boundaries that may require protection. With a single execution manager, where the executor initiates the connection to that execution manager, protection for the executor is not required. It is not required because the executor is, simply, a passive component that obeys its execution manager. If the execution manager is valid, then the executor will not be harmed by improper usage. If the execution manager is not valid, then any harm to the executor is irrelevant, since no other execution managers can be affected.

If my executor implementation was extended such that it could receive requests from more than one execution manager, then its interfaces would require protective functionality. This would prevent one execution manager from destroying any work done by other execution managers. Regardless of whether the executor boundaries require protection, the execution manager must be protected from invalid executors. Most importantly, the execution manager must, somehow, verify the validity of the values returned by its executors. Currently, these values would be the component control interfaces. Without proper protection, one invalid executor could cause the entire execution system to fail.

Facilitating Recovery from Failures

When a failure occurs in part of an executing robot control system, that failure has different effects on the system, depending on its type. For example, if a state object terminates unexpectly, then all behaviors that depend on it will not be able to call its methods. In my implementation, if this situation occurs, the execution system will throw an exception that can be caught by thte calling behavior. If the behavior does not handle these exceptions, then its reaction method will exit and will be recalled as scheduled. If a behavior terminates unexpectedly, then the robot may not be able to complete its tasks.

Preferably, when a part of a behavior, state object, or skill, fails to operate properly, my execution system should be able to detect that occurrence and recreate the component. However, it does not attempt to detect component failure, and it does not attempt any component recreation. My execution system requires a way to provide this functionality in a way that does not negatively affect the executing robot control system.

When a network connection fails, between two components, these components cannot call remote methods on each other. For example, if a connection fails between a behavior and its state object, then the behavior cannot properly call methods on the state object. My execution system requires a way to recognize these network failures and handle them appropriately. Preferably, it should be able to reconnect components. One advantage of Java's RMI implementation is that it does not create hard links between stub objects and their corresponding skeleton objects. Instead, the stub object knows the host name and port number that it can use to find the skeleton object for each remote method call. Therefore, connections between components can resume normal operation, without special reconnection, when the network begins working normally.

If an executor terminates abruptly, then all components that execute within that executor are immediately destroyed. When this occurs, my execution system should do one of two things. First, it can wait for a user to restart the executor and reconnect it to the execution manager. Second, it can recreate the missing components within different executors. Currently, my execution system cannot perform any of these alternatives without reloading and restarting the entire robot control system. Preferably, any repair should take place without affecting the execution of the remainder of the robot control system.

If the execution manager terminates abruptly, the remainder of the system will continue to function normally. This is because all components execute within executors, and these components do not depend on the execution manager. Users, however, will not be able to access any functionality provided by the execution manager, such as stopping the executing robot control system. Preferably, the execution manager, when restarted, should be able to reconnect to its executors and restore its previous state. However, my execution manager does not have these capabilities.

Facilitating State Restoration

If a part of a system fails and must be restarted, my execution system should attempt to restore its previous state. It should, most importantly, recover data stored in state objects. However, other information, such as model specifications and task histories may also be useful. To facilitate this state restoration, my execution system requires a serialization mechanism that writes state object data to files so that it can be used later. Fortunately, The Java API provides classes that know how to serialize data in this manner. These classes require that state object classes implement interface `Serializable`.

One important question that arises is *when* to serialize data. It is important to note that for good maintainability, the coder-supplied state object classes should not be responsible for providing serialization functionality. I have identified two alternatives for when to initiate serialization. The first alternative is to let the state object wrappers initiate the serializaton when their methods are called. Since the read wrapper methods, generally, do not modify state objects, they would not require this functionality, and can execute more efficiently without it. The second alternative is to let the execution manager initiate serialization at timed intervals. Each state object could have its own time interval, since different state objects may change more frequently than others. Both alternatives work well, depending on how often serialization is required and whether every state change must be recorded.

A second question that arises is *where* to serialize data. Each state object file must exist on a particular computer system. Since the execution manager provides centralized control, it makes sense for state object files to be saved on the same computer system as the execution manager. These files should have names and locations that uniquely identify the state object within its model instance. In addition, the file names should have information that distinguishes different times that a state object is serialized. Preferably, each model instance should have its own hierarchy of folders that reflect the component hierarchies in the model specification.

8.1.2 Other Quality Improvements

Further improvements to execution system quality can be made in the areas of *usability*, *efficiency*, *integrity*, and *reusability*. The following paragraphs describe how.

