A DESIGN METHODOLOGY FOR THE CONFIGURATION OF BEHAVIOR-BASED MOBILE ROBOTS

A Thesis
Presented to
The Academic Faculty

By

Douglas Christopher MacKenzie

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in Computer Science

Georgia Institute of Technology

Tech Report #GIT-CS-97/01

for Karen, my friend, my confidant, my wife.

Acknowledgements

The author would like to thank his advisor, Dr. Ronald C. Arkin, for directly supporting this research and for support encompassing the entire period of his enrollment. Special thanks are given to the other committee members: Dr. Christopher G. Atkeson, Dr. Richard J. Leblanc, Dr. Wayne J. Book, and Dr. John T. Stasko for the helpful advice given during the progression of this research and for the time spent reading this document. The author would also like to recognize the Computer Integrated Manufacturing Systems program and the Material Handling Research Center who provided support during his time at Georgia Tech. Special thanks goes to Darrin Bentivegna for conducting the usability experiments and Erica Sadun for helping finalize the experiments. The friendship and camaraderie of Mark Gordon, Tucker Balch, Khaled Ali, Darrin Bentivegna, and Dr. Jonathan Cameron made this period of time one of the author's most enjoyable. And finally, the proof-reading and numerous suggestions supplied by his wife Karen are greatly appreciated.

Contents

List of Tables

List of Figures

Summary

Behavior-based robotic systems are becoming both more prevalent and more competent. However, operators lacking programming skills are still forced to use canned configurations hand-crafted by experienced roboticists. This inability of ordinary people to specify tasks for robots is inhibiting the spread of robots into everyday life. Even expert roboticists are unable to share solutions in executable forms since there is no commonality of configuration descriptions. Further, a configuration commonly requires significant rework before it can be deployed on a different robot, even one with similar capabilities. The research documented in this dissertation attacks this problem from three fronts.

First, the foundational **Societal Agent** theory is developed to describe how agents form abstract structures at all levels in a recursive fashion. It provides a uniform view of agents, no matter what their physical embodiment. Agents are treated consistently across the spectrum, from a primitive motor behavior to a configuration coordinating large groups of robots. The recursive nature of the agent construction facilitates information hiding and the creation of high-level primitives.

Secondly, the *MissionLab* toolset is developed which supports the graphical construction of architecture- and robot-independent configurations. This independence allows users to directly transfer designs to be bound to the specific robots at the recipient's site. The *assemblage* construction supports the recursive construction of new coherent behaviors from coordinated groups of other behaviors. This allows users to build libraries of increasingly high-level primitives which are directly tailored to their needs. *MissionLab* support for the graphical construction of state-transition diagrams allows use of *temporal sequencing* to partition a mission into discrete operating states, with assemblages implementing each state. Support for multiple code generators (currently existing for AuRA and SAUSAGES) ensures that a wide variety of robots can be supported.

Finally, specific usability criteria for toolsets such as *MissionLab* are established. Three usability studies are defined to allow experimental establishment of values for these criteria. The results of carrying out these studies using the *MissionLab* toolset are presented, confirming its benefits over conventional techniques.

Chapter 1

Introduction

Specifying missions for mobile robots is an arduous and error-prone process currently undertaken only by robotics experts. Several limitations in current methodologies conspire to create this problematic situation. Reliance on text-based traditional programming languages requires that operators be fluent programmers. Poor representations encumbered with low-level details are difficult to create, understand, and maintain. The intermingling of hardware-specific issues with mission specifications impedes sharing of solutions and limits the retargetability of configurations. The research documented in this dissertation tackles each of these problems with the goal to spread robotics beyond the confines of the laboratories.

Missions are currently specified for the majority of robots using either C++, LISP or a variant. This reliance on traditional programming languages limits the ability of robot operators to raise their thought process above the level of programming constructs and design with high-level abstract behaviors. Neither procedural nor object class mechanisms provide a direct fit with the data-flow paradigm and recursive compositions so common in behavior-based robot control software. This forces the designer to add extra code to deal with bookkeeping issues imposed by the language's need to support general programming. Further, using traditional programming languages causes data-movement and control-flow code to dominate the design. This intermingles and hides important details of the mission with overhead issues best managed by a run-time system tailored for robotics. The large amount of code necessary to capture a design caused by using traditional languages also introduces greater opportunities for errors and reduces the readability of the solution. However, the major impact of reliance on traditional programming languages is that robot operators must be able to program in the language, and this eliminates a great number of potential users of robots.

Existing specification methods intermingle hardware-specific issues with the generic portions of the configuration. Since most of a configuration can be designed independently of hardware constraints, this intermingling is unnecessary and counterproductive. A major design goal must be to circumscribe the areas of the configuration impacted by hardware issues, since those reflect a binding of the configuration to a particular piece of hardware. Such a partitioning simplifies the task of retargeting the configuration to different robots and promotes the free exchange of solutions between developers.

Until now the common response to the problems associated with traditional languages has been to define robot-programming specific variants of LISP. The Behavior Language (Section 2.2.2) is the most popular of these languages. Its LISP-based syntax is tailored to support the Subsumption Architecture on small embedded microprocessors. A second LISP-based language is called SAUSAGES (Section 2.2.6). Using SAUSAGES, the user specifies the mission in the form of a graph. Each link represents a behavior which, when executed, moves the robot from one state to another. SAUSAGES requires the operator the understand the LISP syntax (*i.e.*, nested parentheses) in order to specify missions. However, this language does provide for a more abstract specification than the Behavior Language.

The research presented in this document presents a language tailored for describing robot configurations. It describes the instantiation and interconnection of behaviors but defers implementations of behavioral primitives to appropriate programming languages. This allows the configurations to be transported to any run-time system which supports the primitives used in the mission. Hardware bindings are made explicit and attached only through a separate binding step, after the configuration has been developed. This Configuration Description Language solves the problems which have been noted with traditional programming languages and their variants.

Merely developing a better specification language is not sufficient to enable non-programmers to create missions. A graphical programming tool which allows users to visually create missions would empower this group to create missions without learning the syntax and semantics of a textual programming language. A graphical programming tool developed for the manufacturing robot domain is the Onika system (Section 2.3.2). Onika includes a simulation system for evaluating control programs targeted for robot arms, but it does not include support for simulating or commanding mobile robots. Graphical program specification systems also exist in other domains. Khoros (Section 2.3.1) provides a graphical means to specify chains of image processing operations. ControlShell (Section 2.3.3) is a commercial graphical programming toolset which is used to construct complex real-time control systems.

This dissertation presents the *MissionLab* toolset which has been created as an integrated development environment for robot programming. *MissionLab* goes far beyond the state of the art by providing a graphical editor specifically tailored for creating and visualizing multiagent missions and supporting explicit hardware binding and the creation of architecturally-independent designs. A multiagent simulator and operator console allow users to deploy and evaluate configurations both in simulation and on supported mobile robots from within the toolset. *MissionLab* was shown to be a powerful design tool in usability experiments, even in the hands of people unfamiliar with robotics and programming languages.

1.1 The Research Questions

1. **How can the difficult task of constructing robot configurations be simplified to allow operators to easily specify complex robot missions?**

   The need for a design methodology to ease the process of generating robot configurations is clear. Reactive systems are becoming both more prevalent and more complex as they gain competence through the addition of behaviors. It is difficult for humans to predict what the performance will be without experimentation, causing problems when the system must operate in hazardous or unique environments. Robots are generally optimized for expected environmental conditions, requiring modifications when the environment changes. Current design techniques require specifying configurations using traditional programming languages such as C and LISP. This increases the effort required to modify configurations and requires robot operators to be fluent in such languages. A methodology is required to formalize and automate the configuration task if progress is to continue. A goal of this research was to create such a methodology and develop tools based on that methodology which empower non-programmers to generate and evaluate configurations.

2. **What is the best representation of a configuration to support retargeting and hardware independence?**

   Attempting to represent configurations using general purpose programming languages is counter-productive. Such languages require an inordinate amount of code to specify data movement and task scheduling issues best left to the robot run-time system. A new language tailored to the peculiarities of robot configurations is necessary to cleanly support the recursive composition of agents found in complex configurations. A goal of this research project was to identify common characteristics of robot configurations and develop a theory of societal agents. With such a theory in hand describing recursive structures which transcend individual configurations, a language was developed capturing these important design patterns.

3. **How can a robot mission specification toolset best be evaluated as to its usability?**

   It is important when tools are presented that there exist methods and procedures to evaluate their usability and utility. The existing research in usability analysis is generally geared for office environment tasks. In these cases the test participants clearly understand what it is they **want** to do; the question to be evaluated by the test procedures is whether they are able to figure out **how** to accomplish the task using the toolset. With robotics toolsets, this simple analysis is insufficient. When specifying a robotics mission, what is needed is a toolset which allows users to incrementally build a mission and to easily evaluate their solution. Since the task performed is so different from office systems, the procedures used to evaluate these systems must correspondingly be reworked. Therefore, an important research goal was to develop criteria and experimental procedures for evaluating robot mission specification toolsets.

1.2 Research Overview

The foundation of this research is the **Societal Agent** theory which describes how agents form abstract structures at various levels in a recursive fashion. It expands on the notion of an agent developed in "The Society of Mind"[58] to extend beyond the confines of an individual. It is as valid to describe a flock of geese flying in a V as a single agent when describing their speed and heading as it is to discuss the energy level of an individual goose or even the primitive flocking motor behavior. It is all a matter of which level in the flock-of-geese structure is being analyzed.

The Configuration Description Language (CDL) was developed to capture the recursive composition of configurations in an architecture- and robot-independent fashion. CDL supports a compact, exact description of individual robot configurations as well as the interactions of societies of cooperating mobile robots. The language is used to specify the *configuration* of behaviors and not the implementation of behaviors. This allows construction of generic configurations which can be bound to specific robots through an explicit binding step.

The *MissionLab* integrated development environment was developed as part of this research to simplify behavior-based robot mission specification. *MissionLab* includes an interactive designer which allows the user to graphically specify robot missions. *MissionLab* is based on CDL and supports the uniform representation of components inherent in that language. Support for explicit binding and multiple code generators allow *MissionLab* to move beyond the confines of individual laboratories.

Using *MissionLab*, the difficult task of constructing robot configurations can be simplified by applying an object-oriented approach to the design of the mission. The editor is structured around the idea of recursive composition of components using the assemblage construct. This allows building high-level components tailored for a specific domain from existing behaviors. These assemblages can be re-parameterized and used in other parts of the mission or archived for use in subsequent projects.

The temporal sequencing methodology is also directly supported by *MissionLab*. This allows the user to partition a task into discrete operating states and create several smaller designs which each implement a single state. This temporal divide-and-conquer strategy fits well with the assemblage construct, with each operating state in the design generally mapping to a single assemblage.

A major benefit of *MissionLab* is the partitioning of development tasks along knowledge boundaries. The robot operator is able to create missions by selecting and parameterizing behaviors from a list of high-level domain-specific primitives developed by support personnel knowledgeable in robotics. In turn, these assemblage developers are supported by programmers experienced in the development of primitive robot behaviors. The partitioning of the development cycle in this fashion allows a few skilled developers to support a large number of robot operators. Further, it allows robot operators skilled in the particular application domain to command robots without knowing programming languages or even the intricacies of assemblage development.

1.3 Structure of the Dissertation

The dissertation explores work related to this research in Chapter 2. Specification languages similar to the Configuration Description Language (CDL) are reviewed. Since CDL is intended to target multiple run-time architectures, potential target architectures are surveyed to highlight any limitations they may impose on designers targeting them from CDL. Finally, graphical programming tools similar to the configuration designer included in *MissionLab* are discussed.

Chapter 3 presents the **Societal Agent** theory which forms the theoretical basis for this work. This theory develops a uniform representation of societies of agents which spans the range of complexities from individual motor behaviors to coordinated groups of robots.

Chapter 4 presents the Configuration Description Language used to represent configurations by the *MissionLab* toolset. CDL is based on the **Societal Agent** theory and captures its uniformity of representation in a concrete language. Complete syntactic and semantic definitions of CDL are presented.

Chapter 5 presents the *MissionLab* toolset. Use of the graphical configuration editor, multi-agent simulation system and multiple code generators is presented. Several missions are developed and their execution documented to further this presentation.

Chapter 6 designs usability experiments suitable for evaluating a robot toolset, such as *MissionLab*. Criteria are developed to allow rating the usability of a toolset. Three experiments are described which allow establishing values for the usability criteria.

Chapter 7 documents the usability studies completed as part of this research to evaluate the *MissionLab* toolset. The experiments designed in Chapter 6 were performed with a group of participants and the results used to establish values for the usability criteria.

The summary and conclusions in Chapter 8 complete the body of the document. Appendix A reproduces the handouts used in the usability experiments.

Chapter 2

Related Work

No work of substance is accomplished in a vacuum and this effort is no exception. In this chapter we will review the body of literature relevant to this research, taking care to highlight facets that were used as stepping stones and the shortcomings of the existing methods which prompted this undertaking. Although it is difficult to set the stage for related work without first presenting the research which causes it to be relevant, an early presentation of the literature has been chosen to provide context for subsequent developments. Therefore, we now preview Chapters 4 and 5 to provide a basis for the work selected for presentation in this chapter.

A major facet of the software developed in conjunction with this research is the Configuration Description Language (CDL) presented in Chapter 4. CDL is used as the underlying representation of the robot configurations in the *MissionLab* toolset. Therefore, in this chapter we will examine related specification languages to determine how they compare to CDL in purpose, power, and utility.

CDL will be presented as a language which allows specifying robot configurations in a generic fashion with subsequent binding to particular robots and run-time architectures (*e.g.*, AuRA, SAUSAGES). In this chapter we will survey the important robot architectures with an eye towards their suitability as targets for CDL. What limitations would be imposed on designers targeting those systems from within CDL?

A visual programming tool called the Configuration Editor is presented in Chapter 5 for manipulating configurations specified in CDL. In this chapter we will survey other visual programming systems to facilitate later comparisons, to gain an understanding of the niche that the Configuration Editor must fill, and to position it against the state of the art.

2.1 Related Specification Languages

Since part of this research involves development of a configuration specification language, it is important to survey existing languages which CDL attempts to displace. Many of the target architectures which will be surveyed in Section 2.2 also include their own programming language. Most of these languages have been created specifically for robotic applications[11, 18], while the remainder are generally extended versions of traditional languages[15].

CDL differs from these languages in several ways. First, it is architecture- and robot-independent, which none of the others are. This allows the construction of configurations which transcend individual robot constraints. Second, it incorporates recursive combination mechanisms to facilitate information hiding and construction of high-level primitives. Finally, it relies on explicit hardware binding to prevent hardware issues from diffusing into the behavioral design. We now examine several of the related specification languages to more closely position them against CDL.

2.1.1 REX/Gapps

The REX/Gapps architecture[36] partitions perception from action and utilizes horizontal decompositions, allowing complicated perceptual processes to be shared by multiple motor modules. REX[37] is a LISP-based language for describing situated automata. Off-line, the REX program is compiled into a synchronous digital circuit which can be executed to implement the specified system. Gapps[38, 39] is a declarative language used to specify the goal-oriented planning component of a mobile robot. The output from the Gapps compiler is a REX program which is then compiled into a circuit for execution. The circuit model allows semantic analysis to be performed to formally prove run-time properties[67]. By viewing the digital elements embedded within the circuit as implementations of logical predicates, it is possible to analyze the epistemic properties of the network (*i.e.*, the knowledge embedded within the REX program). It is also possible to analyze the transfer function of the circuit to determine performance with respect to sample environments.

Gapps is a language for specifying goals for a robot. Figure 2.1 shows an example goal definition written in Gapps. This example defines the goal of the robot having both the saw and the hammer simultaneously. A typical Gapps program would have a large collection of such goals defined and the mission specification would simply be a expression specifying which goals are to be carried out.

Neither REX nor Gapps provides support for information hiding as does CDL with its recursive composition of components into new components. The generated synchronous machines must have universal applicability since failure mechanisms are not included. Exceptions are expected to trigger higher-level processes to determine appropriate actions.

Support for multiagents or a mechanism for capturing inter-agent communication is not specified. Methods for activating/deactivating modules as their utility changes are also absent. Although REX can be formally analyzed, that advantage is lost

```
(defgoalr (ach (have hammer) (have saw))
   (if (have hammer)
      (ach have saw))
   (if (have saw)
      (and (maint have saw)
         (ach have hammer))
      (if (closer-than hammer saw)
         (ach have hammer)
         (ach have saw))))
```

Figure 2.1: Defines a Gapps goal to possess a hammer and a saw simultaneously (After [39], page 43). The `ach` function defines goals to be achieved, and the `maint` function defines a goal to maintain a particular state. The `have` perceptual predicate tests if the robot is holding the object.

because it requires a detailed environmental model, which is unreasonable to expect to exist for all but the most trivial cases.

2.1.2 RS

The Robot Schemas (RS) architecture[50] is based on the port automata model of computation using synchronous communication. Primitive sensorimotor behaviors are called basic schemas, schemas without input ports represent sensors, and actuators are modeled as schemas without output ports. A group of basic schemas can be interconnected using communication links to form an assemblage. An assemblage is a network of schemas which can be treated as a single schema in subsequent constructions. The assemblage mechanism facilitates information hiding, modularity, and incremental development. An example RS statement[50] is shown below.

$$\mathbf{Jmove}_{i,\overline{x}}()(x) = [\mathbf{Jpos}_i()(x), \mathbf{Jset}_{i,\overline{x}}(x)(u), \mathbf{Jmot}_i(u)()]^{C,E}.$$

The example describes a simple position servo using the sensor **Jpos**, the computation schema **Jset**, and the actuator **Jmot**. $\mathbf{Jpos}_i()(x)$ has no inputs, a single output x, and this instantiation has been parameterized to read the position of joint i. $\mathbf{Jmot}_i(u)()$ has a single input u, no outputs, and has been parameterized to control the motor for joint i. $\mathbf{Jset}_{i,\overline{x}}(x)(u)$ is the computational connection between sensing and action for joint i. It takes the input x, computes a transfer function to drive the joint location to the target value $\overline{x}$, and outputs the corresponding motor signal as the output u. The square brackets represent the assemblage construct. The three schemas are interconnected using the network C (**Jpos** output → **Jset** input, and **Jset** output → **Jmot** input). The port equivalence map E specifies how the **Jmove** ports map to the assemblage member's ports. In this case, the output of **Jpos** is the output for the assemblage. The assemblage **Jmove** can be treated as a single schema performing the servo task. When invoked for a specific joint i, the target position is set to $\overline{x}$, and the output of **Jmove** is the current location of the joint.

The computational model that the RS language embodies is rigorously defined, facilitating formal descriptions of complex robotic systems in RS. Unfortunately, the synchronous computational model of RS is not the most natural for mobile robotics since the real world is both asynchronous and noisy. RS can describe such systems, but at a loss of clarity in the description. RS does not provide mechanisms for expressing coordination between multiple robots cooperating on a task. This research expands on the concept of recursive composition of sensorimotor behaviors apparent here in the assemblage construct.

2.1.3 The Robot Independent Programming Language

The Robot Independent Programming Environment (RIPE) project at Sandia Labs[57] uses C++ classes to implement primitive operators. These classes are the task-level primitives which constitute the Robot Independent Programming Language (RIPL). To use RIPL, the designer writes a C++ program using those classes which are relevant to the project. This system is easily extended and specialized as new devices are made available.

The major drawback of such a system is that designers must be fluent C++ programmers in order to use RIPL. This rather loosely defined language also suffers from the effects of intermingling the mission descriptions with the primitive implementations. There is no attempt to maintain a generic description of the mission and retargeting different hardware will require significant effort.

2.1.4 The SmartyCat Agent Language

The SmartyCat Agent Language (SAL) developed at Grumman[47] is based on the Common LISP Object System[7]. SAL is similar to CDL in drawing heavily from the Robot Schemas (RS) architecture[50] and the Society of Mind theory[58]. In SAL a configuration is a data-flow graph of port automata nodes connected with communication links. The *Agencies* construction provides support for hierarchical specification of complex objects. A graphical user interface for constructing the dataflow graphs is mentioned in [47], although no description of its functionality was located.

SAL co-mingles the configuration with the specification of the primitives and buries hardware bindings within the implementations of individual primitives. Coordination is distributed within the primitives, making it difficult to understand and modify policies. These limitations combine to impair the ability of the designer to enhance and debug configurations as well as retarget designs for different robots.

2.1.5 Discrete Event Systems (DES) Theory

The theory of Discrete Event Systems(DES)[65, 64] models systems as finite state automata where inputs are in the form of *observations* (perceptual information extracted by sensors) and outputs are termed *actions* (actuator commands). The perception-action cycle is broken into discrete *events* where an event is an abrupt transition in the system state, either in response to an action or observation. DES simplifies some problems with analyzing robot controllers by making the robot/world interactions discrete. Several researchers have begun using DES techniques to analyze situated behavior-based robot controllers.

RS-L3[49] is a subset of RS which has been implemented as a robot programming language. Experiments using RS-L3 have been conducted to control an intelligent robotic workcell which groups sub-assemblies into kits for traditional robots to assemble. RS-L3 is able to capture the specification of the robot control program as well as situational expectations, allowing reasoning over the system as a whole. Since RS-L3 uses a synchronous model of computation, it is able to use the analysis methodologies developed for DES. In this case, an *evolve* operator is defined which enumerates the set of possible world states which could be achieved by executing the program (the scnset). The scnset can then be analyzed to answer questions about the effectiveness of the control program situated in the specified environment. The problem with this method is that it requires explicit statement of how all relevant world properties can change both asynchronously and in response to actions performed by the robot.

An example is presented of a robot arm making one of four types of kits based on which parts arrive on a conveyor. This is a vastly more constrained domain than a mobile robot ranging over an unstructured environment. It seems impractical to extend this work to general unstructured environments because of the resulting representational explosion.

A small team architecture for multiagent robotic systems[42] has been developed using DES theory. Behaviors are specified as FSA's by enumeration of their operating states and the set of events causing transitions from one state to the next. By describing everything in this terminology it is possible to prove that the resulting system is *controllable* (any operating state of the system can be reached) using DES techniques. Thus, DES theory provides analysis tools for complex systems.

2.1.6 Prepositional Linear Temporal Logic

A declarative representation for cooperating robots[72] has been developed in the distributed AI community. The authors concentrate on the pursuit problem where multiple cooperating blue agents attempt to surround and thus capture the red agent. The exercise takes place on a finite playing field delineated with a square grid. A language based on Prepositional Linear Temporal Logic (PLTL)[16] was developed for representing problems in this domain. The state of the world, the agents, and their capabilities are each described in PLTL. A simulation environment was created which reads these descriptions and simulates the pursuit problem to determine a winner. This allows changing agent capabilities and then checking the impact of the modifications.

The example system is a frame-based construction built on top of the Cyc[46] database. This leads to a declarative representation where, for example, the perceptual capabilities of a robot could be represented as:

1. $G\left(|B'_x - R_x| + |B'_y - R_y| \leq 2\right) \rightarrow (B'.R_x = R_x) \wedge (B'.R_y = R_y)$
2. $G\left(B'.B'_x = B'_x\right)$
3. $G\left(B'.B'_y = B'_y\right)$

Line 1 states that if agent B' is within two units of the red agent R, then B' will know exactly the x, y location of the red agent. Lines 2 and 3 state that the robot will always know its true location.

The PLTL research is striving to provide tools for describing distributed AI systems, including agent reasoning and communication abilities. The goal is to provide tools for describing systems of cooperating agents solving problems in a distributed manner.

Conversely, CDL is concerned with behavior-based approaches and emergent computation. It does not model the reasoning abilities of agents because behavior-based agents don't reason, they react. What CDL represents are the basic sensorimotor behaviors along with their associated coordination processes. However, the representational language PLTL provides formalisms which may prove useful in future efforts.

2.1.7 Multivalued Logic Integration of Planning and Control

The use of multivalued logic provides a mechanism capable of supporting formal analysis of the process of combining behaviors. Techniques are presented[68] for formally describing what happens when behaviors are combined in various fashions. Each behavior has a particular context with which it is associated where the context circumscribes the set of environmental states for which the behavior is applicable.

When the robot operates within the behavior's context, the behavior can knowledgeably rank possible actions as to their desirability with respect to the behavior's goals. Outside of this context the behavior is indifferent and provides no information.

Given that the context can be defined (a difficult problem in general), multivalued logic provides mechanisms to determine the context of the resulting behavioral assemblage. Three combination operators are considered: Conjunction, Blending, and Chaining. Conjunction creates a new object whose context is the intersection of the conjoined behaviors' contexts. This relates to perceptual fusion, in which a more focused behavior is created using multiple perceptual inputs. Blending is cooperative coordination where the context is the union of the components. Chaining is sequenced coordination where the second behavior gains control when it becomes applicable.

An implementation of the multivalued logic architecture is used to control the robot Flakey where the control program takes the form of a fuzzy logic rule-based system. For example, the following rules implement hall following with obstacle avoidance:

1. IF obstacle THEN Keep-Off (ObstacleGrid)
2. IF at (Corridor1) $\wedge$ $\neg$obstacle THEN Follow(Corridor1)

Rule 1 says that if an obstacle is detected, then the `Keep-Off` behavior will cause the robot to avoid the area associated with the obstacle. Rule 2 says that if the robot is at `Corridor1`, and there are no obstacles present, then the `Follow` behavior will move the robot along the corridor. Recalling that these are fuzzy rules a blending of behaviors results based on how certain the system is about the obstacle. That is, both rules (and their behaviors) will be active when the value of `obstacle` is between 0 and 1.

Multivalued logic provides formalisms capable of describing the results of behavioral composition (coordination), allowing formal analysis of configurations as to their performance. This application of multivalued logic to planning requires an explicit high-fidelity model of the environment to support the analysis.

2.1.8 Petri nets

Petri nets are specified with a set of places (nodes), a set of transitions between nodes, a set of input conditions under which each event can occur, and the effects of the occurrence of each output event. Each place is allowed to contain zero or more tokens, where a token is an arbitrary piece of information. The petri net functions by moving tokens from one place to another in response to input events.

A three layer hierarchical architecture has been developed which uses petri nets as the middle layer[45, 77]. This "coordination" layer receives sequences of tasks to be executed from the higher layer and is responsible for coordinating the execution of the tasks by the bottom layer. Figure 2.2 shows a graphical representation of a petri net used to coordinate the execution of a vision algorithm to locate a particular spot in an image.

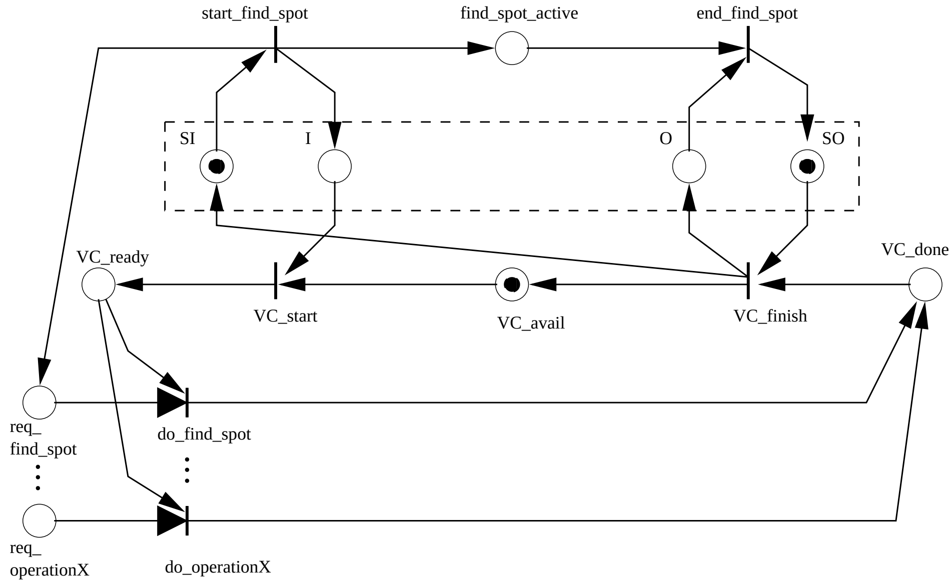


Figure 2.2: Graphical representation of a petri net to coordinate a vision algorithm which finds a particular spot in an image (After [45], Figure 5).

The petri net is able to capture the unsynchronized movement of data in complex control structures. This ability to represent data movements between finite state automata is useful for describing coordination procedures. Petri nets are also useful for describing how coordination procedures are related by providing a common representation for varied coordination algorithms. However, petri nets do not support the recursive construction of components and thus eliminate hierarchical designs.

2.1.9 A Language For Action (ALFA)

ALFA[20, 18, 19] (A Language For Action), with its roots in the subsumption architecture, is used to describe behavior-based robots using a LISP-like syntax. It uses a data flow architecture consisting of named computational modules connected with a network of named communication channels. Channels are limited to transmitting one analog (pulse width modulated) voltage which in practice limits the information content that channels can transmit to small integer values (0-255 or perhaps 0-1023).

Channels combine inputs from several modules by using the minimum input value, the maximum input value, the average of all input values, or the value from the highest priority input as the output value of the channel. The particular combination algorithm to be used is specified in the channel definition. Modules are parallelizable because the computation of each command is disjoint. Outputs to channels are set with a command of the form *(DRIVE channel_name expression)* and input channels are referenced by name. For example, *(DRIVE chan_1 chan_2)* is a null process which passes the value of the input channel *chan_2* to the output channel *chan_1*.

ALFA has been used to specify the reactive execution component of a hybrid architecture (ATLANTIS). In this usage, higher-level control is injected by setting up channels which read their inputs from blackboard (global) variables or by a higher-level process actively enabling and disabling modules as their usefulness changes.

ALFA is more of an operating system and design methodology than just a language for describing mobile robot behaviors. It is a good example of how higher-level control can be injected into a reactive execution component of a hybrid architecture. The use of blackboard variables as perceptual inputs from a planner allows construction of a persistent configuration for the reactive component where sequenced coordination activates major subsets of the configuration based on inputs from a deliberative planner.

2.2 Potential Target Architectures

When one begins turning a machine into a taskable robot, one of the first decisions made is the choice of a run-time architecture. Unfortunately, the current situation is that robots are generally delivered from the factory with only minimal software support. Even more disturbing is that, in response to this shortcoming in support, nearly every user develops their own run-time architecture. The resulting fragmentation severely limits the exchange of software modules and the ability of users to collaborate on solutions.

This confusion has led to a major goal of this research: development of a generic specification language which is robot and run-time architecture independent. Being able to specify generic configurations which can later be bound to one of several supported architectures will allow users to raise the design process above hardware and architecture issues.

There exists a large body of literature describing architectures for behavior-based robotics. We now examine several of the better known robot architectures with an eye towards ones which could be supported as targets under this scheme.

2.2.1 The AuRA architecture

The Autonomous Robot Architecture (AuRA)[3, 2] is the platform in common use in the Georgia Tech mobile robot lab and is the system from which the *MissionLab* toolset grew. Therefore, it is natural that AuRA was the first architecture targeted by the CDL compiler and that an AuRA-based simulation system is incorporated into *MissionLab*, allowing a close interaction between specification and evaluation when that architecture is targeted.

The AuRA architecture is a hybrid system which includes planner, navigator, and pilot components spanning the range from deliberative to reactive execution. The pilot is the low-level reactive execution module which grounds the other modules in physical devices. This behavior-based module is the target of the *MissionLab* system when AuRA is the target architecture.

AuRA uses a vector-based approach to inter-behavior communication where motor behaviors generate a vector denoting the direction and speed they would like the vehicle to travel. Cooperative coordination occurs by summing the vectors to get a composite vector for the group and competitive coordination occurs by ignoring the desires of losing behaviors.

In configurations generated by *MissionLab*, we rely on the human to replace the deliberative system. In this case, the operator crafts a suitable behavioral assemblage which can be instantiated and executed to complete the desired task. These configurations commonly utilize temporally sequenced coordination to encode the operator's knowledge of appropriate performance changes based on run-time perceptual feedback.

2.2.2 Subsumption architecture

The subsumption architecture[9] is probably the most widely known behavior-based mobile robot architecture. Sensations arriving on channels from sensors are processed by behaviors which transmit actions on their output channels to actuators for execution. Prioritization of behaviors is handled through gating operators on input and output data streams. A behavior may control the operation of lower priority behaviors by either overwriting their normal input data streams or overwriting the data they output with its own values. Incremental development is handled by adding new layers of competence on top of existing layers in an evolutionary way. Each layer can consist of several individual behaviors and normally embodies a single higher order skill, such as *Avoid_obstacles*, *Follow_walls,* or *Exploration.*

The subsumption architecture has been used to construct complicated mobile robots[10] as well as societies of robots[55, 56]. However, the subsumption architecture remains more of a design philosophy than a formal specification. The Behavior

Language[11] is the LISP-based parallel programming language used to specify subsumption configurations. Subsumption is rather restrictive in that all coordination occurs via prioritized competition, precluding any cooperative interaction between behaviors. Subsumption is prone to becoming unwieldy and difficult to debug as new layers are added, since there are no restrictions on how upper layers interact with lower layers. There is no mechanism to support information hiding since the layering scheme is transparent. In general, it is not clear that layering is always the best partitioning of control. Mechanisms for clustering and managing groups of behaviors within layers are helpful (perhaps even necessary) as the task complexity grows. It is possible to describe subsumption-style configurations within the language we present, but they would need to be restricted to using only competitive coordination.

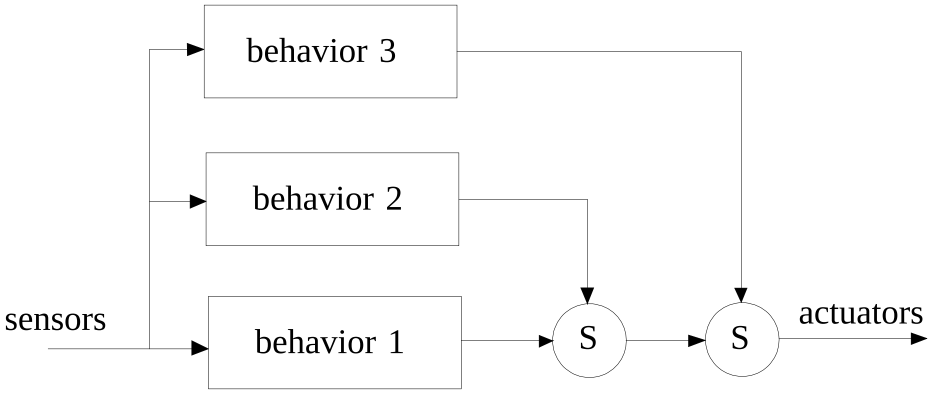


Figure 2.3: Example Colony Architecture Network.

A variant of the subsumption architecture is the colony architecture[14]. This is an important target since it has also been used to drive non-trivial robots. The colony architecture modifies subsumption in several ways: First, it removes the ability of behaviors to overwrite the inputs of other nodes and relies exclusively on inhibition at outputs using suppression nodes. Second, messages are transmitted continuously on connections while a given command persists, instead of being latched and presumed active until overwritten, allowing suppressed behaviors to regain control as soon as the inhibition is removed. As in the subsumption architecture, individual modules are interconnected into a hierarchy using a fixed priority arbitration network. When two modules are both generating output values, the one with the higher priority will suppress the output of the lower priority module. Figure 2.3 shows an example network where **S** is a suppression node.

The colony architecture moves subsumption towards a data flow architecture by removing the input inhibition nodes and constantly outputting messages instead of latching state information. The colony architecture also relies exclusively on priority-based competitive coordination and could similarly be targeted as a supported run-time architecture.

2.2.3 Maes' action-selection architecture

Maes' use of action-selection mechanisms for behavior coordination[54, 53] employs a spreading activation mechanism to reactively control situated agents. Individual behaviors are termed *competence modules* after Minsky's Society of Mind theory[58] and represent simple computational agents, encapsulating a particular expertise. Since multiple modules are likely executable at each instant, they must compete for control of the system. The coin of the realm is a module's level of *activation*, and the module with the highest activation gains control.

Individual agents are connected via activation and inhibition links into a complex graph. Agents which directly compete under similar circumstances will inhibit each other in proportion to their level of activation. For example, *pick_up_block* and *put_down_block* agents would try to inhibit each other to ensure that the active agent can finish its task without interruption.

Sensors provide a continuous source of activation from the environment which then flows outward through the behavior graph. In turn, agents will pass on a fraction of the activation they are receiving to those agents which can most help make them applicable (satisfy their preconditions). For example, a *pick_up_block* agent might pass some of its activation to behaviors which would move the system towards a block.

Activation also flows from the current set of system goals provided from a higher-level process to those agents which can most closely achieve the desired actions. Agents receive activation proportional to how applicable they are in the current situation. Ones having their preconditions met will receive the most activation, while others receive less based on how much environmental change must occur before they can execute. This spreading of activation both from goals and the current environmental situation tends to maximize the activation level of agents which are both currently applicable and useful in achieving the system goals.

A large problem with this architecture is how appropriate activation and inhibition links are created between agents. It is proposed in [54] that a second network be used to configure the primary one, functioning as a meta-level planner. However, this merely raises the problem up one level, leaving the issue of how the meta-layer is constructed. Spreading activation implements a distributed competitive coordination mechanism. Behavior-based systems decompose tasks into small manageable behaviors, leaving the major task when constructing a configuration as the selection and parameterization of the coordination mechanisms. Therefore, a goal for an architecture should be the localization and compartmentalization of behavioral coordination. Using this yardstick, spreading activation falls short.

Spreading activation is interesting to consider as a target architecture since it is so different from other behavior-based architectures. However, since it makes such a strong commitment to the spreading activation coordination mechanism, it is not very flexible as a target. It would be useful to consider ways to support spreading activation coordination in a generic sense, but it is unclear how one would generate compatible configurations using other styles of coordination.

2.2.4 ALLIANCE: The Cooperative Robot Architecture

The subsumption architecture has been used as the basis of the ALLIANCE architecture for controlling a society of heterogeneous robots[63, 60, 61, 62]. Individual agents are guided by subsumption-based control programs using both sensor readings and information received via communication with other robots. Each agent is "selfish" in that it cares only about its own goals and also "lazy" since it is happy to let another robot work on its tasks as long as the tasks are being accomplished. These simple rules allow the robots to cooperate and not interfere with each other while achieving objectives in unstructured environments.

This architecture relies on specialized motivational nodes added to the basic subsumption architecture which selectively enable certain *behavior sets* when appropriate. Each behavior set contains the capabilities to accomplish a distinct high-level task such as *clean_floor* or *empty_garbage*. Behavior sets are defined as conflicting and the motivational nodes are required to ensure that only the most relevant behavior set is active at any time.

The use of the specialized motivational nodes to enable particular assemblages adds a new type of competitive coordination to the basic subsumption architecture. A weakness of this scheme is that there is no way for two or more behavior sets to be active at the same time. Some behavior sets are not mutually exclusive (for example, *clean_floor* and *search_for_key*) and the ability to activate more than one behavior set would be beneficial.

The main distinction between this architecture and subsumption is the addition of the motivational nodes. This coordination adds the ability to turn large chunks of the configuration on and off as they gain and loose applicability. As a target architecture, this makes it more flexible than plain subsumption.

2.2.5 The Distributed Architecture for Mobile Navigation

The Distributed Architecture for Mobile Navigation (DAMN) is used as the behavioral arbiter on the ARPA Unmanned Ground Vehicles (UGV's). DAMN uses a fuzzy logic approach to cooperative coordination. Each behavior has a certain number of votes available and is able to allocate them to the available actions. The action with the most votes is undertaken. DAMN grew out of the Fine Grained Alternative to the Subsumption Architecture[66].

Limitations of the subsumption architecture are that it excludes cooperative coordination mechanisms and internalizes state. Since all coordination occurs via priority-based inhibition nodes, it is not possible for two behaviors to simultaneously contribute to the actions of the robot. One of the major tenets of the subsumption architecture philosophy is that existing layers of competence are not modified and continue to function unchanged when new layers are added. Since higher layers must inhibit lower layers to gain control of the system, designers of new layers must have intimate knowledge of the lower levels that they are subsuming, to ensure inhibition only occurs when appropriate. Also, the new level may require access to internal state information within lower level nodes to monitor its applicability. This can require rewriting the lower levels to make the required state information externally available.

The fine-grained alternative[66] to the subsumption architecture addresses these problems by converting behaviors into a collection of simple transfer functions which are not allowed to contain state information. This transforms a subsumption architecture into a connectionist architecture, where a new layer is added as a cluster of nodes. The network is highly structured, with each node potentially computing a different transfer function. The only restriction is that the inputs and output at each node are real values in the range $-1..1$.

There are several advantages to this architecture over subsumption: Cooperative coordination can now easily occur since the transfer functions operate on all the inputs and are not binary operators like the suppressor nodes in subsumption. All state information is external so new clusters can easily access needed information. A fuzzy arbiter is used to merge the demands of active behaviors, allowing for cooperation as opposed to subsumption's reliance on priority-based competition.

DAMN does not fit well as a target architecture, but it is important nonetheless since it is used as the behavior arbiter on the ARPA Unmanned Ground Vehicles (UGV's)[25]. However, the target architecture for the UGV systems will be at the level of the SAUSAGES interface (described in Section 2.2.6) with the DAMN arbiter functioning as the behavior coordination mechanism.

2.2.6 SAUSAGES

The System for AUtonomous Specification, Acquisition, Generation, and Execution of Schemata (SAUSAGES)[27, 26] provides a behavior configuration specification language as well as run-time execution and monitoring support. At run time, SAUSAGES functions as the robot pilot, executing relevant behaviors, monitoring for failures, and interacting with higher-level processes. A SAUSAGES program is reminiscent of a flowchart-like graph, where behaviors are specified as operations which, when executed, move along a link. After executing a link, the link informs the executor which new link to execute, allowing the flow of control to follow a complex directed graph. This flow of control over the links is implementing sequenced coordination. Since each link can be constructed from sub-links which run in parallel, a form of cooperative coordination is also available to the designer.