Usability

The usability of my software framework and execution system can be greatly improved with the addition of appropriate graphical user interfaces. Most importantly, my execution system requires a graphical configuration management tool that allows designers to define model specifications. This tool would allow designers to specify components and connect them together within a network. It would also allow designers to save their networks to files, such that these files can be loaded by client applications and passed to the execution manager.

Another useful graphical tool is one that allows users to monitor the execution of *any* robot control system. This tool would allow users to browse the state objects, behaviors, and skills, that are part of an executing system. For each state object, users would be able to retrieve its properties. They would also be able to retrieve, and interact with, graphical representations of its user interfaces. Users would be able to configure the graphical tool such that it knows the graphical components that correspond to the different user interfaces. For each behavior, users would be able to retrieve its properties, including whether the behavior is currently enabled, and whether its current task is complete. For each skill, users would be able to view its tasks, identify the currently executing task, and specify new tasks for execution.

Efficiency

The most significant efficiency bottleneck in my execution system is the RMI communication between components. Thus, work must be done to find an alternate communication mechanism. Because my software framework implementation decouples coder-supplied implementations from specific communication mechanisms, it can easily use something other than RMI. My execution system should be tested with different alternatives to determine which one provides the fastest, and most reliable, communication.

One common alternative to RMI is CORBA, the Common Object Request Broker Architecture. CORBA, created by the Object Management Group (OMG), specifies an interface definition language and protocols that allow remote method invocation to take place. Unlike RMI, which is strictly Java-based, CORBA does not have any strict programming language dependence. Software components in any language can interoperate within a CORBA system if they obey the protocol specifications. While CORBA can be more efficient than RMI, it can be more complex to use. Nonetheless, Java's RMI system has support for interacting with CORBA, so CORBA can be used as the main remote method invocation mechanism with minimal changes to existing code.

Other alternatives to RMI include Java Web Services and the Remoting Framework in *.NET*. Java Web Services are geared more toward businesses sharing data, and provide excessive functionality beyond simple remote method calls. The Remoting Framework in *.NET*, is Microsoft's attempt to support remote method invocation in a style similar to RMI. While these alternatives look promising, I would prefer to incorporate a mechanism that is as easy to use as RMI. In response to the slowness of RMI, researchers have implemented more efficient replacements, which are identical in usage. Philippsen *et al.* [PHN00], for example, implemented KaRMI, which completes remote method calls several times faster than the existing RMI implementation.

Integrity

For improved integrity, my execution system requires the implementation of a protocol to prevent impostors from accessing a robot control system. This would provide the execution manager with the ability to recognize valid executors and client applications. A password system can help the execution manager with this. A sample scenario is one where a client application passes a new model specification to the execution manager. However, with that model specification, the client application can specify a password. Any application that wishes to control an instance of the given model specification must know that specified password. Another useful improvement, for integrity, would be the addition of a mechanism for verifying the integrity of existing data within a robot control system.

Reusability

For improved reusability, my software framework should allow designers to package parts of a model specification into modules. In fact, it should allow designers to define hierarchies of modules, where modules can contain other modules. Each module should represent parts of a system as a single unit that can be integrated within an existing model specification. In other words, modules should be structural abstractions of system parts, and they should not provide any functionality not representable by existing component types. For this to be true, a module-less system must allow the formation of distinct module boundaries.

Module boundaries can be formed by breaking connections between components and replacing them such that those broken connections go through a module port. Thus, each component outside a given module connects to components within the module through one of these ports. Once the ports are defined, modules can be clearly defined as single units, where external components connect to the modules through the defined ports. The exact internal composition of each module is irrelevant to the components outside of the module. Similarly, the exact external components that are connected to the module are irrelevant