What SAUSAGES does not provide is abstraction. CDL facilitates abstraction through recursive composition of components into new components. A variant of SAUSAGES called MRPL is used in the ARPA UGV's. SAUSAGES is supported as a target architecture from CDL. This allows testing configurations constructed with this system on the SAUSAGES simulation system available from CMU and comparison with code developed for the ARPA UGV's as well as close comparisons with the existing techniques in the UGV community.

2.2.7 The Procedural Reasoning System

The Procedural Reasoning System (PRS)[21] is a general purpose reasoning system which is able to exploit opportunism as the robot moves through the environment. PRS consists of a knowledge base which maintains the current beliefs about the world, a set of active goals waiting to be achieved, a set of action plans which purport to achieve goals, and a Short Term Memory (STM) containing those plans which have been chosen for application. The control program adds plans to STM relevant to the active goals with respect to the current environmental feedback and executes the plans in STM. The UM-PRS system[44, 35] is a newer implementation of PRS generated at the University of Michigan for use as a robot mission planning system. It was recoded in C++ to increase speed and to support deployment on mobile robots.

UM-PRS is important since it has been considered for inclusion in the UGV architecture as the behavioral controller. In such an application the action plans used as the planner's primitives would be behavioral assemblages. Likely candidates are behaviors like *follow_road* and *cross_country_travel*. Using this strategy, UM-PRS becomes a planner-based coordination module which at run time selects which assemblages should be active. Based on the point of view of planning as coordination, UM-PRS is not so much a target architecture for this research as it is a possible component to be incorporated.

2.2.8 Reactive Action Packages (RAPs)

Reactive Action Packages[17] (RAPs) are mechanisms to specify reactive programs. Each RAP encapsulates a single action or competency the reactive robot controller is capable of performing, such as *Move_down_hall* or *Load_into_truck*. The RAPs are intended to be used as a set of primitive actions by a deliberative planner which chooses a RAP to activate at each step as part of a plan to fulfill the system goals. Several different methods (strategies) for accomplishing an action will exist within a given RAP. At execution time, one of the methods is chosen as most applicable based on precondition tests. The individual methods are allowed to fail, but are required to cognizantly report any failures. The requirement that methods be self-monitoring allows the RAP to try different methods before the plan itself possibly fails and causes re-planning.

The concept of a RAP matches somewhat with the notion of an assemblage. Each encapsulates various sets of primitive behaviors which are coordinated by some mechanism at run time. RAPs are presented as a way of grouping action packages and specifying coordination between methods which is consistent with assemblages.

RAPs is a difficult target architecture because the coordination mechanisms are distributed throughout the behaviors. Each RAP coordinates itself until failure or success when the planner regains control. Further study is necessary to determine what class of configurations could be deployed on this architecture from within CDL.

2.2.9 Supervenience

Supervenience[73] is a theory of abstraction defining a hierarchy where higher levels are more abstract with respect to their "distance from the world". Lower levels represent the world in greater detail and perhaps more correctly while higher levels represent the world more abstractly, possibly allowing erroneous beliefs to exist. The supervenience architecture is defined as a non-monotonic representation to allow coping with these contradictions. The supervenience architecture is targeted for use in dynamic-world planners.

The Abstraction-Partitioned Evaluator (APE) is one example system based on the supervenience architecture. The author is careful to state that APE is not *the* supervenience architecture, but is instead a particular implementation based on the supervenience ideas. APE is a five level hierarchical architecture implemented in common LISP. Inter-level communication is allowed only between adjacent levels. Communication within APE is accomplished with a blackboard architecture. Monitoring daemons can be attached to the blackboard and trigger on arbitrary changes. The operators contain traditional planner features such as add-lists and delete-lists and the procedural part of each operator is encoded as a Petri net.

The most abstract level is called *Conventional* and represents the system's knowledge of social norms and rules; that is, those things which are true based only on convention. The *Causal* level encodes the system's knowledge of cause and effect. The *Temporal* level can reason about time and contains explicit temporal relations. The *Spatial* level organizes perceptual information based on spatial relationships. Spatial reasoning, such as path planning, also resides at this level. The lowest level is named *Perceptual/Manual* and encapsulates virtual sensors and virtual actuators. Information represented at this level is in the form of individual percepts such as current location and actuator commands (*e.g.*, move three feet forward).

The author implemented a simulated robot using APE which operates within a simulated home. Several tasks are demonstrated to highlight the capabilities of the system. The HomeBot is able to cope with unexpected emergencies unrelated to the currently executing task. If an unexpected obstacle is discovered while the robot is moving along a planned route, and the obstacle is subsequently removed during the replanning process, the robot immediately halts replanning and resumes execution of the previous plan.

Supervenience is the formalization of the process of partitioning a control structure into abstraction levels. It provides guidance which may prove useful to users of the proposed configuration designer: Information containing similar levels of abstraction should be grouped together, and the layers should be ordered such that more exact descriptions are lower.

The APE architecture is a deliberative dynamic-world planner and does not lend itself to targeting from CDL. However, it might be possible to integrate supervenience in some form with CDL as a deliberative planner-based coordination mechanism.

2.2.10 Behavioral Architecture for Robot Tasks (BART)

BART[40] (Behavioral Architecture for Robot Tasks) is an architecture and programming language for specifying and controlling behavior-based mobile robots. BART was designed to allow large, rapid changes in active behaviors. A situation where this is necessary is a patrol robot which hears an unexpected noise, at which time it should switch to a stealth mode so as not to divulge its own position while it investigates the noise.

To facilitate such changes *Task Groups* aggregate individual behaviors into groups based on semantic characteristics such as *noisy-tasks* and *self-preservation-tasks*. Notice that a given behavior may belong to many task groups. A *Task Class* provides a specification for an individual behavior of which the system is capable, such as *move-on-path*. Each task class may have zero or more instances active at any time. However, task instances only execute when they enter the current task mix. The *Current Task Mix* represents those currently executing task instances. The Focus of Attention Manager (FOAM) is responsible for determining which members of the current task mix should be active at each step. To perform this function, each task determines its own utility in the current situation. The FOAM is then able to select the most relevant tasks without intimate knowledge of each task's capabilities. The LISP-based BART programming language is used to specify the individual tasks. Multi-agent control is distributed and explicit. For example, a particular robot may be told to go into formation with another robot.

The interesting facet of the BART architecture with respect to this research is its support for high-level operator input provided by the semantic task groups. These groupings allow operators to specify control in the form of high-level operating concepts instead of low level details. The idea of semantic grouping is a powerful one reflected in the recursive constructions within CDL. A particular assemblage such as *forage* represents a semantic grouping of behaviors useful to accomplish the forage task. These semantically meaningful primitives simplify the creation and maintenance of configurations as well as promote high-level operator input. Unfortunately, it appears that BART was never fully developed.

2.3 Graphical Programming Tools

One of the primary goals of this research project was to empower non-programmers to specify robot missions. The development of a graphical configuration editor was chosen as the best way to accomplish this task. This allows users to add icons to a workspace and connect them into a specification for a robot mission mission without knowing how the underlying system works.

There are several other visual programming tools of note available and a survey here will attempt to highlight the similarities and differences between the *MissionLab* configuration editor and these systems.

2.3.1 Khoros

The inspiration for the graphical construction of configurations in *MissionLab* was the Khoros[76] image processing workbench. Khoros is a powerful system for graphically constructing and running image processing tasks from a collection of primitive operators. The user selects items from a library of procedures and places them on the work area as icons (called glyphs). Connecting dataflows between the glyphs completes construction of the "program".

Figure 2.4 shows the Cantata graphical workbench from the Khoros system. A program has been constructed which (reading left to right) loads an image file, runs a histogram equalization on the image, segments the image using a dynamic thresholding algorithm, and then displays the resulting line segments. Once constructed,

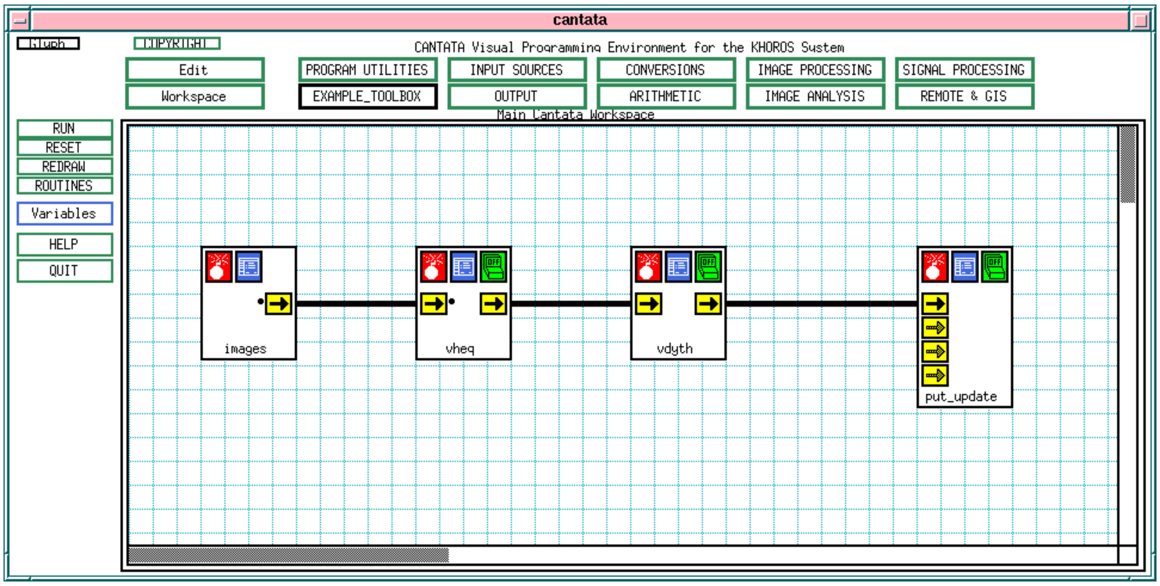


Figure 2.4: Cantata graphical workbench with a program to execute a histogram equalization and segmentation of an image before displaying it.

the program can be executed by clicking on the `run` button and the results will be displayed within a popup window. Each glyph in a Khoros program represents a distinct program which is instantiated as a separate UNIX process at run time.

Visual programming using the data-flow paradigm is a powerful way of describing and presenting image processing algorithms. This idea has been extended to allow the recursive specification of robot control programs in the *MissionLab* toolset.

2.3.2 Onika

Probably the graphical programming environment most relevant to this research is the Onika system[74, 22] from CMU. Onika is optimized for rapid graphical construction of control programs for robot arms. It is tightly integrated with the Chimera real-time operating system, also from CMU. This integration allows Onika to start, monitor, and terminate Chimera tasks based on the operator's manipulation of graphical icons on the desktop.

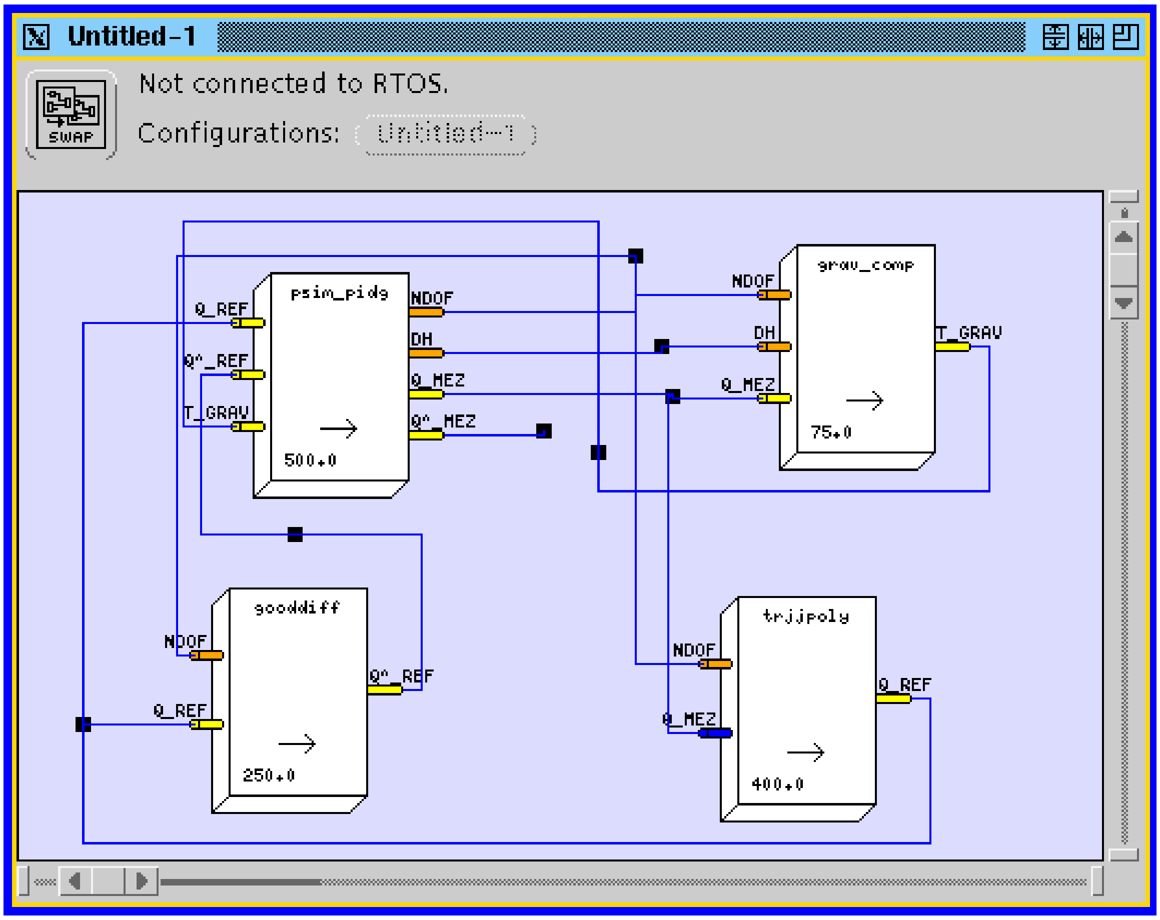
Programs are constructed by placing a linear sequence of icons using a puzzle piece metaphor. Compatibilities between objects are represented on the input and output side of tasks via a different joint shape and color. This physically delimits which tasks can follow each other and is a very good metaphor, especially for casual users. Once programs are constructed, they can be saved to a library for later retrieval and deployment, or executed immediately. Figure 2.5 shows the engineer's view of a robot arm control program loaded into the Onika system. Figure 2.6 shows the programs available for users as puzzle piece icons. Figure 2.7 shows a completed user application.

Onika is network-based and allows components to be included from libraries physically located on remote systems. These symbolic links are then traversed when the configuration is deployed to locate the run-time modules. This facilitates code sharing by allowing the development process to be dispersed.

Onika includes a simulation system for evaluating control programs targeted for robot arms, but it does not include support for simulating or commanding mobile robots. The presentation style is reminiscent of electronic schematic diagrams. Onika modules are allowed multiple output connections while CDL uses a functional notation where each module has only a single output.

2.3.3 ControlShell

ControlShell[70] is a commercial graphical programming toolset from Real-Time Innovations which is used to construct complex real-time systems. It is similar to the Engineering level of Onika (*e.g.*, Figure 2.5) and presents the same electronic schematic-like look and feel. A data-flow editor is used to graphically select and place



Untitled-1

Not connected to RTOS.

Configurations: Untitled-1

Figure 2.5: The engineer's view of the Onika graphical programming system with a program loaded for controlling a robot arm. (www.cs.cmu.edu/afs/cs.cmu.edu/project/chimera/www/logos/Onika_sim_conf.gif)

components into the workspace, and connect them into control systems. The code generator and operating system components allow deploying on embedded systems supporting VxWorks. The state programming editor supports graphical specification of state transition diagrams which complete the desired tasks.

ControlShell has several features important for embedded systems: It uses small grained concurrency where modules executing at the same rate are grouped into a single process. It supports transparent network communications. The run-time system preserves module names, allowing dynamic binding at run time.

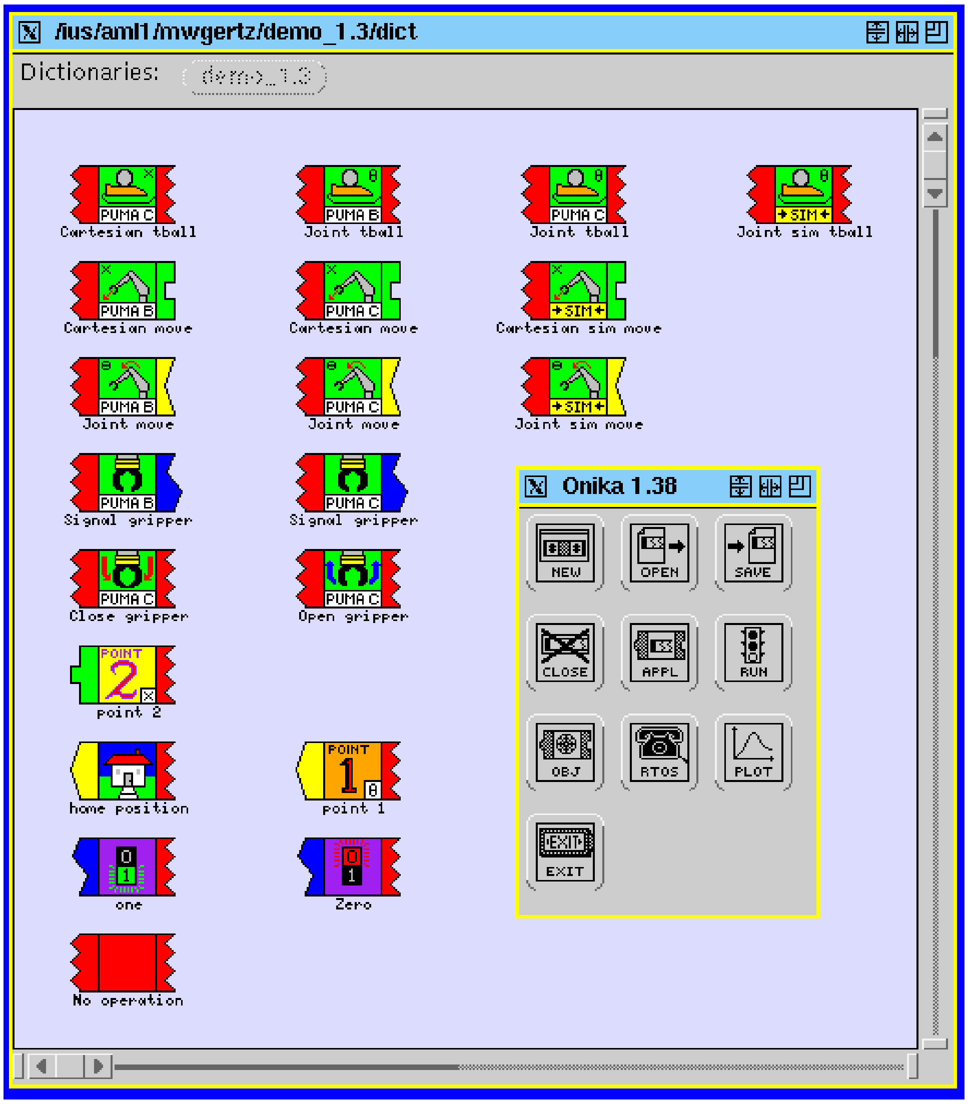
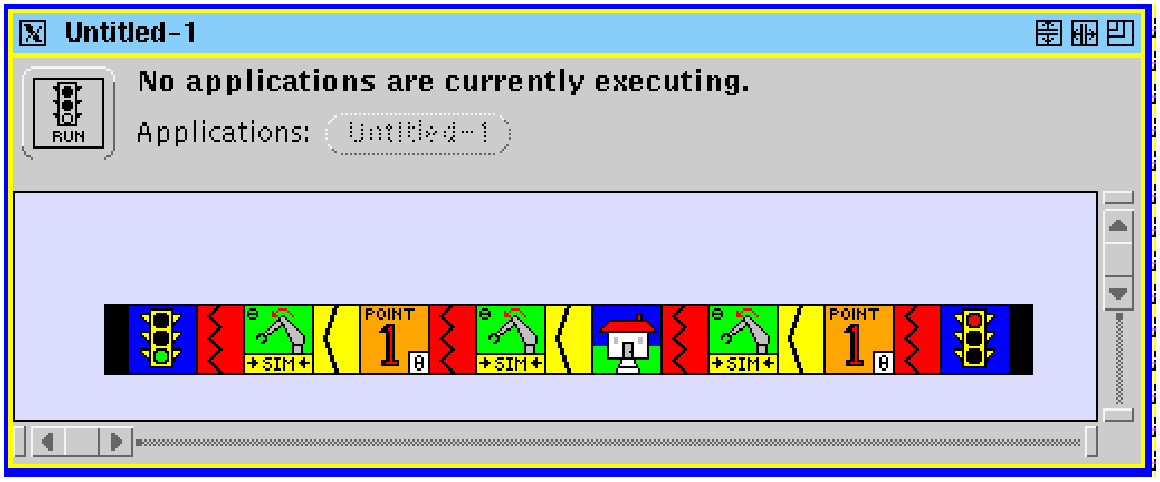


Figure 2.6: The available programs in Onika, shown as puzzle piece icons for the user.
(www.cs.cmu.edu/afs/cs.cmu.edu/project/chimera/www/logos/Onika_upper_start.gif)

However, ControlShell does not support recursive construction of components. There exists a single layer of primitive components, a second layer of so called transition modules constructed from the primitives, and finally the state diagram denoting the sequencing of operating states. This lack of support for recursive construction limits reuse and information hiding in complicated designs.



No applications are currently executing.

Applications: Untitled-1

Figure 2.7: A completed user application in Onika which causes the robot arm to move to a user defined point in space, home the controller, and then move to a second point. (www.cs.cmu.edu/afs/cs.cmu.edu/project/chimera/www/logos/Onika_full_appl.gif)

ControlShell is a commercial package targeted towards hard real-time control applications. This is compared to *MissionLab* which is targeted to the mobile robot community. ControlShell does not support explicit hardware binding or support any run-time architectures but its own.

2.3.4 GI Joe Graphical FSA Editor

The GI Joe toolset[8] allows graphical construction and visualization of finite state automata. These state machines can then be exported to a format used by COSPAN for analysis. COSPAN[28] is a specification language and verification system for state-based control systems. The specification language is called S/R and the analysis tool is called COSPAN. The analysis of an FSA in COSPAN determines if the language accepted by the FSA includes the language that the user proposes for test. This allows automatic verification of designs against the corresponding requirements specifications.

Figure 2.8 is a S/R definition of a simple flip/flop state machine. When the current state is `OFF` and the select variable has the value `on`, it moves to the `ON` state. Correspondingly, when the current state is `ON` and the select variable has the value `off`, it moves to the `OFF` state.

```
proc ALT /* Flip/Flop */
    selvar   #:(off,on)
    stvar    $:(OFF,ON)
    init     OFF

  trans
  OFF      {off,on}
      -> ON  : #=on
      -> OFF : else;

  ON       {off,on}
      -> OFF : #=off
      -> ON  : else;
end
```

Figure 2.8: Definition of a flip/flop using COSPAN S/R (After [8], Figure 1).

Figure 2.9 is the graphical representation of the flip/flop shown in Figure 2.8 as it would be constructed in the `GI Joe` editor. From this graphical representation the `COSPAN S/R` definition is generated automatically.

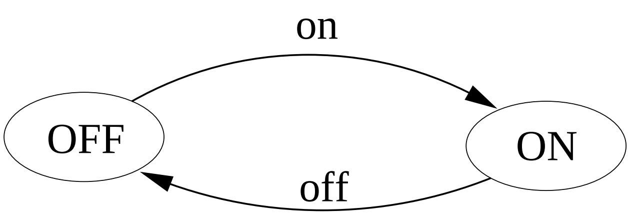


Figure 2.9: The graphical representation of 2.8 in `GI Joe` (After [8], Figure 2).

GI Joe allows running the COSPAN verification system against the FSA's constructed using the editor to check their correctness. It also is able to generate C code for the state machine itself, although there is no provision for specifying the primitives active in each state from within the system. Overall, GI Joe and COSPAN provide support for FSA's but little else. The *MissionLab* toolset goes far beyond these capabilities in providing users with an integrated development environment.

2.3.5 Mechanical design compiler

A mechanical design compiler has been created[79, 78] which, given schematic specifications and a utility function, will generate a design for a mechanical system meeting those goals. The output is in the form of catalog numbers for the selected components which combine to create a design optimal with respect to the utility function. Tested domains include mechanical and hydraulic power transmission units.

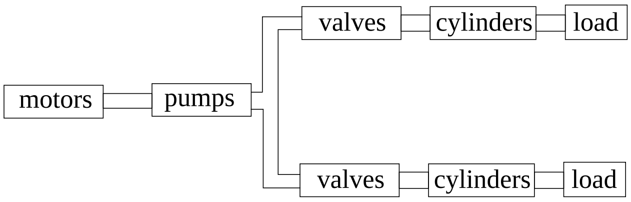


Figure 2.10: Example schematic for a hydraulic power train.

An example[78] will show the application of the compiler. The user would interactively create the schematic for a hydraulic power train, shown in Figure 2.10, using a graphical interface. The mechanical design compiler concentrates on searching catalogs for components which can be connected in a consistent manner to meet the design specifications with competing designs judged relative to the user's utility function.

This work is interesting since it demonstrates a form of automatic design. In this case, a description of the high-level structure of a system along with a specification of the utility function are sufficient for the system to propose a mapping of available building blocks onto the original design, maximizing the utility function. In the example, this involves looking up part numbers for specific devices and ensuring that they all function together correctly.

2.4 Summary

An examination of the work related to any new research project is always necessary to properly position the contributions against the state of the art. However, it is just as necessary to acknowledge the work of others which proved fruitful as building blocks and stepping stones in the new research. In this chapter we have attempted to mention and provide insight into the existing work which was and is important to this research.

There are many architectures in common use for developing robot control software. We have surveyed the most popular architectures to attempt to evaluate which would be suitable target architectures for the CDL compiler. Of the architectures mentioned, AuRA and SAUSAGES were chosen as the first two target architectures. AuRA was a natural choice due to its use at Georgia Tech and the author's familiarity with its reactive execution module. SAUSAGES became important for researchers in the lab when it was chosen as the basis of the language used to describe missions in the ARPA UGV robots. Since the author was working on the UGV project while completing this research, it became natural to look to SAUSAGES as a second target architecture. The availability of a SAUSAGES simulation system also influenced this choice.

The strengths and weaknesses of the other popular specification languages which CDL is competing were also presented. Of the group, CDL is the only one to support the recursive construction of reusable components which may transcend individual robots. Of course, it is difficult to justify claims related to the suitability of a language without an understanding of the target domain. As the next chapters unfold, the points made regarding each of the languages should gain more credence.

The existing graphical programming tools were surveyed and found to consist primarily of special purpose tools targeted to narrow domains. The Onika system is the closest competitor to CDL. It has been developed to support the flexible manufacturing domain and does not provide specific support for mobile robots.

In the final analysis, there are many useful building blocks scattered through the literature. The *MissionLab* toolset is the first to combine these ideas into a multiagent mission specification system usable by non-programmers. Basing the configuration editor on a language which supports explicit hardware binding and recursive constructions extends the capabilities of the users while reducing the complexity presented to the casual user.

Chapter 3

The Societal Agent

Thinking of societal agents conjures up mental images of herds of buffalo roaming the plains, flocks of geese flying south for the winter, and ant colonies with each ant seemingly performing exactly the task that will provide the maximum utility to the colony as a whole. Human examples tend more towards hierarchies, with the prime examples being large corporations and military organizations. In each of these example societies, the components are physical objects such as animals or humans.

Using Minsky's powerful "Society of Mind" representation, each buffalo, goose, ant, and human can be thought of as possessing a behavior-based controller consisting of a society of agents. This leads to the view of a flock of geese as a huge society with thousands of interacting agents. Within this society, nature has drawn boxes around collections of agents and said, "These agents physically comprise a goose." Recognizing each individual primitive behavior as an autonomous agent is generally intuitive. However, it is sometimes a struggle to accept the description of coordinated societies of these agents as cohesive agents in their own right. These higher-level, more complex agents are as concrete as the primitive behavioral agents.

This abstraction is equally apparent in military organizations. When commanders refer to their command they don't speak of individuals, but the unit abstractions. A company commander might ask for "the strength of Bravo platoon" or "the location of Alpha platoon", but rarely refers to a particular soldier in one of those platoons. The hierarchical structure of military units is intentional. A squad consists of specific members who live and train together as a group until they form the cohesive unit called a squad. The squad has specific commands that it can respond to such as "deploy at location Zulu" or "attack objective Victor". Squads are intended to be as interchangeable as possible, in that they present the same responses to a command as any other would. All of this serves to abstract the group of soldiers into a "squad", a high-level agent which is as cohesive and concrete as an individual soldier.

As a second example of complex agents consider the well-documented sexual behavior of the three-spined stickleback[75] shown in Figure 3.1. As the schematic shows, the sexual behavior involves a complex temporal chain of behaviors which transcends the individual male and female fish. The arrival of a female showing the "ready to spawn" display signs triggers the male to do a zig-zag dance, which triggers the female to swim towards the male, which triggers the male to lead her to the nest, and so on. The individual behaviors such as the zig-zag dance, follow, and show-nest are in fact individual agents within the **Societal Agent** representation. A coordination operation transcending the individual fish uses these primitive agents as operators to create the sexual behavior apparent in this example.

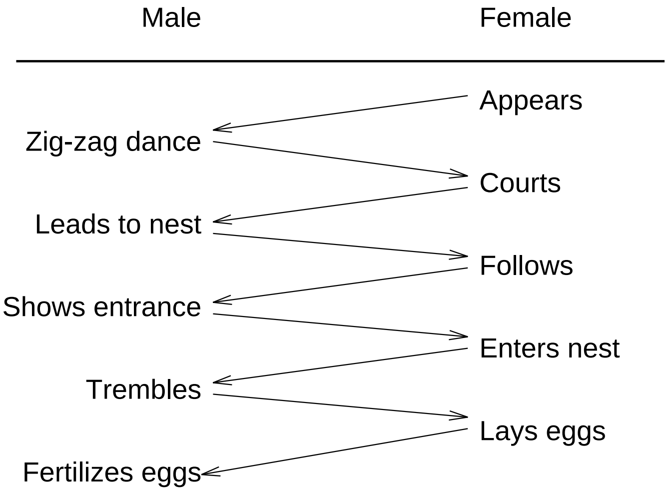


Figure 3.1: Sexual behavior of the three-spined stickleback (after [75]).

Now consider how one would specify a multiagent robotic society capable of exhibiting this mating behavior. A design can be implemented and tested to determine its validity, as opposed to explanations of biological systems which are difficult to validate. Figure 3.2 shows a schematic of the behaviors and coordination operators active during the stickleback mating behavior. Level a shows the representation of the reproductive agent. While this behavior is dominant, the two fish are functioning as a single coherent agent, much as one would speak of a herd of buffalo or a marching band as a cohesive unit having substance, direction, and purpose. This is decomposed in Level b to show the two individuals. Level c shows the various operating states present in each of the two fish to support the mating ritual.

The linear chain of behaviors shown in Figure 3.1 can be represented as a Finite State Automaton (FSA) using the methods of Temporal Sequencing [4]. Temporal sequencing formalizes methods for partitioning a mission into discrete operating states and describing the transitions between states. The FSA is partitioned into the relevant male and female portions and distributed within the respective robots (fish). However, the abstraction remains valid that a linear chain of behaviors transcending an individual fish is sequenced using perceptual triggers. In robotic systems, a separate process may implement the FSA, perhaps even executing on a computer physically remote from the robots, or it may be distributed similarly to the biological solution. In either case, the implementation choice does not impact the abstract description of the configuration.

3.1 The Atomic Agent

The specification of the components, connections, and structure of the control system for a group of robots will be called the **configuration**. A configuration consists of a collection of active components (agents), inter-agent communication links (channels), and a data-flow graph describing the structure of the configuration created from the agents and channels. Configurations can be either **free** or **bound** to a specific architecture and/or robot. The agent is the basic unit of computation in the configuration with agents asynchronously responding to stimuli (arrival of input values) by generating a response (transmission of an output value). There are two types of agents: atomic and assemblages. The atomic agents are parameterized instances of primitive behaviors while assemblages are coordinated societies of agents which function as a new cohesive agent. Agent assemblages are defined in Section 3.3 below.

The term *agent* has been overused in the literature but seems to most closely convey the essence of what is intended in this instance. Agent will be used to denote a distinct entity capable of exhibiting a behavioral response to stimulus. This definition is intentionally broad to allow application to a spectrum of objects ranging from simple feature-extracting perceptual modules, perceptual-motor behaviors, complex motor skill assemblages, individual robots, and coordinated societies of multiple robots.

Primitive behaviors are computable functions implemented in some convenient programming language, and serve as the configuration building blocks. An example of a primitive behavior is a *move-to-goal* function which, given the goal location, computes a desired movement vector to bring the robot closer to the goal. Figure 3.3 shows a schematic of a simple atomic agent parameterized with the configuration parameters $parm_1, parm_2, \ldots, parm_n$.

To construct a formal definition of primitive behaviors let f be a function of n variables, $(v_1, v_2, \ldots, v_n)$, computing a single output value, y. Define $V_1, V_2, \ldots, V_n$ as the set of input variables (either discrete or continuous). For f to be a suitable function for a primitive behavior it is required to be computable, meaning that it is defined on all n-tuples created from the Cartesian product $V_1 \times V_2 \times \ldots \times V_n$. Otherwise, there will exist input sets which cause f to generate indeterminate operation of the agent. Equation 3.1 formalizes this requirement of computable behaviors.

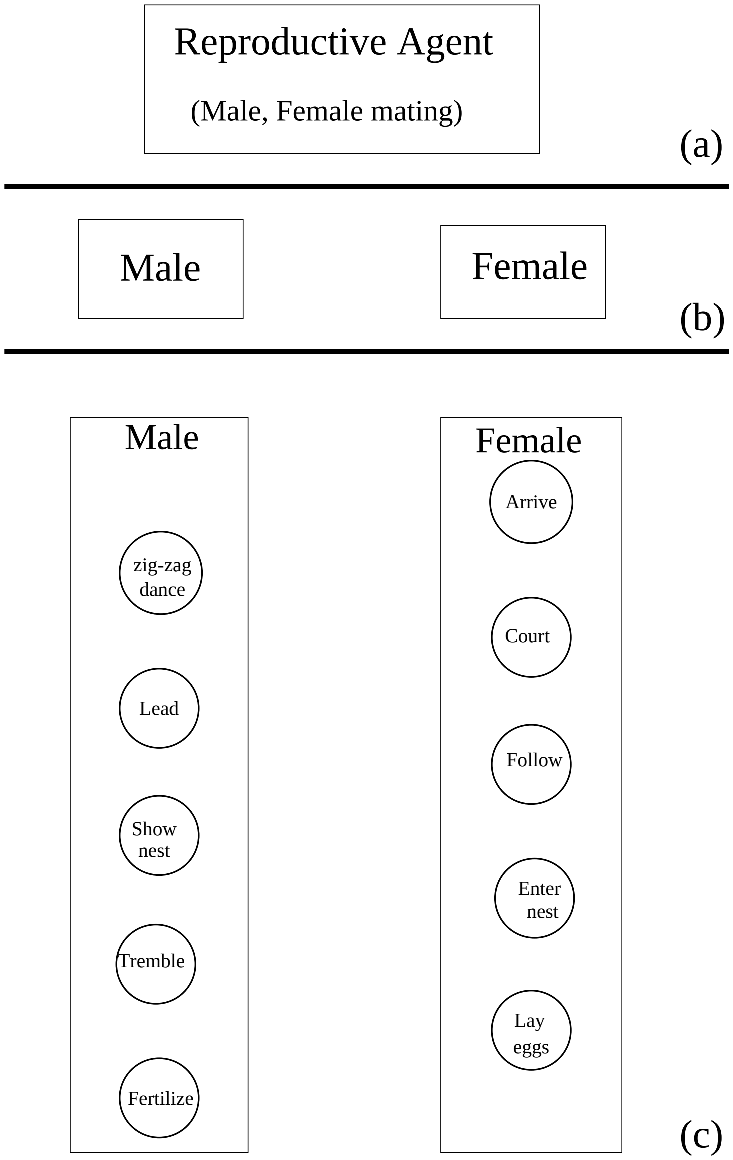


Figure 3.2: Schematic of three-spined stickleback mating behavior showing three levels of abstraction. Level *a* represents the mating behavior as a single agent, level *b* shows the two individual fish, and level *c* shows the various operating states required to create the mating behavior.

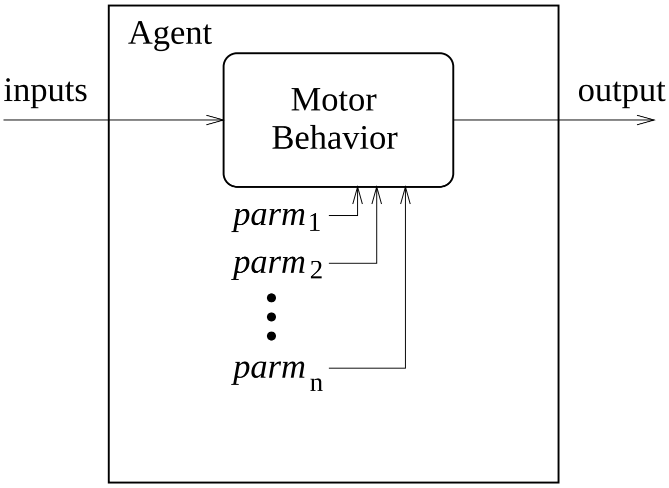


Figure 3.3: Schematic diagram of an atomic agent.

$$y = f(v_1, v_2, \ldots, v_m) \mid f \textbf{ is defined } \forall (v_1 \times v_2 \times \ldots \times v_m) \tag{3.1}$$

Equation 3.2 specifies that any entity capable of stimulus-response behavior can be treated as a distinct agent.

$$Agent \equiv Behavior(Stimulus) \tag{3.2}$$

This leads to the question of whether a computable function exhibits such behavior. In answer, one can easily view the inputs to the function as the stimulus and the computed output from this stimulus as the response.

Definition 1
$Agent \equiv f(v_1, v_2, \ldots, v_m) \mid \exists v_i \in [v_1, v_2, \ldots, v_m]$ *where* v_i *varies temporally*

Definition 1 specifies that a **situated** computable function is in fact an agent. The restriction that the function be situated requires that the inputs are not simple constants but, in fact, dynamic dataflows providing temporally varying stimuli over the lifetime of the agent in response to environmental changes. This definition expands the definition of an agent presented in "The Society of Mind"[58, pages 23,326] to encompass all situated computable functions. This is a controversial assertion in some circles, but clearly follows from the previous discussion.

3.2 Primitive Behavior Classes

To support the construction of atomic agents from primitive behaviors, a function definition will be provided for each module class. Primitive behaviors have been partitioned into four classes based on the actions they perform: *sensor*, *actuator*, *perceptual*, and *motor*.

3.2.1 Sensors

Sensors are hardware dependent and are not present in the free configuration. Instead, input *binding points* are used as place holders to mark where the sensor device drivers will be connected during the hardware binding process. Input binding points are represented as a source for the configuration dataflows. Formally, an input binding point is represented as generating a stream of sensations S where each sensation $s_1, s_2, \ldots \in S$ represents one sampling event. Sensations are generated at a rate dictated by the capabilities of the particular sensor to which it is bound and the demands of the perceptual modules using the sensation stream. Figure 3.4 shows a schematic diagram of an input binding point bound to a sensor.

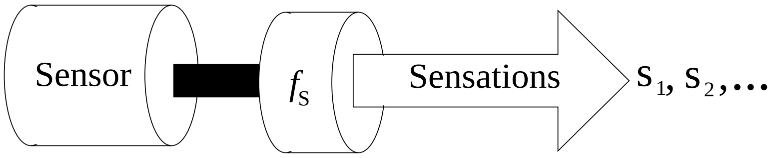


Figure 3.4: A schematic diagram of an input binding point. One is present for each physical sensor in the robot configuration.

The same input binding point can be represented using functional notation as the function $f_S\,(\;)$ which generates the sensation s_i at time t. Equation 3.3 presents this functional notation.

$$s_i = f_S(t) \tag{3.3}$$

3.2.2 Perceptual Modules

Perceptual modules function as virtual sensors[30, 29] which extract semantically meaningful features from one or more sensation streams and generate as output a stream of features (individual percepts). Viewing perceptual modules as virtual sensors facilitates hardware-independent perception and task-oriented perceptual processing relevant to the current needs of the configuration.

This generic view of a perceptual module also supports affordance-based[24, 23] implementations. In this case, the features extracted from the environment would each be affordances for the robot, allowing it to generate some response. The theory of affordances contends that perception extracts from the environment not geometric information, but information about what actions the environment affords the observer. For example, perception for a *sit_down* behavior would locate an object that has the affordance of *sittable* as a target object upon which to rest.

Formally, a perceptual module extracts features from one or more sensation streams $S^1, S^2, \ldots, S^n$ and generates a stream of features P where each $p_1, p_2, \ldots \in P$ represents one percept. Features are generated at a rate dictated by the capabilities of the sensors and the demands of the modules using the feature stream. Each p_i is made available (via broadcast, shared memory, etc.) to all modules using the perceptual module. Figure 3.5 shows a schematic diagram of a perceptual module consuming sensations to generate a feature stream.