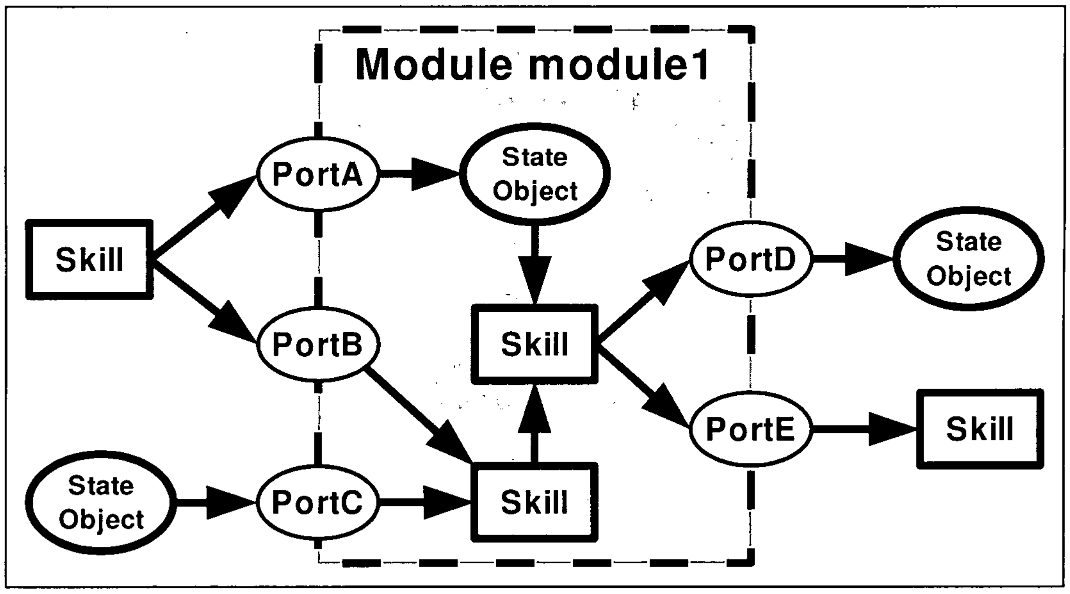

Figure 8.1: A Module with Different Port Connections

to the module. All that is important is that one component provides an interface that is specified by the port and the other component communicates with the former component using that interface. More briefly, the interface types must match.

The previous paragraph established that connections must be broken to define module boundaries. Any type of broken connection is possible, as depicted in Figure 8.1. First, an external behavior can access a state object inside the module (as in `PortA`). Second, an external behavior can set tasks for a skill inside the module (as in `PortB`). Third, a behavior inside the module can read input from an external state object (as in `PortC`). Fourth, a behavior inside the module can modify an external state object (as in `PortD`). Finally, a behavior inside the module can set tasks for an external skill (as in `PortE`). Each port may have any number of source components, which call methods, but it must have exactly one target component.

My model specifications should allow designers to define modules as separate entities that are not part of any specific model specification. This allows designers to insert modules into existing model specifications to yield larger robot control systems. Obviously, when inserting a module, the designer must connect the module ports to the necessary state objects and skills. It is possible to think of a complete robot control system as a single module that has no ports. In that sense, a robot control system can be defined, simply, as a hierarchy of modules, where the modules contain all the system components and their interconnections.

Work should be done to determine the ease of my execution system interacting with external planners, task sequencers, and robot control systems. Perhaps, for particular functionality, external systems may work better. In some cases, using these external systems, in collaboration with my own system, may result in reduced coding effort. In any case, my software framework and execution system should, whenever possible, allow the leveraging of existing technologies.

8.1.3 Robot-specific Improvements

Further improvements to my software framework can be made in the areas of *task sequencing*, *synchronization*, and *self-adaptation*. The following paragraphs describe how.

Task Sequencing

My software framework allows multiple behaviors to set tasks for a single skill. This works well when at most one of these behaviors is enabled at any given time, since when a behavior sets a task for a skill, it is assured that no other behavior will modify that task setting. Thus, when the behavior polls the skill for its task completion status, and other information, the behavior is assured that the returned information is related to the expected task. These assurances disappear when multiple behaviors concurrently control the same skill.

Perhaps these task coordination problems can be reduced if a skill can perform multiple tasks simultaneously. One possibility is allowing each skill to maintain a collection of tasks, where the skill enables the union of all its set tasks. With each task, the skill can store a value that identifies the behavior that set the task. That way, each task sequencing behavior has its own view of its target skill.

Another possiblity for task coordination is allowing each skill to maintain a single task, where each behavior can query whether the current task was set by that specific behavior. A coordination mechanism would help in this case, such as one that assigns priorities to task sequencing behaviors. Allowing each behavior to be registered with more than one task can smoothen task switching, as not all behaviors would need to stop at once. A related problem that must be handled is stopping an executing task when its task sequencer becomes disabled.

As described in Section 7.2, my software framework requires greater support for exception handling. If a behavior sets a task for a skill, and the skill cannot complete the task, then the skill should be able to signal the task sequencing behavior. This requires that the skill has the ability to identify which behaviors set its tasks. It also requires that the task sequencing behaviors have special methods that are configured to handle exceptions related to the specific task. Along with exception handling, skills should provide more ways to detect task completion, beyond the completion of all behaviors. For example, if the behaviors yield the same result, but with different algorithms, the first behavior completion should be enough to signal task completion.

Synchronization

Section 6.3 noted synchronization issues within my execution system. In particular, it does not guarantee the required ordering of behavior executions when a behavior in one skill depends on the completion of behaviors in other skills. The likelihood of achieving the proper ordering decreases as the time period decreases between successive reaction method invocations. Thus, my navigation system had to use a slower time rate in order to yield satisfactory results. As a solution to the synchronization issues, Section 6.3 suggested allowing behaviors to be awakened when specific events occur.

Currently, behaviors in skills have their reaction methods invoked at, approximately, fixed time rates. My software framework could, instead, allow designers to specify events that trigger the reaction method invocations. These events would be based on the state objects and skills with which the given behavior can interact. For example, if the behavior takes a state object interface as its reaction method argument, then the designer would be able to use the modification of that state object as such an event. Likewise, if the behavior takes a skill interface, then the designer would be able to use task switches and completions as events. Instead of triggering reaction method invocations, these events could, instead, trigger the invocation of special handler methods written by the coder. The observer pattern would be useful here.

For better control over behavior synchronization, my execution system should perform its own sequencing of reaction method invocations, rather than let the Java API handle it. This would ensure that behaviors are executed in the same order each time when they have the same invocation rates. Each executor could, for example, maintain a list, ordered by time, where each entry maps a time to a specific behavior. For each entry, the executor would invoke, at the specified time, the reaction method for the specified behavior. If the reaction method for a given behavior is still executing when it is scheduled to be reinvoked, then the executor could wait for that reaction method to return. It would then increase the times in the list to maintain the order of reaction method invocations. Alternately, the executor could skip the reinvocation of a reaction method that is still executing.

Self-adaptation

Self-adaptation allows a robot control system to reconfigure itself, at run time, for different situations. My software framework can support this self-adaptation with two main improvements, First, it should allow robot control systems to learn appropriate combinations of behaviors for different tasks. This requires components that know about specific behaviors within skills and have the ability to reconfigure skills. Essentially, these components would need to configure different behavior combinations and evaluate their overall performances.

The second improvement is allowing the forementioned components to modify the method call information that is passed into filters. These components can, similarly, try different values for different behaviors and evaluate which ones yield the best overall performance for particular tasks. My navigation system, for example, could benefit from learning appropriate weightings for the behaviors that evaluate the possible curvatures.

8.2 Comparison with Existing Implementations

My work was influenced by three existing implementations not mentioned previously in this thesis. The first is Mobility (TM) [Rea99], which consists of a software framework and execution system for developing robot control systems. The second is ArchJava [ASCN03], which is an extension to the Java programming language. The third is CNJ [Son02], which provides a graphical tool and interchange format for developing robot control systems using constraint nets. This section compares my software framework and execution system with these three implementations, in the order given.

8.2.1 Mobility

Mobility [Rea99] is distributed by iRobot for use with their mobile robot systems. Similar to my software framework, the software framework provided by Mobility allows coders to define components (with the C++ language) that are executed within its execution system. The execution system in Mobility provides a graphical user interface that allows users to browse through the executing components. For each component, Mobility allows users to interact with, and monitor, that component through the graphical user interface.

Robot control systems developed using Mobility can be spread over several processing units. Components in Mobility communicate using CORBA, which provides remote method invocation. One drawback with Mobility is that coders must write CORBA-specific code to access remote objects. In my Java RMI-based implementation, users do not write special code for remote object access.

Mobility has the advantage of being widespread in the sense that researchers around the world have used it to develop control systems for autonomous robots. In addition, it is unlikely that these researchers will rewrite their entire Mobility implementations for use with my execution system. Therefore, integration of my execution system with Mobility may yield favorable results. Research should be done to investigate the ability to integrate my execution system with Mobility and other systems.

8.2.2 ArchJava

ArchJava [ASCN03] was developed by Aldrich *et al.* to add architecture specification features to the Java programming language. It allows coders to define ports, through which objects communicate with other objects. It also provides constructs that allow coders to connect specific objects together, via their ports. Unlike my software framework, ArchJava utilizes extensions to the basic Java syntax. Thus, ArchJava source code does not facilitate the advantages provided by common Java debugging tools.