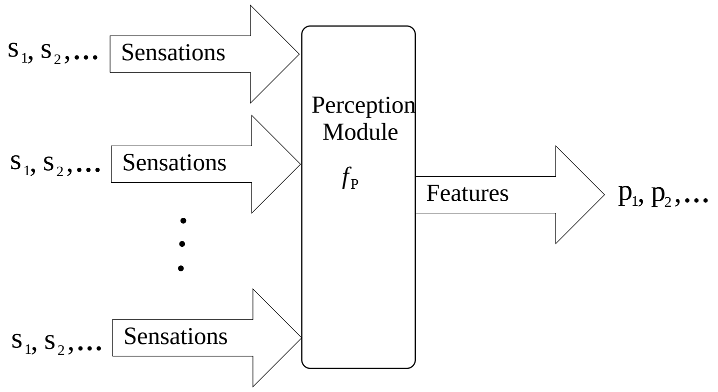


Figure 3.5: Schematic diagram of a perceptual module. There will exist many perceptual modules in a configuration, each tailored to the specific needs of one or more motor modules which consume the output feature stream.

The perceptual module can also be represented as a function f_P over the task-specific input sensation stream state vector $S_1, S_2, \ldots, S_n$ where each sensation stream

S_i is generated by a sensor module. Equation 3.4 denotes this functional representation.

$$p_i \;=\; f_P\left(s_1, s_2, \ldots, s_n\right) \tag{3.4}$$

3.2.3 Motor Modules

Motor modules consume one or more feature streams (perceptual inputs) to generate an action stream (a sequence of actions for the robot to perform). Formally, a motor module M uses information from one or more feature streams $P_1, P_2, \ldots, P_n$ to generate an action a_i at time t. Figure 3.6 shows a schematic diagram of a motor module consuming perceptual features to generate a stream of actions to be performed.

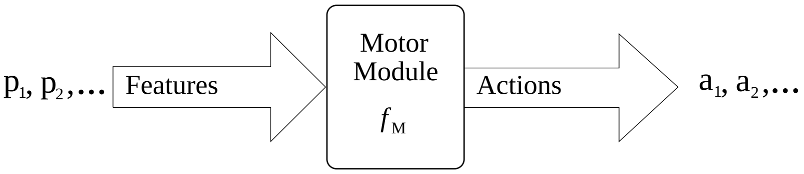


Figure 3.6: Schematic diagram of a motor module. There will exist many motor modules in a configuration, each closely coupled with a perceptual module. These pairs of perceptual and motor modules are called sensorimotor behaviors and form the basis for stimulus response actions within the configuration.

The motor module can also be represented as a function $f_M\left(\right)$ over the specific perceptual feature stream state vector $P_1, P_2, \ldots, P_n$ where each feature stream P_i is generated by a perceptual module. Equation 3.5 formalizes this view of the motor behaviors.

$$a_t \;=\; f_M(p_1, p_2, \ldots, p_n) \tag{3.5}$$

3.2.4 Actuators

Similar to sensors, actuators are not present in the free configuration. Instead, output binding points are used to mark where the actuator will be connected during binding. The output binding point is represented as a data-flow sink in the configuration. Formally, an output binding point is represented as a consumer of a stream of actions A where each $a_1, a_2, \ldots \in A$ represents one action to be performed. Figure 3.7 shows a schematic diagram of an output binding point bound to an actuator.

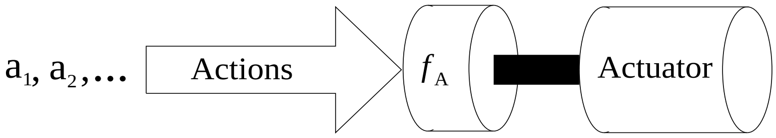


Figure 3.7: A schematic diagram of an output binding point. One is present for each actuator in the robot configuration.

An output binding point can also be represented as function f_A which at time t consumes the action a_i while attempting to make the changes to the environment specified in the action. Equation 3.6 presents this functional notation.

$$f_A\left(a_i\right) \Longrightarrow \text{Environmental Changes} \tag{3.6}$$

3.3 The Assemblage Agent

An assemblage is actually a coordinated society of agents which are treated as a new coherent agent. For example, an agent can be constructed from a society of other agents using a suitable coordination operator, C as follows:

$$Agent = C\left(Agent_1, Agent_2, \ldots, Agent_i\right)$$

When an assemblage agent is constructed, *subordination* occurs with one or more agents placed subordinate to the coordination operator. The construction creates a new assemblage agent which encapsulates the subordinates, thereby concealing them from other agents and forcing all interactions with the subordinates to be initiated via the coordination operator. Figure 3.8 shows a schematic diagram for a simple configuration. Each box represents an agent, with nested boxes denoting agents subordinate to the surrounding agent. In the example, the agents are labeled with either A_i for atomic and assemblage agents and C_j for coordination operators.

The assemblage construction is denoted functionally. For example, in Figure 3.8, the case of A_5 created by making A_4 subordinate to the coordinator C_2 is denoted $C_2\left(A_4\right)$. Equation 3.7 provides a complete expansion of the construction of Figure 3.8.

$$C_5\left(C_2\left(C_1\left(A_1, A_2\right)\right), C_4\left(C_3\left(A_6, A_7\right), A_8\right)\right) \tag{3.7}$$

3.4 Classes of Coordination Modules

A coordination module modifies the activities of the group of agents it is managing, exploiting the strengths of each to execute a specific task. This intervention may

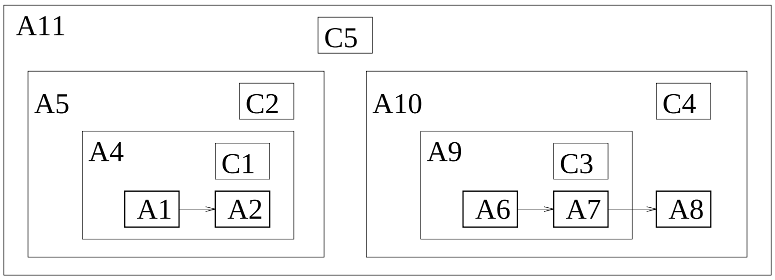


Figure 3.8: Schematic diagram of a configuration.

range from none (the null case) to explicit control of every action of the members. Figure 3.9 shows a taxonomy of the coordination mechanisms presented in this section. Notice that coordination is partitioned at the top level into state-based and continuous classes.

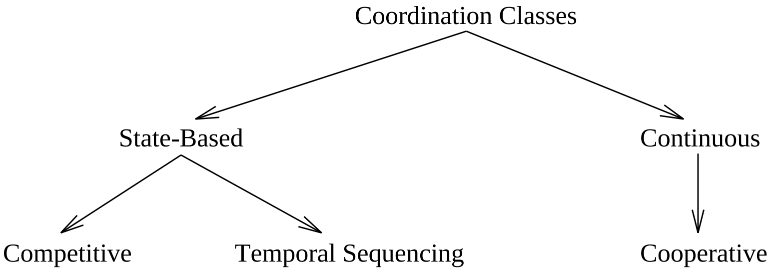


Figure 3.9: Classes of Coordination Modules.

State-based coordination mechanisms partition the agents they are managing into distinct groups, allowing only a single group representing a subset of the total agents to be active at any one time. This behavior allows the operating system to suspend execution and perhaps de-instantiate all but the active group of agents to conserve resources. Temporally sequenced coordination is the prime example of state-based mechanisms.

Continuous coordination mechanisms utilize results from all the agents they are managing to generate the desired output. This behavior requires that all the agents remain instantiated and executing. Cooperative coordination, which merges the outputs from each individual agent into a single value, is perhaps the best example of continuous coordination.

Each of the leaf nodes in Figure 3.9 represent distinct types of coordination and will be explained below.

3.4.1 Competition

The competition style of coordination selects a distinguished subset of the society to activate based on some metric. The process of determining this collection of active members (arbitration) can use a variety of techniques including spreading activation, assigning fixed priorities, or using relevancy metrics. Architectures using competition mechanisms include spreading activation nets[54], and the subsumption architecture[9].

The colony architecture[14] is a variant of the subsumption architecture which tends to simplify the behavioral networks (Section 2.2.2). As in the subsumption architecture, individual modules are interconnected into a hierarchy using a fixed priority arbitration network. Figure 3.10 shows a simple colony architecture network with three behaviors and two suppression nodes (labeled **S**). The design is that if behavior 3 has something to contribute, then it overwrites any outputs generated by behaviors 1 and 2. Otherwise, behavior 2 is given a chance to control the robot if it determines it is applicable, and finally, behavior 1 will output commands as a default behavior.

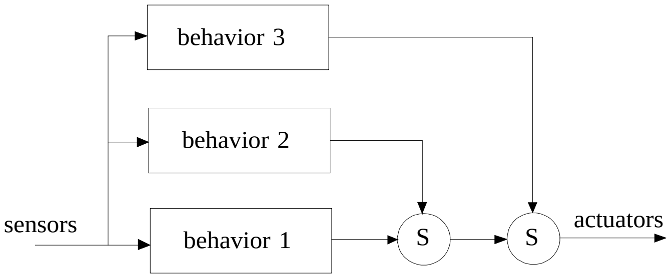
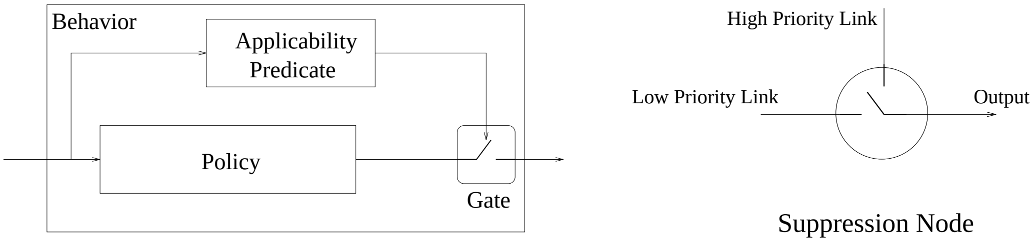


Figure 3.10: Example Colony Architecture Network.

Figure 3.11 shows a schematic of a behavior and suppression node in the colony architecture. Each behavior consists of a *policy* which generates control commands, and an *applicability predicate* which enables the output of the policy when it is perceived relevant. The suppression node passes on the high priority input whenever it is valid and falls back to passing the low priority input otherwise.



Suppression Node

Figure 3.11: Example Colony Architecture Behavior and Suppression Node.

Consider how the simple colony architecture network shown in Figure 3.10 would be represented in the **Societal Agent** architecture. First, define the functions computing the three behavior policies as *policy1*, *policy2*, and *policy3* which transform input values to outputs. Next, define the three boolean applicability predicates as the functions *valid1*, *valid2*, and *valid3* which determine if the corresponding policies are relevant and likely to generate useful values or not. These six functions would need to be implemented by the user to actually compute the expected algorithms; in this construction they are simply referenced by name. Equation 3.8 defines a suitable suppression function, *suppress*, for use in implementing the network, where *hi* is the value of the function when the boolean flag *hi_valid* signals that the high priority input is valid and *low* is the default value of the function.

$$\mathit{suppress}(\mathit{hi}, \mathit{hi\_valid}, \mathit{low}) = \begin{cases} \mathit{hi} & \text{if } \mathit{hi\_valid} \\ \mathit{low} & \text{otherwise} \end{cases} \tag{3.8}$$

Using Equation 3.8 twice allows specification of Figure 3.10 functionally as shown in Equation 3.9.

$$\mathit{suppress}(\mathit{policy3}, \mathit{valid3}, \mathit{suppress}(\mathit{policy2}, \mathit{valid2}, \mathit{policy1})) \tag{3.9}$$

Notice that, based on the definition of *suppress* in Equation 3.8, *policy3* correctly dominates when it is valid, otherwise *policy2* dominates *policy1* when it has valid data and *policy1* is only allowed to generate an output when both of the other behaviors are not generating useful data.

3.4.2 Temporal Sequencing

Temporal sequencing is a state-based coordination mechanism which uses a Finite State Automaton (FSA)[34, 1] to select one of several possible operating states based on the current state, the transition function, and perceptual triggers. Each state in the FSA denotes a particular member agent which is dominant when that state is active. This type of coordination allows the group to use the most relevant members based on current processing needs and environmental conditions.

Equation 3.10 provides a formal definition of temporal sequencing using the coordination function f_{seq}. This function uses the FSA α containing the set of perceptual triggers along with the set of agents $[a_1, a_2, \ldots, a_m]$ to select the specific agent to activate based on the current state in the FSA.

$$f_{seq}(a_1, a_2, \ldots, a_m, \alpha) = a_i \mid \text{state } i \text{ is active in } \alpha \tag{3.10}$$

Without loss of generality, assume that there is a one-to-one mapping of states in the FSA to the list of members $[a_1, a_2, \ldots, a_m]$, with agent a_i active when the FSA is operating in state i. The FSA α is specified by the quadruple[34] (Q, δ, q_0, F) with

- Q the set of states, $\{q_0, q_1, \ldots, q_m\}$ where each q_i is mapped to a_i.
- δ the transition function mapping the current state (q_i) to the next state q_{i+1} using inputs from the perceptual triggers is generally represented in tabular form.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accepting states which signal completion of the sensorimotor task.

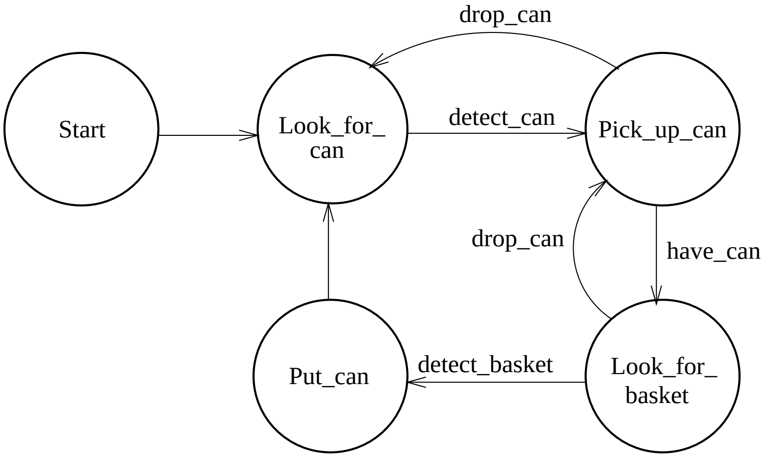


Figure 3.12: An FSA for a trash collecting robot.

Consider specification of a configuration implementing a janitorial task for a team of robots (*e.g.*, 1994 AAAI mobile robot competition[5]). Specifically, each robot should wander around looking for empty soda cans, pick them up, wander around looking for a recycling basket, and then place the can into the basket. Figure 3.12 is a graphical representation of an FSA for such a robotic trash collector. The circles represent the possible operating states with the label indicating the assemblage agent active during that state. The arcs are labeled with the perceptual triggers causing the transitions, where relevant. Unlabeled transitions leave the state immediately after the assemblage executes the first time. There are no accepting states since during the competition the robots ran until they were manually turned off.

The FSA in Figure 3.12 would be represented by the quadruple

$$(\{Start, Look\_for\_can, Pick\_up\_can, Look\_for\_basket, Put\_can\}, \delta, Start, \emptyset)$$

Table 3.1: Tabular representation of δ function for Figure 3.12 FSA.

State	*normal*	*terminal*	*error*
Start	Look_for_can	Look_for_can	$\emptyset$
Look_for_can	Look_for_can	Pick_up_can	$\emptyset$
Pick_up_can	Pick_up_can	Look_for_basket	Look_for_can
Look_for_basket	Look_for_basket	Put_can	Pick_up_can
Put_can	Put_can	Look_for_can	$\emptyset$

The transition function δ for the trash collecting FSA is specified in Table 3.1. Powering up in the *start* state, the robot begins to wander, looking for a suitable soda can, operating in the *Look_for_can* state. When a can is perceived, the *Pick_up_can* state is activated and, if the can is successfully acquired, a transition to the *Look_for_basket* state occurs. Loss of the can in either of these states causes the FSA to fall back to the previous state and attempt recovery. When a recycling basket is located, the *Put_can* state becomes active and the can is placed in the basket. A transition back to the *Look_for_can* state repeats the process.

3.4.3 Cooperation

The cooperative class of coordination manages the actions of members of the society to present the appearance and utility of a single coherent agent. The vector summation in the AuRA[3, 2] architecture is such a mechanism. The AuRA gain-based cooperative cooperation operator can be represented functionally as a weighted vector summation, as shown in Equation 3.11. In this case, the coordination function f scales each of the input vectors v_i by its corresponding weight (gain) w_i before computing the vector sum of these scaled inputs as the output for the group.

$$f\left(\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n, w_1, w_2, \ldots, w_n\right) = \overset{\rightarrow}{\sum_{k=1,n}} \left(\vec{v}_k * w_k\right) \tag{3.11}$$

Figure 3.13 shows a schematic example of gain-based cooperation in AuRA. All of the behaviors *Avoid_obstacles*, *Move_to_goal*, *Noise*, and *Probe* are active and generate a two dimensional vector denoting the direction and speed they would like the robot to move. This representation allows a simple vector summation process to compute a composite vector which represents the group's behavioral consensus.

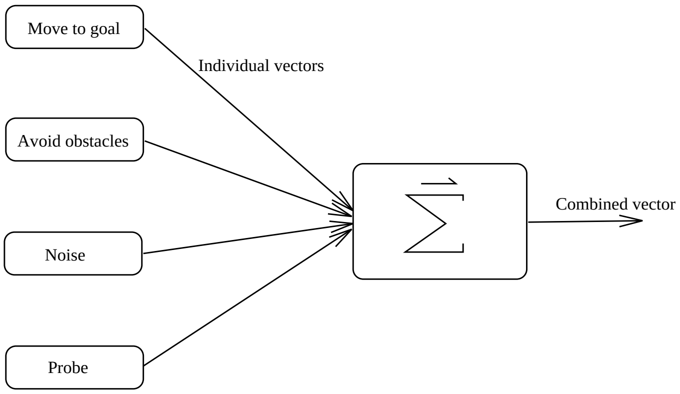


Figure 3.13: Schematic diagram of vector summation in AuRA.

3.5 Overview of Binding

Binding is covered in depth in Sections 4.2 and 5.2. However, a brief overview here will provide a better perspective on why physical entities (robots) were not mentioned in the preceding discussions.

The design process specifies a **free** configuration using abstract data types and behavior specifications. Instances of this configuration may then be bound to a particular architecture and specific robot hardware. Using these bound configurations, executables for the chosen target architecture and hardware may then be generated automatically.

Architectural binding selects a particular implementation library and code generator. This binding replaces the abstract data types with architecture-specific implementations. Additional configuration parameters may be added to behaviors to control architecture-specific features.

Hardware binding attaches sensor and actuator device drivers to the input and output binding points used in the free configuration. These devices will add configuration parameters to control device-specific features.

It is important that the notion of an agent in the free configuration remain devoid of hardware limitations so it can be mapped to robots with disparate functionality. For example, one particular binding of a configuration may attach the perceptual agents to separate robots to implement distributed perception where another uses a single, complex robot. If the free configuration contains any explicit partitioning into physical objects flexibility has been lost and there exist societies to which the configuration can no longer be mapped.

3.6 Summary

The "Society of Mind"[58] develops a particularly appealing behavior-based model of intelligence where the overt behavior of the system emerges from the complex interactions of a multitude of simple agents. This model fits naturally with the work in behavior-based robotics where the controller is clearly separable from the vehicle. This representation shows that societies of robot vehicles should simply comprise a new level in the hierarchical description of the societies of agents comprising each robot.

The **Societal Agent** representation has been developed which formalizes this viewpoint. Two types of agents are defined: instantiations of primitive behaviors, and coordinated assemblages of other agents. This recursive construction captures the specification of configurations ranging in complexity from simple motor behaviors to complex interacting societies of autonomous robots. Coordination processes which serve to group agents into societies are partitioned into state-based and continuous classes. State-based coordination implies that only the agents which are members of the active state are actually instantiated. Continuous coordination mechanisms attempt to merge the outputs from all agents into some meaningful policy for the group.

The power of this architecture is the uniformity of its recursive descriptions of complex systems and its hardware and architecture independence. Configurations are constructed and manipulated without commitment to hardware, architecture, or specific implementations of behaviors. A binding step creates instances of free configurations bound to particular architectures and robots. Making the binding step explicit facilitates retargeting and reuse of configuration components.

Chapter 4

The Configuration Description Language

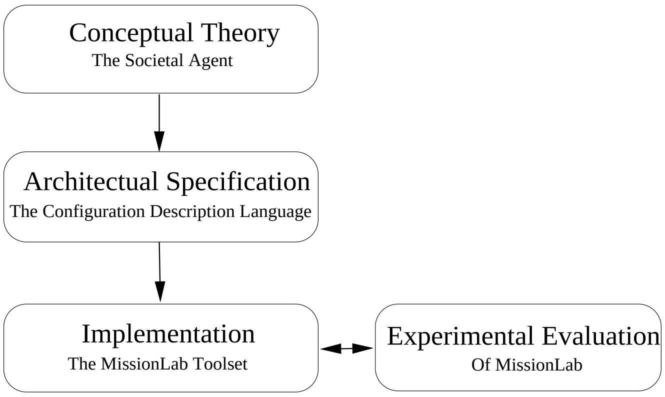


Figure 4.1: The research components.