ArchJava allows coders to define connections such that objects can communicate without knowing the concrete connection type. Thus, it promotes flexibility by decoupling objects from specific communication protocols and the objects they are connected to. My software framework provides similar decoupling through state object interfaces and skills. However, users of my software framework define connections through the model specification, where ArchJava allows connections to be defined directly in the main application code. In this respect, my software framework facilitates further reusability, through decoupling, but ArchJava has the advantage of providing direct support for runtime flexibility.

8.2.3 CNJ

CNJ (Constraint Nets in Java) [Son02] is a Java-based solution for specifying and executing constraint nets. Constraint nets were developed by Zhang and Mackworth [ZM93] as a constraint-based approach to robot control. A constraint net can be thought of as a directed graph, where the nodes can be either *locations* or *transductions*, and data flows in the direction of the edges. Locations are components that maintain the state of a constraint net by holding values. Transductions are components that, given values from input locations, compute new values for output locations. When compared with my software framework, locations are similar to state objects, and transductions are similar to behaviors.

CNJ has several main features that are of interest to my software framework. First, CNJ provides a visual programming environment, which allows developers to specify, and connect, locations and transductions, as well as execute constraint nets. My software framework and execution system require a similar environment, as described in Section 8.1. Second, CNJ leverages JavaBeans, which is an API for developing component-based software systems. Perhaps JavaBeans may be useful in my implementation.

The third interesting feature provided by CNJ is a portable XML-based interchange format. My software framework requires a similar interchange format for model specifications. Fourth, CNJ allows the definition of modules, which are similar to what was described in Section 8.1. Finally, to ensure proper data flow, CNJ executes transductions in an order based on topological sorting. My execution system may benefit from such an ordering.

8.3 Summary

This chapter noted work that should be done to improve my software framework and execution system. It began with robustness improvements, which require boundary strengthening and the addition of mechanisms for error recovery and serialization. It continued with improvements for usability, efficiency, integrity, and reusability, as well as improvements related to task sequencing, synchronization, and self-adaptation. After noting future work, this chapter described Mobility, ArchJava, and CNJ, which provide features similar to those provided by my software framework and execution system.

Chapter 9

Conclusion

For my thesis project, I developed a software framework and distributed execution system that allows developers to create high quality control systems for autonomous robots. Systems developed using this framework are behavior-based, but these behaviors provide support for deliberative reasoning. Specifically, systems consist of networks of *skills* and *state objects*, where skills have *behaviors*, and state objects have *interfaces* and *filters*.

Interactions among *behaviors* and *state objects* provide reactivity and deliberation. *Skills*, with task sequencing behaviors, simplify the development of deliberative systems and promote strong cohesion and loose coupling. *Filters* facilitate coordination of state object inputs and can help state object robustness. *State object interfaces* are crucial for remote method invocation, and they decouple behaviors from specific state object implementations.

My example navigation system demonstrated the usefulness of my software framework and execution system. It showed how developers can create complete, distributed, robot control systems without including any networking code. Features of the Java programming language made this possible. However, my framework and execution system do require considerable work. Most importantly, their robustness, efficiency, and reusability, can be greatly improved. In addition, they require better ways to keep behaviors synchronized with one another. Nonetheless, I believe that my software framework and execution system do show great potential.

Bibliography

[AB97] Ronald C. Arkin and Tucker R. Balch. AuRA: Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 9(2/3):175–188, April 1997.

[Act99] ActivMedia Robotics. *Saphira Manual, Version 6.2*, August 1999.

[AML87] James S. Albus, Harry G. McCain, and Ron Lumia. NASA/NBS standard reference model for telerobot control. Technical Report 1235, Robot Systems Division, National Bureau of Standards, 1987.

[Ark98] Ronald C. Arkin. *Behavior-based Robotics.* MIT Press, 1998.

[ASCN03] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In *Proceedings of the European Conference on Object-oriented Programming*, 2003.

[BD00] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems.* Prentice-Hall, 2000.

[BFG$^+$97] R. Peter Bonasso, R. James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Marc G. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 9(2/3):237–256, April 1997.

[Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[Bro89] Rodney A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 253–262, 1989.

[Bro91] Rodney A. Brooks. Intelligence without reason. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, 1991.

[Car01] Peter Carbonetto. Avoiding obstacles: Implementation of an improved smooth navigation controller. Technical Report for CPSC515 Project, Department of Computer Science, University of British Columbia, 2001.

[Cou92] R. Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical Report CMU-RI-TR-92-01, The Robotics Institute, Carnegie Mellon University, 1992.

[dWdBvdHM98] Mathijs de Weerdt, Frank de Boer, Wiebe van der Hoek, and John-Jules Meyer. Specifying uncertainty for mobile robots. Technical Report INF-SCR-98-19, Utrecht University, 1998.

[EHL$^+$02] Pantelis Elinas, Jesse Hoey, Darrell M. Lahey, Jefferson D. Montgomery, Don Murray, Stephen Se, and James J. Little. Waiting with Jose, a vision-based mobile robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2002.

[EHL03] Pantelis Elinas, Jesse Hoey, and James J. Little. HOMER: Human Oriented MEssenger Robot. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence*, 2003.

[Elf89] Alberto Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, June 1989.

[FBT97] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 4(1), 1997.

[FBT98] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. A hybrid collision avoidance method for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1238–1243, 1998.

[FBT99] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, 11:391–427, 1999.

[Fir89] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Yale University, 1989.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[KM96] Kurt G. Konolige and Karen L. Myers. The Saphira architecture for autonomous mobile robots. *AI-based Mobile Robots: Case Studies of Successful Robot Systems*, 1996.

[Kon97] Kurt G. Konolige. COLBERT: A language for reactive control in Sapphira. In *Proceedings of the German Conference on Artificial Intelligence*, pages 31–52, 1997.

[LBV02] Scott Lenser, James Bruce, and Manuela M. Veloso. A modular hierarchical behavior-based architecture. In *RoboCup 2001: The Fifth RoboCup Competitions and Conferences*, 2002.

[LRDG90] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald R. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In *Proceedings of SIGGRAPH*, pages 327–335, 1990.

[McC93] Steve McConnell. *Code Complete.* Microsoft Press, 1993.

[MJ97] Don Murray and Cullen Jennings. Stereo vision-based mapping and navigation for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1997.

[ML98] Don Murray and James J. Little. Using real-time stereo vision for mobile robot navigation. In *Workshop on Perception for Mobile Agents at CVPR'98*, 1998.

[Nil84] Nils J. Nilsson. Shakey the robot. Technical Report 323, AI Center, SRI International, 1984.

[OC03] Anders Orebäck and Henrik I. Christensen. Evaluation of architectures for mobile robotics. *Autonomous Robots*, 14(1):33–49, 2003.

[OK93] Masatoshi Okutomi and Takeo Kanade. A multiple-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(4):353–363, 1993.

[PHN00] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[Pir99] Paolo Pirjanian. Behavior coordination mechanisms — state-of-the-art. Technical Report IRIS-99-375, Institute of Robotics and Intelligent Systems, School of Engineering, University of Southern California, 1999.

[Pou98] Pascal Poupart. A smooth navigation controller. Technical Report for CPSC515 Project, Department of Computer Science, University of British Columbia, 1998.

[Rea99] Real World Interface. *Mobility 1.1 Robot Integration Software User's Guide*, 1999.

[Ros95] Julio K. Rosenblatt. DAMN: A Distributed Architecture for Mobile Navigation. In Henry H. Hexmoor and David Kortenkamp, editors, *Proceedings of the AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, 1995.

[Ros00] Julio K. Rosenblatt. Maximizing expected utility for optimal action selection under uncertainty. *Autonomous Robots*, 9(1):17–25, 2000.

[SA98] Reid G. Simmons and David Apfelbaum. A task description language for robot control. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 1998.

[SGH$^+$97] Reid G. Simmons, Richard Goodwin, Karen Zita Haigh, Sven Koenig, and Joseph O'Sullivan. A layered architecture for office delivery robots. In *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 245–252, 1997.

[Sim94] Reid G. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, 1994.

[Sim96] Reid G. Simmons. The curvature-velocity method for local obstacle avoidance. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3375–3382, 1996.

[SLL02] Stephen Se, David G. Lowe, and James J. Little. Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks. *International Journal of Robotics Research*, 21(8):735–758, 2002.

[SM94] Michael K. Sahota and Alan K. Mackworth. Can situated robots play soccer? In *Proceedings of Canadian AI-94*, 1994.

[Son02] Fenguang Song. CNJ: A visual programming environment for constraint nets. Master's thesis, University of British Columbia, October 2002.

[Thr02] Sebastian Thrun. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann, 2002.

[Tso97] John K. Tsotsos. Intelligent control for perceptually attentive agents: The S* proposal. *Robotics and Autonomous Systems*, 20:5–21, 1997.

[Wel99] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[ZM93] Ying Zhang and Alan K. Mackworth. Constraint programming in constraint nets. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *First Workshop on Principles and Practice of Constraint Programming (PPCP'93)*, Providence, RI, 1993.