As shown in Figure 4.1, the next step after developing the **Societal Agent** theory (Chapter 3) is to mold it into a concrete specification presented in the form of a language. The Configuration Description Language (CDL) captures the critical uniform representation of recursively defined agents developed in the **Societal Agent** theory. CDL supports the specification of architecturally-independent configurations which are not couched in the terms of a particular robot architecture. It is an agent-based language which encourages the construction of new agents from coordinated collections of existing agents. These new agents are treated as atomic objects with the same stature as all other agents available to the system designer. The recursively constructed configurations created using this process faithfully follow the **Societal Agent** theory.

To support the construction of generic configurations and thereby the ability to target disparate robot run-time architectures, hardware bindings are separately and explicitly represented only when a CDL configuration is deployed on particular devices. Raising generic configurations above run-time constraints ensures maximum code reuse opportunities by minimizing machine dependencies.

Mechanisms for implementing the individual software primitives which ground the recursive constructions are architecture-dependent and reside below the scope of a CDL representation. CDL's strict partitioning of the implementation of primitives from the configuration of those primitives allows the mission specification task to be decoupled from the implementation of primitive behaviors. For our purposes it is sufficient to be assured that a suitable collection of primitives is available and that each supported robot run-time architecture can utilize some subset of this collection. CDL supports mechanisms for describing the interfaces to such primitives so they are available for use. The task for the configuration designer is to take these building blocks and describe how they are to be combined and deployed to perform a particular mission. The use of explicit hardware bindings simplifies the task of retargeting an existing configuration to a new vehicle.

4.1 Overview of CDL

CDL is used to specify the instantiation and coordination of primitives and not their implementation. Therefore, each of the primitives must have a CDL definition which specifies its programming interface. For example, a primitive which adds two numbers and returns their result might have a CDL definition such as

$$\text{defPrimitive integer Add (integer A, integer B);}$$

This defines the primitive `Add` which takes two `integer` inputs `A` and `B` and outputs an `integer`.

An agent can be instantiated from the `Add` primitive just by referencing it, as in

$$\mathrm{Add}\,(\mathrm{A} = \{3\}, \mathrm{B} = \{5\})\,;$$

This statement creates an instance of the `Add` primitive and assigns the constant initializer 3 to the input `A` and 5 to the input `B`. Although the implementation of `Add` is not specified, we expect that the output of the agent will be 8. Notice from this example that parameters are specified by name and passed by value in CDL. These features support tight syntax checking and eliminate side effects. All constant initializer strings are surrounded in { } brackets to simplify parsing.

The previous statement created an anonymous (unnamed) agent. CDL uses a functional notation to specify recursive construction of objects and the only mechanism for extracting the output value from such an agent is to embed the invocation on the right-hand side of an assignment statement using standard functional notation. For example, this statement creates two nested anonymous agents where the input parameter `A` for the outermost one gets the output value 8 from the innermost one and then adds 1 to it.

$$\mathrm{Add}\,(\mathrm{A} = \mathrm{Add}\,(\mathrm{A} = \{3\}, \mathrm{B} = \{5\})\,, \mathrm{B} = \{1\})\,; \tag{4.1}$$

Although powerful, this nesting can become cumbersome when carried to great depths. It also prevents the output value from an agent to be shared by multiple consumers. However, if an agent is given a name the output value can be referenced using that name. This partitions the specification of the agent from its usage and allows the output value to be used in multiple places. Creating a named agent is accomplished using the `instAgent` keyword.

$$\mathrm{instAgent\ myAgent\ from\ Add}\,(\mathrm{A} = \{3\}, \mathrm{B} = \{5\})\,;$$

Now other agents can reference the output of `myAgent` by name.

$$\mathrm{Add}\,(\mathrm{A} = \mathrm{myAgent}, \mathrm{B} = \{1\})\,;$$

is equivalent to the earlier nested agents declaration. Notice the uniformity with usage of the in-line anonymous agents.

It is important to be aware that each agent instantiated from a particular primitive is a unique entity, disjoint from all other instantiations of the primitive. When data values must be distributed to multiple consumers the named agent mechanism must be used to ensure that the same process is providing the data to all consumers.

An important feature of CDL is the support for recursive construction of assemblages. An assemblage is a coordinated society of agents which can be treated exactly the same as a primitive behavior. A common situation is for a designer to spend time building and debugging a configuration which completes a single high-level task such as traveling down a hallway or chasing a moving target. Once completed, this configuration should be archived to a library as a new high-level assemblage for later reuse. CDL provides a simple mechanism for converting a complex agent instantiation to an assemblage which can later be instantiated.

In CDL assemblages are created using the `defAgent` keyword. Consider, for example, Statement 4.1 which demonstrated the nesting process. We can turn that agent into an assemblage as follows:

$$\mathrm{defAgent\ Add8\ from\ Add(A} = \mathrm{Add}\,(\mathrm{A} = \{3\}, \mathrm{B} = \{5\})\,, \mathrm{B} = \{\hat{}\ \mathrm{Val}\});$$

Notice the use of the {`^ Val`} deferral operator to push up a parameter to the level of the new assemblage definition. This mechanism allows the designer to provide values for those parameters which are truly internal to the new construction while making relevant parameters visible to the user. In this case the value of input `B` is deferred and also renamed to `Val`. This creates an assemblage which has an interface equivalent to the following primitive definition. Agents can be instantiated from this assemblage in exactly the same manner as from a true primitive.

defPrimitive integer Add8 (integer Val) ;

When an agent is instantiated from the assemblage the value assigned to `Val` will replace the deferral operator, and is the value assigned to input `B`.

This completes our cursory overview of the usage of CDL. There are many syntactic constructions related to defining the operators, binding points, and data types which have yet to be explained. Some of these will be presented in the next section during development of the example configuration and the remainder when the full language is presented.

4.1.1 Example Janitor Configuration

The use of CDL is further demonstrated here by constructing an example robot configuration for the *cleanup the office* (or janitor) task using three robots. This task is similar to the 1994 AAAI mobile robot competition[5] where the robots retrieved soda cans and placed them near wastebaskets.

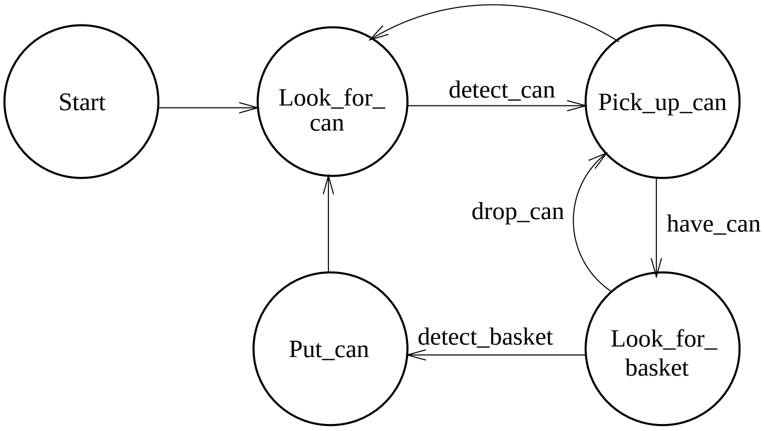


Figure 4.2: FSA for a trash collecting robot (`cleanup` agent).

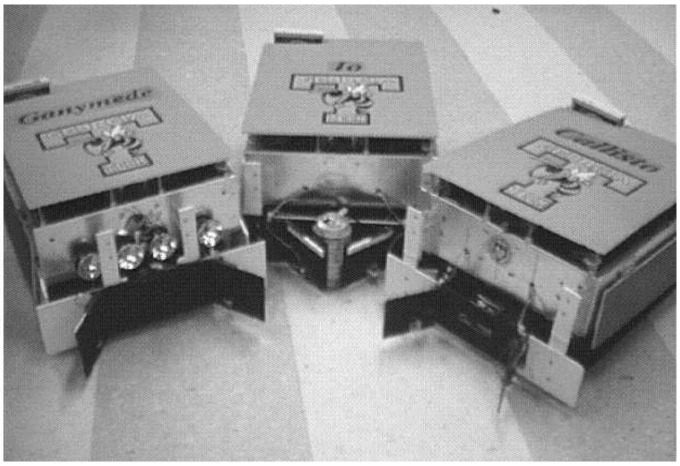


Figure 4.3: Three trash collecting robots from AAAI94[5].

Reconsider the trash collecting state-transition diagram Figure 3.12 from Section 3.4.2 reproduced in Figure 4.2. Let's call this the `cleanup` agent. During the actual AAAI competition a similar cleanup agent was deployed on each of the three vehicles shown in Figure 4.3 to retrieve soda cans and place them in wastebaskets[1]. We will use this `cleanup` agent to construct a `janitor` configuration similar to the one used in the AAAI competition.

The CDL description for the top level of the generic `janitor` configuration shown in Figure 4.4 represents the three cleanup robots as a single janitor entity. We will now examine each statement of the `janitor` configuration.

Statement 1 defines a prototype `cleanup` behavior. The prototype creates a placeholder which allows building a particular level in the configuration before moving down to define the implementation of the `cleanup` behavior as either an assemblage or a primitive. This is an important feature when building configurations in a top-down manner. The `defProto` keyword is not yet supported in *MissionLab* and only a single built-in prototype behavior is available. It is used in the example to demonstrate its utility. Conversion of *MissionLab* to support the `defProto` syntax will expand the ability of designers to work in a top-down fashion using the toolset.

The prototype `cleanup` agent in Statement 1 generates an output of type `movement`. The `movement` data type is used to send motion commands to control motion of the robot and contains the desired change in heading and speed.

Statements 2, 3 and 4 instantiate three agents based on the `cleanup` behavior. Since this configuration is being constructed from the top down and it is known *a*

---

[1] Due to hardware limitations the robots in Figure 4.3 only placed the cans near the wastebaskets.

```
   /* Define cleanup behavior as a prototype */
1. defProto movement cleanup();

   /* Instantiate three cleanup agents */
2. instAgent Io from cleanup();
3. instAgent Ganymede from cleanup();
4. instAgent Callisto from cleanup();

   /* Create an uncoordinated janitor society */
5. instAgent janitor from IndependentSociety(
      Agent[A]=Io,
      Agent[B]=Ganymede,
      Agent[C]=Callisto);

   /* janitor agent is basis of configuration */
6. janitor;
```

Figure 4.4: Partial CDL description of multiagent `janitor` configuration. Note that comments are bounded by /* */ and that line numbers were added to allow reference to particular statements and are not part of CDL.

priori that it will control three robots, an early commitment to a three agent society is made in these statements.

Statement 5 creates a society of three of the cleanup agents and gives it the name `janitor`. It also introduces new notation which requires explanation. CDL partitions the primitive behaviors from the operators used to coordinate them. This helps to keep both the behaviors and operators independent and understandable. In Statement 5, IndependentSociety is a coordination operator which can be defined as follows:

defOperator movement IndependentSociety CONTINUOUSstyle(list integer Agent);

This defines the `IndependentSociety` operator as coordinating a list of agents. The `CONTINUOUSstyle` keyword means that the operator is not state-based and that the output will be a function of the instantaneous inputs. This provides information to the CDL compiler allowing it to generate more efficient code. The `list` keyword defines the input parameter as a list of `Agent` entries. Assignments to lists use the [ ] brackets to denote the index, in this case A, B, and C are used for the indices. These only matter when the list consists of two or more inputs which must be kept in correspondence. The `IndependentSociety` operator is implemented to have no coordinative effect on the individual robots in Figure 4.3, allowing them to operate independently.

Statement 6 specifies that the `janitor` society is the top level in the configuration. This extra step is necessary since some or all of the preceding statements could be placed in libraries and this reference would cause their linkage. This completes the high-level design of the `janitor` configuration.

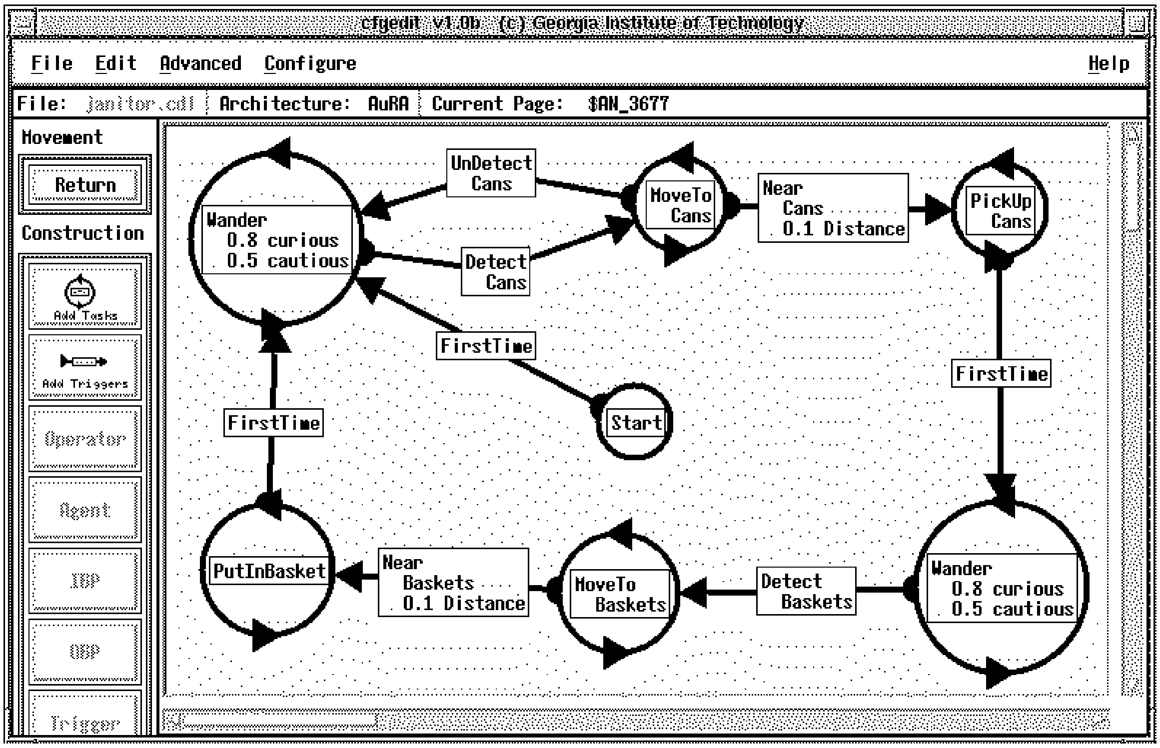

File: janitor.cdl | Architecture: AuRA | Current Page: $AN_3677

Figure 4.5: Screen snapshot of the `cleanup` FSA in the Configuration Editor. See Section 5.1.2 for a description of the editor.

The next step in the development cycle is to define an implementation for the `cleanup` prototype specified in Statement 1 of Figure 4.4. Because the cleaning task is state-based, it was decided to implement `cleanup` using an FSA and a state-transition diagram of the solution was presented in Figure 3.12. To implement the `cleanup` behavior, this diagram was entered graphically using the *MissionLab* configuration editor (which will be presented in Chapter 5). A screen snapshot of the FSA in the editor is shown in Figure 4.5. The configuration editor will be described in Section 5.1.2.

```
      /* Define the FSA coordination operator */
1.    defOperator movement FSA STATEstyle(
                    list movement states, list boolean triggers);

      /* Instantiate the cleanup agent from the FSA operator */
2.    instAgent cleanup from FSA(
           /* Define which agent will be active in each state */
2a.        states[Start] = Idle(),
2b.        states[LookForCan] = Explore(),
2c.        states[MoveToCan] = MoveTo(objects = {^}, class = Can),
2d.        states[PickUpCan] = PickUp(objects = {^}, class = Can),
2e.        states[LookForBasket] = Explore(objects = {^}),
2f.        states[MoveToBasket] = MoveTo(objects = {^}, class = Basket),
2g.        states[PutCan] = PutCan(),

           /* Define the transitions out of each state */
2h.        triggers[Start] = if [true] Trigger LookForCan,
2i.        triggers[LookForCan] = if [DetectObject(objects = {^}, class = Can)]
                    Trigger MoveToCan,
2j.        triggers[MoveToCan] = if [Near(objects = {^}, class = Can)] Trigger PickUpCan,
2k.        triggers[MoveToCan] = if [Not(A=DetectObject(objects = {^}, class = Can))]
                    Trigger LookForCan,
2l.        triggers[PickUpCan] = if [HaveCan()] Trigger LookForBasket,
2m.        triggers[PickUpCan] = if [Not(A=HaveCan())] Trigger MoveToCan,
2n.        triggers[LookForBasket] = if [DetectObject(objects = {^}, class = Basket)]
                    Trigger MoveToBasket,
2o.        triggers[LookForBasket] = if [Not(A=HaveCan())] Trigger PickUpCan,
2p.        triggers[MoveToBasket] = if [Near(objects = {^}, class = Basket)] Trigger PutCan,
2q.        triggers[MoveToBasket] = if [Not(A=HaveCan())] Trigger PickUpCan,
2r.        triggers[PutCan] = if [true] Trigger LookForCan,

           /* push up the objects parameter to our parent */
2s.        objects = {^} );
```

Figure 4.6: CDL description of `cleanup` agent.

The FSA code generated by the editor (reformatted for clarity) is shown in Figure 4.6. The FSA is specified with agents attached to each operating state (`states[ ]`) and each perceptual trigger (`triggers[ ]`). At run-time, the output of the active agent (attached to the active state) is passed through as the output of the FSA. Then the transition function manifested in the perceptual triggers uses perceptual inputs and the current state to determine the new (possibly the same) state.

Statement 1 defines the operator `FSA` which is the basis of the temporal sequencing coordination. The built-in styles of coordination operators are `STATEstyle` and `CONTINUOUSstyle`. `STATEstyle` operators maintain internal state and use that past history along with the current inputs to compute the output value. An example is spreading activation, where the activation level of behaviors is constant from instant to instant and only changes in response to excitation and inhibition links, and global rates of decay. The `CONTINUOUSstyle` operators do not maintain any state information so the output is a function only of the instantaneous inputs. An example is a function which simply passes the "strongest" of its inputs through as the output. The choice as to which is "strongest" is made each time without any reliance on what occurred in the past.

Statement 2 declares the cleanup agent as an instance of the `FSA` operator. Compare Figure 4.6's statements with the graphical representation in Figure 4.2. The states (circles) are represented as a list of assignments (Statements 2a-2g). The right-hand side of these statements is the agent which will provide the output for the `FSA` when that particular state is active. These agents are unrestricted in stature and can contain their own FSA's or other agents in the normal recursive manner.

Statements 2h-2r denote the transitions from state to state. The left-hand side lists the state that the transition leaves from and the new state is listed after the `Trigger` keyword. The agent within the square brackets on the right-hand side is a perceptual trigger which evaluates to a boolean `true` or `false`. The FSA remains in the the current state until an output transition originating from the active state becomes true. When this occurs the state mentioned in the transition becomes the new active state.

Figure 4.7 provides a definition of `LookForCan` as a representative example of the motor agents implementing the states in the `cleanup` FSA. The `WeightedCombination` coordination operator is defined in Statement 1. This operator computes a weighted combination of its inputs as the output for the society.

Statement 2 defines the `LookForCan` agent as the coordinated combination of the `Wander`, `Probe`, `AvoidObstacles`, and `AvoidRobots` agents. The `objects` input parameter has been deferred and will be determined at the FSA level. The `WeightedCombination` coordination operator uses the list of matching weights in the combination process to control the relative contributions of the three agents to the group's output.

The `AvoidObstacles` agent is shown in Figure 4.8. Statement 1 defines a new class of input binding points and gives them the name `sense_objects`. The input

```
   /*Define weighted combination coordination operator*/
1. defOperator movement WeightedCombination
      CONTINUOUSstyle(list movement inputs,
         list float weights);

   /* Create explore agent from coordination operator */
2. instAgent LookForCan from WeightedCombination(
      /* Define the agents active when explore is active */
2a.   inputs[A] = Wander(persistence = {10}),
2b.   inputs[B] = Probe(objects = {^}),
2c.   inputs[C] = AvoidObstacles(objects = {^}),
2d.   inputs[D] = AvoidRobots(objects = {^}),

      /* Define each agent's contribution */
2e.   weights[A] = {0.5},
2f.   weights[B] = {1.0},
2g.   weights[C] = {1.0},
2h.   weights[D] = {0.8},

      /* Push up specification of parameter to parent */
2i.   objects = {^});
```

Figure 4.7: Partial CDL description of `LookForCan` agent.

```
   /* Define a new class of input binding points */
1. defIBP ObjectList sense_objects(
      number max_sensor_range);

   /* Create the AvoidRobots agent */
2. instAgent AvoidRobots from AvoidObjects(
2a.   horizon = {2.0},
2b.   safety_margin = {0.5},

      /* Defer specification of the objects parameter */
2c.   objlist = FilterObjectsByColor(
         color = {Green}, objects = {^}),

      /* Push up objects parameter to our parent */
2d.   objects = {^});
```

Figure 4.8: Partial CDL description of `AvoidRobots` agent.

binding points serve as connection points for input sensor device drivers when configurations are bound to specific robots. The definition declares that sensors of this type generate streams of `ObjectList` readings and require a configuration parameter `max_sensor_range` denoting the distance beyond which sensor readings are ignored. Note the uniform representation of input binding points and the other primitives. CDL attempts to keep the syntax similar for all objects.

Statement 2 creates the `AvoidRobots` agent as an instance of the `AvoidObjects` primitive. This primitive motor module uses `horizon` and `safety_margin` parameters to determine the strength of its reaction to objects. Statement 2c specifies the perceptual filter `FilterObjectsByColor` will construct the list of robot objects for the `AvoidRobots` behavior by removing those objects from its input list whose color doesn't match the specified value. In this example, the robots are green.

`AvoidObjects` is a primitive and CDL does not include a facility for directly specifying the implementation of primitive behaviors. Instead, for each supported robot run-time architecture in which a particular primitive is to be available, an agent prototype definition describing the interface to the module is used to make the primitive available to the designer.

The CDL syntax has been overviewed and an example configuration developed in detail. The uniform treatment of objects in CDL provides a clean syntax for the user. The recursive support for the construction of assemblages allows building high-level primitives and archiving them for later reuse.

An example `AvoidObjects` implementation for the AuRA architecture is shown in Figure 4.9 for completeness. Discussion of how primitives are implemented will be deferred until Chapter 5.

4.2 Binding

One of the strengths of CDL is its support for retargeting configurations through the use of generic configurations and explicit hardware binding. The binding process maps an abstract configuration onto a specific collection of robots; linking the executable procedures and attaching the binding points to physical hardware devices. At this point the user commits to specific hardware devices. The hardware binding process must ensure that required sensing and actuator capabilities are available, with user interaction guiding selection when multiple choices are available. The first step during binding is to define which portions of the configuration will be resident on each of the target robots. This partitioning can occur either bottom up or top down.

Working from the bottom up, the input and output binding points can be matched with the capabilities of the pool of available robots to create a minimal mapping. For example, a surveillance configuration might specify use of both vision and sound detectors. Such a configuration might be deployed on one robot which has both

```
procedure Vector AvoidObjects with
   double horizon;
   double safety_margin;
   ObjectList   objlist;
header
body
   /* Loop for each input object */
   for(int i=0; i<objlist.count; i++)
   {
      /* If close enough that we should react */
      if (dist <= safety_margin + horizon )
      {
         /* Get a vector from the robot to the object */
         Vector contribution = objlist.objects[i].location;

         /* compute the distance to this object */
         double dist = length_of_vector( contribution );

         /* React strongly if closer than our minimum safety margin */
         if (dist < safety_margin )
         {
            /* Construct a very large vector away from this object */
            contribution = set_vector_magnitude(contribution, INFINITY);
         }
         else /* Normal case: A linear drop in repulsion over distance */
         {
            /* Magnitude is 0 at edge of horizon and 1 at safety margin */
            double magnitude = (horizon - (dist-safety_margin)) / horizon;

            contribution = set_vector_magnitude(contribution, magnitude);
         }

         /* Add it to the running sum */
         output = vector_sum(output, contribution);
      }
   }

   /* Thread automatically sends the variable "output" to consumers */
pend
```

Figure 4.9: Example AuRA implementation of `AvoidObjects` primitive.

sensors available, or two robots, each with a single sensor. A second use of the list of required sensor and actuator capabilities is to use it as a design specification for the robotic hardware. In this scenario, the configuration is constructed based on the mission requirements. The actual hardware is later tailored to the requirements originating from this design.

An alternate method of completing the binding process is to work from the top down. In this case, the configuration may be partitioned along the lines of the behavioral capabilities required on each vehicle or based on the desired number of vehicles. For example, mission requirements may specify four scouting robots and one support robot. These requirements may be driven by desired coverage, protocol, redundancy, and budget constraints.

Binding a portion of a configuration to a specific robot will also bind that portion to a specific architecture since robots are modeled as supporting a single architecture. If a particular robot happens to support multiple architectures, multiple robot definitions can be created with different names, one for each architecture. Therefore, we can restrict a single robot definition to supporting a single run-time architecture with no loss of generality. During binding to a particular architecture, the system must verify that all components and coordination techniques used within the configuration are realizable within the target architecture since certain behaviors may not have been coded for that architecture and some coordination operators can be architecture specific.

Figure 4.10 shows the relevant CDL code for the `janitor` after it has been bound to the three robots shown in Figure 4.3. Statement 1 defines a class of robots called `blizzard`. This definition also specifies the set of sensors and actuators available on robots of this class. The actuator driving the vehicle is called `wheelActuator` and has a data type of `movement`. The only sensor on the robots, `objectSensor`, returns a list of perceived objects.

Statements 2-4 define three particular blizzard robots, `Io`, `Ganymede`, and `Callisto`. Statements 5-7 bind an instance of the cleanup agent to each of the robots. Statement 8 creates a society of the three robots and gives it the name `janitor`. Statement 9 specifies that the `janitor` society is the top level in the configuration.

This binding process completes construction of the configuration bound to the three available blizzard robots. The configuration is now ready for the code generators to create executables for each of the three robots. Once the executables are complete, the configuration can be deployed on the vehicles and executed.

The graphical configuration editor built into the *MissionLab* toolset (presented in the next section) supports automatic binding of configurations to robots. When the user clicks on the `bind` button, the system analyzes the configuration, matching output and input binding points to robot capabilities. It attempts to minimize the number of robots required to deploy a configuration and prompts for user input when

```
   /* Define new blizzard class of robots */
1. defRobotModel AuRA blizzard(
      movement wheelActuator; objlist objectSensor);

   /* Specify there are three blizzard robots */
2. defRobot Io isA blizzard;
3. defRobot Ganymede isA blizzard;
4. defRobot Callisto isA blizzard;

   /* Bind the robots to copies of cleanup agent */
5. bindRobot Io(wheelActuator =
      cleanup(objects=objectSensor));
6. bindRobot Ganymede(wheelActuator =
      cleanup(objects=objectSensor));
7. bindRobot Callisto(wheelActuator =
      cleanup(objects=objectSensor));

   /* Create uncoordinated society of the agents */
8. instAgent janitor from IndependentSociety(
      Agent[A]=Io,
      Agent[B]=Ganymede,
      Agent[C]=Callisto);

   /* Specify janitor agent as basis of configuration */
9. janitor;
```

Figure 4.10: CDL description of `janitor` configuration bound to the three trash collecting robots. Notice that Statements 8 and 9 are unchanged from the initial design in Figure 4.4, demonstrating the separation of binding issues from generic portions of the configuration.

choices are required. This vastly simplifies the binding process and promotes the creation of generic configurations.

4.3 CDL Syntax: An Attribute Grammar

The examples in the previous section were an attempt to provide some intuition for how CDL is used to specify configurations. Now it is time to formally define the language. Construction of a formal definition of the context-sensitive syntax of CDL is necessary to ensure that the language is completely and unambiguously specified. Presenting only a definition of the context-free aspects would leave various constructions open to differing interpretations and thus subvert attempts by others to implement compatible systems. Examples include scoping rules limiting which names are visible to the compiler and type checking requirements in assignments.

The choice of which formalization technique to use in specifying the CDL language was made after considering several choices. The traditional *Backus-Naur Form* (BNF) grammars can be used to specify only the context-free aspects of a language. It is then necessary to specify the context-sensitive features in an *ad hoc* manner, perhaps in tabular or narrative form. Two-level grammars are capable of representing context-sensitive language features through the use of meta-grammars. The designer specifies a grammar used to construct the actual grammar for the language. This allows the creation of an infinite number of actual productions in the target grammar using a small number of rules within the meta-grammar. However, it appeared that it would be quite complex to specify various aspects of CDL using a two-level grammar (*e.g.*, scoping rules). Therefore, the choice was made to utilize an attribute grammar.

Attribute grammars were first defined by Knuth in [41] and subsequently used in a *post priori* formal definition of the context-sensitive features of Pascal[80]. The syntax of CDL is defined using such an attribute grammar following the style promoted in [59]. Standard *Backus-Naur Form* (BNF) notation is used to establish the underlying context-free portions of the grammar but, in this style of grammar these productions are augmented with *attributes*, *evaluation rules*, and *conditions* to capture the context-sensitive portions of the language. The augmentations serve to prune the set of strings accepted by the context-free grammar to only those meeting the context sensitive restrictions. These restrictions capture requirements within the grammar itself such as type matching in assignments and the need to define objects before referencing them. The actual pruning of the acceptance set occurs through use of the *Assert* conditional operator (*Cond* in [59]). A rule augmented with an *Assert* conditional requirement still has the normal restriction on its application (in that the right-hand side must match) as well as the new restriction that the assertion must hold.

The productions for the CDL grammar are written in standard BNF, augmented with the attribute construction and evaluation rules. An example production is shown below:

$\langle$lhs$\rangle \Longleftarrow \langle$rhs$\rangle$
Attribute ← constructor
Assert: Attribute

Non-terminals are denoted as ⟨nonterm name⟩, and terminal symbols are specified as bold faced names such as `terminal` and sometimes are enclosed in single quotes like `'a'` when it might be confusing otherwise. The $\Longleftarrow$ operator separates the right-hand side of the production from the left. The rule

$$\langle a \rangle \Longleftarrow \langle b \rangle \, \mathtt{Z}$$

can be read as follows:[2] A non-terminal symbol $\langle b \rangle$ followed by the terminal symbol `Z` can both be replaced by the non-terminal symbol $\langle a \rangle$. Attribute construction is denoted by the ← assignment operator, and attribute evaluation occurs through use of the *Assert* conditional operator. A particular production is only allowed to fire when both the right-hand side matches and any conditionals attached to the rule are satisfied.

Match variable declarations like "int a" and "int b".
1a. ⟨vardef⟩ $\Longleftarrow$ ⟨type name⟩ ⟨name⟩ ;
1b. DataType (⟨name⟩) ← Tag (⟨type name⟩)
1c. SymbolType (⟨name⟩) ← {*Variable*}

If a name refers to a variable, rewrite it as "variable name".
2a. ⟨variable name⟩ $\Longleftarrow$ ⟨name⟩ ;
2b. *Assert* : Equal (SymbolType (⟨name⟩) , {*Variable*})

Match assignment statements like "a = b", but only when their types match.
3a. ⟨statement⟩ $\Longleftarrow$ $\langle$variable name$\rangle_1$ '=' $\langle$variable name$\rangle_2$;
3b. *Assert* : Equal (DataType ($\langle$variable name$\rangle_1$) , DataType ($\langle$variable name$\rangle_2$))

Figure 4.11: Portion of attribute grammar for assignment statements.

---

[2] This is an acceptance-style description. It is equally valid to discuss a grammar as generating the right-hand side from the left.

Figure 4.11 could be a portion of a grammar used to ensure type matching in assignment statements. Line 1a of Figure 4.11 is a standard BNF production rule for the non-terminal symbol `vardef`. The convention is followed in the grammar that non-terminals are surrounded in brackets, yielding ⟨vardef⟩. Therefore, Line 1a says that a non-terminal ⟨type name⟩ symbol next to a non-terminal ⟨name⟩ symbol can be reduced to a ⟨vardef⟩.

Line 1b is an example of the use of inherited attributes. The `DataType` attribute for the variable name ⟨`name`⟩ is set to the textual name of the ⟨`type name`⟩. Therefore, the variable's data type will be the name of the "type" used in its definition. Line 1c sets the `SymbolType` attribute to the *Variable* keyword, allowing subsequent rules to validate its usage.

Line 2a is a rule which allows use of the symbol ⟨`variable name`⟩ for ⟨`name`⟩ symbols which have been defined to be variables. Notice that the assertion in Line 2b will prevent this rule from applying to any ⟨`name`⟩ symbols which do not have the `SymbolType` attribute set to *Variable*. This rule allows other rules dealing with variables to be simpler, since they all do not need to verify that ⟨`name`⟩ is actually a variable.

Line 3a is a production rule matching variable to variable assignments. The assertion in Line 3b ensures that the types for the two variables are the same by forcing the `DataType` attributes to be equivalent. This textual matching of the type names is quite restrictive, requiring an exact match of the type names.

Given the grammar fragment in Figure 4.11, and the following code fragment:

```
int a;
int b;
string c;

a = b;
a = c;
```

The statement `a = b` is legal, while `a = c` will not be accepted since the assertion requiring type consistency is not satisfied.

4.3.1 The CDL Attribute Grammar

The `Assert` conditional tests attached to rules within the grammar allow constraining the application of the rule to just those cases where the assertion holds. The pieces of information that the assertions test are called **attributes**. The attributes are chunks of information that are attached to non-terminal symbols within the grammar and then copied, modified, or tested by rules which mention the non-terminal. There are five different attributes used within the CDL grammar (`Tag`, `Parms`, `DeferredParms`, `Definition`, and `SymbolTable`) and each requires explanation. A tabular description of the attributes and their associated forms is presented in Table 4.1.

Table 4.1: The attributes used within the CDL grammar.

CDL Attributes and Values

Attribute	Values
`Tag`	sequences of letters ('a', . . . , 'z') Each ⟨name⟩ symbol has its print string attached as a `Tag` attribute
`Parms`	sets of tuples of the form (`Type`, `Tag`) Parameter definitions are specified using a `Parms` attribute
`DeferredParms`	sets of Parms which have been deferred in an instantiation.
`Definition`	n-tuple of the form determined by the first field: Define a new data type. ('Type', `Tag`, "StreamIn", "StreamOut", "TextIn", "TextOut", "Validation") Define a new run-time architecture. ('Arch', `Tag`) Define a new class of robots. `Parms`$_1$ is the list of sensors, `Parms`$_2$ is the list of actuators. ('Robot', `Tag`, `Parms`$_1$, `Parms`$_2$) Define a new primitive so it is available for instantiation. ('Prim', `Tag`, OutputType, `Parms`) Define a new class of input binding points. ('IBP', `Tag`, OutputType) Define a new class of output binding points. ('OBP', `Tag`, InputType) Define a new agent. ('Agent', `Tag`, OutputType)
`SymbolTable`	set of Definitions There are two symbol tables: *GlobalSymbolTable* and *ArchSymbolTable*.

The attribute `Tag` is a character string containing the name of something (*e.g.*, a type name, an agent name, etc.). For simplicity, and with no loss of generality, this grammar restricts the allowable character set for such names to just the lowercase letters.

The attribute `Parms` is a set of tuples defining the interface to an object. Each element of the set denotes the type and name of a particular parameter (*e.g.*, {(int a),(string b)}). This attribute is a set and not an ordered list since, in CDL, parameter assignment is by name and not by position.

When a new composite agent is defined as an assemblage of other agents it is likely premature to commit to values for some of the parameters. These parameters are indicated by assigning them to the deferral operator ({^}). This set of deferred parameters defines the interface for the new composite agent. The attribute `DeferredParms` is a set of `Parms` which have been deferred in parameter assignments in such a construction.

The `Definition` attribute is the most complicated. It contains the information required to define a particular object. Since the type of information required varies based on the kind of object being defined, the fields in a `Definition` also vary. The first field in the record specifies the kind of object being defined and determines the remaining record fields.

Definition of a new data type in CDL creates a `Definition` of the form

$$('Type', \texttt{Tag}, \text{“StreamIn”}, \text{“StreamOut”}, \text{“TextIn”}, \text{“TextOut”}, \text{“Validation”})$$

where `Tag` is the name the user gave to this new type. The `StreamIn` and `StreamOut` parameters are text strings giving the name of functions the code generators should reference to convert instances of the data type from and to machine-independent byte streams. The `TextIn` and `TextOut` parameters are text strings giving the name of functions the code generators should reference to convert instances of the data type from and to human readable text strings. In particular, initializer data will be loaded using the `TextIn` function. The `Validation` parameter is the textual description of a LISP function to be used to validate initializer data before it is passed to the `TextIn` input function to ensure that the data is correctly formed.

Defining a new architecture generates a declaration $('Arch', \texttt{Tag})$, where `Tag` is the name of the new architecture. Definition of a new class of robots requires more information and generates a declaration of the form $('Robot', \texttt{Tag}, \texttt{Parms}_1, \texttt{Parms}_2)$, where `Tag` is the name of this new robot class, $\texttt{Parms}_1$ is the list of physical sensors available on robots of this style, and $\texttt{Parms}_2$ is the list of actuators. Definition of a new software primitive is represented as $('Prim', \texttt{Tag}, \text{OutputType}, \texttt{Parms})$, where the name of the new primitive is `Tag`, the data type for the output value is `OutputType`, and the parameter list for the module is `Parms`.

The input and output binding point definitions are represented using the triples $('IBP', \texttt{Tag}, \mathrm{OutputType})$ and $('OBP', \texttt{Tag}, \mathrm{OutputType})$, respectively. The `Tag` is the name of the binding point and `OutputType` constrains the classes of sensors and actuators which can be attached.

The final type of definition is generated to specify a new agent. The form of the definition is $('Agent', \texttt{Tag}, \mathrm{OutputType})$ where `Tag` is the name of the agent and `OutputType` defines the type of value generated by the agent.

The `SymbolTable` attribute is a set of `Definitions` which define the symbols visible to the parser. Two symbol tables are used in the grammar, the *GlobalSymbolTable* and the *ArchSymbolTable*. The global table captures definitions which are global in scope and should be visible to all. An architecture-specific table is attached to each defined architecture and captures definitions specific to a particular architecture. Only one *ArchSymbolTable* is active at any one time.

Each non-terminal symbol in the grammar may have both inherited and synthesized attributes with which it is associated, although it is likely that the rules within the grammar will not use all the available attributes on any particular class of non-terminals such as $\langle\texttt{name}\rangle$. Consider a particular non-terminal symbol occurring on the left-hand side of a production in a grammar. The set of inherited attributes are those already attached to the symbol when the right-hand side of a particular rule is matched and the synthesized attributes are those assigned values by functions attached to the rule and are passed back up the parse tree.

Inherited attributes are passed down the parse tree from above and are available when the right-hand side of a rule matches the input so that conditionals attached to the rule can check if the rule should fire. For example, in the following grammar fragment the attribute `SymbolTable` is inherited and available for testing within the production. The conditional can then easily verify that the types of the two variable names match (based on their most recent definitions) before allowing the production to match the string.

$$\begin{aligned}
&\langle\text{statement}\rangle \Longleftarrow \langle\text{name}\rangle_1 = \langle\text{name}\rangle_2\\
&\qquad\qquad \mathit{Assert}\colon \mathit{Equal}(\mathit{latesttype}\,(\mathit{Tag}\,(\langle\text{name}\rangle_1)\,, \mathit{SymbolTable}\,(\langle\text{statement}\rangle))\,,\\
&\qquad\qquad\qquad\qquad \mathit{latesttype}\,(\mathit{Tag}\,(\langle\text{name}\rangle_2)\,, \mathit{SymbolTable}\,(\langle\text{statement}\rangle))\,)
\end{aligned}$$

The second and more common class of attributes is synthesized by functions attached to the right-hand side of productions. The action of the production firing executes attached statements which modify the synthesized attribute associated with the left-hand side of the rule. For example, in the following grammar fragment the attribute `Tag` is created for $\langle\texttt{letter sequence}\rangle$ as the concatenation of the inherited `Tag` attributes for $\langle\texttt{letter sequence}\rangle_1$ and $\langle\texttt{letter}\rangle$.

$$\langle\text{letter sequence}\rangle \Longleftarrow \langle\text{letter sequence}\rangle_1\, \langle\text{letter}\rangle$$

$$\text{Tag}\left(\langle\text{letter sequence}\rangle\right) \leftarrow \mathit{concat}\left(\text{Tag}\left(\langle\text{letter sequence}\rangle_1\right), \text{Tag}\left(\langle\text{letter}\rangle\right)\right)$$

Each of the non-terminal symbols used in the CDL grammar is listed in Table 4.2 along with the number of the rule which defines it in the grammar.

Table 4.3 lists the non-terminal symbols which also carry attributes around within the grammar. Although there are five attributes available, only a small subset are ever used with any given symbol. This table allows looking up a particular symbol to see which attributes are actually used. The distinction between inherited and synthesized attributes helps to check which attributes should be defined in any particular instance of the symbol within productions in the grammar. A dash (—) is used to indicate that no attributes of that class are used with the indicated symbol.

Table 4.2: The set of non-terminal symbols and the rule which defines them in the grammar.

The Non-Terminal Symbols and Defining Rule

Rule	Symbol Name
1	⟨letter⟩
2	⟨digit⟩
3	⟨character sequence⟩
4	⟨name⟩
5	⟨data type⟩
6	⟨robot class⟩
7	⟨architecture defs⟩
8	⟨global defs⟩
9	⟨preamble⟩
10	⟨agent⟩
11	⟨parm def⟩
12	⟨parm list⟩
13	⟨parmset list⟩
14	⟨parmset⟩
15	⟨binding list⟩
16	⟨binding⟩
17	⟨configuration⟩

Table 4.3: The set of non-terminal symbols which carry attributes within the grammar. A dash (—) indicates no attributes of that class are used with that symbol.

Non-Terminal Symbols and Associated Attributes

Nonterminal	Inherited attributes	Synthesized attributes
⟨letter⟩	—	Tag
⟨digit⟩	—	Tag
⟨character sequence⟩	—	Tag
⟨name⟩	—	Tag, Type
⟨data type⟩	—	Tag, Definition
⟨robot class⟩	—	Tag, Definition
⟨parm def⟩	Parms	—
⟨parm list⟩	Parms	—
⟨parmset list⟩	DeferredParms	Parms
⟨parmset⟩	DeferredParms	Parms
⟨binding list⟩	Definition	Parms
⟨binding⟩	Definition	Parms

The functions used to manipulate attributes within the grammar are defined in Table 4.4. These functions are used within production in the grammar to define and check the values of attributes associated with symbols.

Table 4.4: Attribute manipulation functions used within the grammar.

---

1. Return the first element in a sequence.
   $\mathit{first}(s)$
      The first element of the sequence s.

2. Return all but the first element in a sequence.
   $\mathit{rest}(s)$
      The sequence s with the first element removed.

3. Return the concatenation of all the inputs.
   $\mathit{concat}(\text{item}_1, \text{item}_2, \ldots, \text{item}_n)$
      The sequence resulting from juxtaposing each of the sequences $\text{item}_1, \text{item}_2, \ldots, \text{item}_n$.

4. Check if a name is defined in the specified symboltable.
   $\mathit{IsUndefined}(SymbolTable, tag)$
      $true$ if SymbolTable $= <>$
      $false$, if $\exists d \in \mathit{first}(SymbolTable)$ where $d = (p, tag, q)$ for some p, q;
      $\mathit{IsUndefined}($ rest(SymbolTable), tag), otherwise.

5. Return a copy of the named definition from the specified symboltable.
   $\mathit{LookUpSymbol}(SymbolTable, tag)$
      $\langle$'undefined'$\rangle$ if SymbolTable $= <>$
      d, if $\exists d \in \mathit{first}(SymbolTable)$ where $d = (p, tag, q)$ for some p, q;
      $\mathit{LookUpSymbol}($ rest(SymbolTable), tag), otherwise.

6. Return a copy of the named parameter definition from the `parms` list.
   $\mathit{LookUpParm}(Parms, tag)$
      $\langle$'undefined'$\rangle$ if Parms $= <>$
      d, if $\exists d \in \mathit{first}(Parms)$ where $d = (p, tag, q)$ for some p, q;
      $\mathit{LookUpSymbol}($ rest(Parms), tag), otherwise.

7. Check if the two sequences are equivalent.
   $\mathit{Equal}(tag_1, tag_2)$
      true, if $(tag_1 = \emptyset) \wedge (tag_2 = \emptyset)$;
      false, if $first(tag_1) \neq first(tag_2)$;
      $\mathit{equal}(\mathit{rest}(tag_1), \mathit{rest}(tag_2))$, otherwise.

8. Check if the initializer is consistent with the data type.
   $\mathit{Conformable}(t, str)$
      true, if the initializer string str can be converted to the type t; false, otherwise.

The CDL attribute grammar is defined below. Each production is commented and numbered. The comments use Roman typeface while symbols and operators are in `typewriter` font.

Production and Attribute Evaluation Rules for CDL

1. A `letter` is one of the upper or lower case characters and has an attribute `Tag` which is the textual value of the `letter`.
⟨letter⟩ ⟸
a
Tag (⟨letter⟩) ← ⟨‘a’⟩
| ...
...
| **z**
Tag (⟨letter⟩) ← ⟨‘z’⟩
| **A**
Tag (⟨letter⟩) ← ⟨‘A’⟩
| ...
...
| **Z**
Tag (⟨letter⟩) ← ⟨‘Z’⟩

2. A `digit` is one of the characters 0 to 9.
⟨digit⟩ ⟸
0
Tag (⟨digit⟩) ← ⟨‘0’⟩
| ...
...
| **9**
Tag (⟨digit⟩) ← ⟨‘9’⟩

3. A `character sequence` is a string of one or more characters and the `Tag` attribute for the sequence is the textual value of the string.
⟨character sequence⟩ ⟸
⟨letter⟩
Tag (⟨letter sequence⟩) ← Tag (⟨letter⟩)
| ⟨digit⟩
Tag (⟨letter sequence⟩) ← Tag (⟨digit⟩)
| ‘_’
Tag (⟨character sequence⟩) ← ⟨‘_’⟩
| ⟨letter sequence⟩$_2$ ⟨letter⟩
Tag (⟨letter sequence⟩) ← *concat* (Tag (⟨letter sequence⟩$_2$) , Tag (⟨letter⟩))
| ⟨letter sequence⟩$_2$ ⟨digit⟩
Tag (⟨letter sequence⟩) ← *concat* (Tag (⟨letter sequence⟩$_2$) , Tag (⟨digit⟩))
| ⟨letter sequence⟩$_2$ ‘_’
Tag (⟨letter sequence⟩) ← *concat* (Tag (⟨letter sequence⟩$_2$) , ⟨‘_’⟩)

4. Define `name` as string of letters. The `Tag` attribute is copied from the sequence.

```
⟨name⟩ ⟸
      ⟨letter⟩
          Tag(⟨name⟩) ← Tag(⟨letter⟩)
    | ⟨letter⟩⟨letter sequence⟩
          Tag(⟨name⟩) ← concat(Tag(⟨letter⟩), Tag(⟨letter sequence⟩))
```

5. Define `data type` as name which has been defined using **defType.**

```
⟨data type⟩ ⟸
      ⟨name⟩
          Make sure the data type is defined.
          Let a = LookUpSymbol(GlobalSymbolTable, Tag(⟨name⟩))
          Assert: Equal(SymbolType(a), {Type})
          Pass the definition for the data type along with the symbol.
          Definition(⟨data type⟩) ← a
          Tag(⟨data type⟩) ← Tag(⟨name⟩)
```

6. Define `robot class` as name which has been defined using **defRobotClass.**

```
⟨robot class⟩ ⟸
      ⟨name⟩
          Look up the definition for ⟨name⟩ and make sure it is a robot class.
          Let a = LookUpSymbol(GlobalSymbolTable, Tag(⟨name⟩))
          Assert: Equal(SymbolType(a), {Robot})
          Pass the definition for the robot class along with the symbol.
          Definition(⟨robot class⟩) ← a
          Tag(⟨robot class⟩) ← Tag(⟨name⟩)
```

7. Those definitions available only within a specific architecture.

```
⟨architecture defs⟩ ⟸
      Define a new input binding point within the current architecture.
      defInputBindingPoint ⟨data type⟩⟨name⟩
          ArchSymbolTable ← ArchSymbolTable ∪ {IBP, Tag(⟨name⟩), Tag(⟨data type⟩)}

      Define a new output binding point within the current architecture.
    | defOutputBindingPoint ⟨data type⟩⟨name⟩
          ArchSymbolTable ← ArchSymbolTable ∪ {OBP, Tag(⟨name⟩), Tag(⟨data type⟩)}

      Define a new class of robots.
    | defRobotClass ⟨name⟩(⟨parm list⟩₁, ⟨parm list⟩₂)
          ArchSymbolTable ← ArchSymbolTable ∪
                {Robot, Tag(⟨name⟩), Parms(⟨parm list⟩₁), Parms(⟨parm list⟩₂)}

      Define a new type of coordination operator.
    | defOperator ⟨name⟩(⟨parm list⟩₁, ⟨parm list⟩₂)
          ArchSymbolTable ← ArchSymbolTable ∪
                {Prim, Tag(⟨name⟩), Parms(⟨parm list⟩₁), Parms(⟨parm list⟩₂)}
```

```
    Define a new agent.
  | defAgent ⟨name_1⟩ From ⟨name_2⟩ (⟨parm list⟩)
        Make sure the name we are building on is defined appropriately.
        Let $a$ = LookUpSymbol(ArchSymbolTable, Tag(⟨name⟩_2))
        Assert: Equal(SymbolType(a), {Prim})
        Create a definition for the new primitive and add it to the symbol table.
        ArchSymbolTable ← ArchSymbolTable ∪
            {Prim, Tag(⟨name⟩_1), ThirdField(a), Parms(⟨parm list⟩)}

    Define a primitive behavior.
  | defPrimitive ⟨data type⟩ ⟨name⟩ (⟨Parm list⟩)
        ArchSymbolTable ← ArchSymbolTable ∪
            {Prim, Tag(⟨name⟩), Tag(⟨data type⟩), Parms(⟨parm list⟩)}
```

8. Those definitions which create symbols globally scoped.

```
⟨global defs⟩ ⟸
    Define a new target robot architecture.
    defArch ⟨name⟩
        Make sure the name is not already in use.
        Assert: IsUndefined(GlobalSymbolTable, Tag(⟨name⟩))
        Define the new architecture.
        GlobalSymbolTable ← GlobalSymbolTable ∪ {Arch, Tag(⟨name⟩)}

    Define a new data type.
  | defType ⟨name⟩ "StreamIn" "StreamOut" "TextIn" "TextOut" "Validation"
        Make sure the name is not already in use.
        Assert: IsUndefined(GlobalSymbolTable, Tag(⟨name⟩))
        Define the new type.
        GlobalSymbolTable ← GlobalSymbolTable ∪
            {Type, Tag(⟨name⟩), "StreamIn", "StreamOut",
             "TextIn", "TextOut", "Validation"}
```

9. Collect all the rules which defined types, primitives, *etc.* into a single entity called the preamble

```
⟨preamble⟩ ⟸
    ⟨global defs⟩
  | ⟨architecture defs⟩
  | useArchitecture ⟨name⟩
        Find the architecture definition and make sure it is valid.
        Let $a$ = LookUpSymbol(GlobalSymbolTable, Tag(⟨name⟩))
        Assert: Equal(SymbolType(a), {Arch})
        Switch to the desired architecture.
        ArchSymbolTable ← $a$

  | ⟨preamble⟩_1 ⟨global defs⟩
  | ⟨preamble⟩_1 ⟨architecture defs⟩
  | ⟨preamble⟩_1 useArchitecture ⟨name⟩
        Find the architecture definition and make it sure is valid.
        Let $a$ = LookUpSymbol(GlobalSymbolTable, Tag(⟨name⟩))
```

```
            Assert: Equal (SymbolType ($a$) , {Arch})
            Switch to the desired architecture.
            ArchSymbolTable ← $a$

10. Instantiate a new agent
    ⟨agent⟩ ⟸
          Instantiate an anonymous (unnamed) agent
        ⟨name⟩ (⟨parmset list⟩)
            Look up the definition for ⟨name⟩ and make sure it is a primitive.
            Let $a$ = LookUpSymbol (GlobalSymbolTable, Tag (⟨name⟩))
            Assert: Equal (SymbolType ($a$) , {Prim})
            Pass the parameter list for the definition down to the parameter assignment rules.
            Parms (⟨parmset list⟩) ← FourthField ($a$)


          Instantiate named agents.
        | instAgent ⟨name⟩$_1$ From ⟨name⟩$_2$ (⟨parmset list⟩)
            Look up the definition for ⟨name⟩$_2$ and make sure it is a primitive.
            Let $a$ = LookUpSymbol (GlobalSymbolTable, Tag (⟨name⟩$_2$))
            Assert: Equal (SymbolType ($a$) , {Prim})
            Pass the parameter list for the definition down to the parameter assignment rules.
            Parms (⟨parmset list⟩) ← FourthField ($a$)
            Create a definition for the new agent and add it to the symbol table.
            ArchSymbolTable ← ArchSymbolTable ∪
                  {Agent, Tag (⟨name⟩$_1$) , Tag (⟨name⟩$_2$)}

          A bound robot is treated as an anonymous (unnamed) agent.
        | bindrobot (⟨agent⟩ , ⟨robot class⟩ , ⟨binding list⟩)
            Pass the definition for the robot class down to the parameter assignment rules.
            Definition (⟨binding list⟩) ← Definition (⟨robot class⟩)

11. Define a parameter as part of a primitive definition.
    ⟨parm def⟩ ⟸
        ⟨data type⟩ ⟨name⟩
            Construct a parameter definition and pass it up.
            Parms(⟨parm def⟩) ← {(Tag (⟨data type⟩) , Tag (⟨name⟩))}

12. Collect the list of parameters for a primitive definition.
    ⟨parm list⟩ ⟸
        ⟨parm list⟩$_2$ ',' ⟨parm def⟩
            Combine the parameter definitions and return them.
            Parms(⟨parm list⟩) ← Parms (⟨parm list⟩$_2$) ∪ Parms (⟨parm def⟩)

        | ⟨parm def⟩
            Return the parameter definitions.
            Parms(⟨parm list⟩) ← Parms (⟨parm def⟩)
```

```
13. Match a list of parameter assignments.
    $\langle\texttt{parmset list}\rangle \Longleftarrow$
        $\langle\texttt{parmset list}\rangle_1$ ‘,’ $\langle\texttt{parmset}\rangle$
            Pass the parameter list for the definition down to the assignment rules.
            $\texttt{Parms}(\langle\texttt{parmset}\rangle) \leftarrow \texttt{Parms}(\langle\texttt{parmset list}\rangle)$
            Combine and return any deferred parameters.
            $\texttt{DeferredParms}(\langle\texttt{parmset list}\rangle) \leftarrow$
                $\texttt{DeferredParms}(\langle\texttt{parmset list}\rangle_2) \cup \texttt{DeferredParms}(\langle\texttt{parmset}\rangle)$

      | $\langle\texttt{parmset}\rangle$
            Pass the parameter list for the definition down to the assignment rules.
            $\texttt{Parms}(\langle\texttt{parmset}\rangle) \leftarrow \texttt{Parms}(\langle\texttt{parmset list}\rangle)$
            Return this deferred parameter.
            $\texttt{DeferredParms}(\langle\texttt{parmset list}\rangle) \leftarrow \texttt{DeferredParms}(\langle\texttt{parmset}\rangle)$

14. Assign a value to a parameter.
    $\langle\texttt{parmset}\rangle \Longleftarrow$
        The value is a constant.
        $\langle\texttt{name}\rangle$ = $\{\texttt{initializer string}\}$
            Look up the parameter.
            Let $p = \texttt{LookUpParm}(\texttt{Parms}(\langle\texttt{parmset}\rangle), \texttt{Tag}(\langle\texttt{name}\rangle))$
            Make sure the parameter is defined.
            $\textit{Assert}$: $\texttt{NotEqual}(p, \emptyset)$
            Check that the initializer string makes sense given the type of the parameter.
            $\textit{Assert}$: $\textit{Conformable}(\texttt{Type}(p), \{\texttt{initializer string}\})$

        The value is generated by an agent.
      | $\langle\texttt{name}\rangle$ = $\langle\texttt{name}\rangle_2$
            Make sure parameter and output type of the agent have the same data type.
            Let $p = \texttt{LookUpParm}(\texttt{Parms}(\langle\texttt{parmset}\rangle), \texttt{Tag}(\langle\texttt{name}\rangle))$
            Let $a = \texttt{LookUpSymbol}(mboxArchSymbolTable, \texttt{Tag}(\langle\texttt{name}\rangle_2))$
            $\textit{Assert}$: $\texttt{Equal}(\texttt{Type}(p),\ \texttt{Type}(a))$

        The value is deferred and will be assigned by the user of this module.
      | $\langle\texttt{name}\rangle$ = $\{\hat{}\}$
            Look up the parameter.
            Let $p = \texttt{LookUpParm}(\texttt{Parms}(\langle\texttt{parmset}\rangle), \texttt{Tag}(\langle\texttt{name}\rangle))$
            Make sure the parameter is defined.
            $\textit{Assert}$: $\texttt{NotEqual}(p, \emptyset)$
            Return this deferred parameter.
            $\texttt{DeferredParms}(\langle\texttt{parm set}\rangle) \leftarrow \texttt{DeferredParms}(\langle\texttt{parm set}\rangle_2) \cup p$

15. Collect the list of hardware bindings for a robot.
    $\langle\texttt{binding list}\rangle \Longleftarrow$
        $\langle\texttt{binding list}\rangle_1$ , $\langle\texttt{binding}\rangle$
            Pass the port list for the robot class down to the binding rules.
            $\texttt{Parms}(\langle\texttt{binding}\rangle) \leftarrow \texttt{Parms}(\langle\texttt{binding list}\rangle)$
```

$\mid\ \langle\texttt{binding}\rangle$

Pass the port list for the robot class down to the binding rules.

$\texttt{Parms}\,(\langle\texttt{binding}\rangle) \leftarrow \texttt{Parms}\,(\langle\texttt{binding list}\rangle)$

16. Specify binding a sensor or actuator to a binding point.

$\langle\texttt{binding}\rangle \Longleftarrow$

$\langle\texttt{name}\rangle_1$ ‘=’ $\langle\texttt{name}\rangle_2$

Make sure the two names have the same data type.

$\texttt{Let}\ h = \texttt{LookUpParm}\,(\texttt{Parms}\,(\langle\texttt{binding}\rangle)\,, \texttt{Tag}\,(\langle\texttt{name}\rangle_1))$

$\texttt{Let}\ b = \texttt{LookUpSymbol}\,(mbox\mathit{ArchSymbolTable}, \texttt{Tag}\,(\langle\texttt{name}\rangle_2))$

$\mathit{Assert}:\ \texttt{Equal}\,(\texttt{Type}\,(h)\,,\ \texttt{Type}\,(b))$

17. The distinguished non-terminal (the start symbol) is the `configuration`.

$\langle\texttt{configuration}\rangle \Longleftarrow$

$\langle\texttt{preamble}\rangle\,\langle\texttt{agent}\rangle$

4.4 CDL Semantics: An Axiomatic Definition

The semantic description of a language serves as a design specification. While the syntax description defines what the language looks like and circumscribes the set of legal programs, it is the semantic definition of the language which states **what** those programs will actually do when they are executed.

There are three popular methods for specifying language semantics. The first is the operational approach, where the semantics are specified in terms of the operation of some abstract machine which executes the language being defined. The Vienna Definition Language (VDL)[81] is a meta-language used to construct such definitions. The second method is called the denotational approach, where a mathematical description is created to represent the meaning of each of the language constructs. A collection of semantic domains are specified and a corresponding set of semantic equations map programs onto these domains, thereby describing their meaning. The final method used to describe the semantic meaning of programming languages is the axiomatic approach. The axiomatic method is attributed to Hoare[32] and has been used to define the semantics of PASCAL[33]. This style uses a collection of axioms which are true of any correct program and serve to define the meaning of the program. These axioms provide a set of conditions which are to hold in any particular implementation of the language, without specifying how the system is to be implemented.

The Axiomatic technique was chosen for use in defining the semantics of CDL based on its ability to specify the semantic meaning without constraining the implementation. The style suggested in [59] has been followed as closely as possible in the following presentation. CDL is a language for describing how threads of execution are instantiated from externally defined procedures and, since configurations transcend individual robot run-time implementations, the semantic description of CDL centers on thread execution and data movement issues and not the implementations of the primitive behaviors.

4.4.1 Data Type Axioms

Recall the syntax for a data type definition named `T` reproduced here:

defType `T` “StreamIn” “StreamOut” “TextIn” “TextOut” “Validation”

Data types are merely symbolic tokens within CDL, since the underlying representations are implementation-dependent. To support movement of arbitrary data structures across thread, process, and machine boundaries, it is necessary for the user to specify procedures to convert an instance of the data type to a machine-independent byte stream and a procedure to reverse the process within the receiver.

The `StreamOut` parameter references a function in the run-time library which creates a byte stream version of an instance of the data type `T`. The `StreamIn` function can be used to reverse the process and construct an equivalent instance of the data type `T`. Axiom 1 formalizes the requirement that the `StreamIn` and `StreamOut` functions reverse each others effects.

Axiom 1 $StreamIn(StreamOut(i)) \equiv i \ \ \forall i \mid i \ instance \ of \ \mathtt{T}$

The `validation` parameter is a string of LISP code used by a suitable LISP interpreter to check the syntax of an arbitrary text string. It is responsible for verifying that the string is a valid specification of an instance of the data type `T`. Such validation procedures are desirable to allow checking that the initializer strings are well formed in assignments to constant inputs. The desire to validate initializer strings causes a rather large problem, making it necessary to allow specification of such a validation procedure as arbitrary executable code within the `defType` statement itself. Rather than encumbering CDL with general purpose programming constructs to solve this limited issue we choose to specify inclusion of a LISP subsystem with implementation-dependent capabilities that will process the validation procedures. The specific capabilities of the LISP are not restricted, allowing the designer to follow the most natural implementation (see Section 5.1.3 for an example). If validation is not desired, the LISP subsystem can be eliminated and the validation procedures ignored. Axiom 2 specifies that a validation function for data type `T` must be able to check if a string is a well formed initializer for data type `T`.

Axiom 2 *LISP(validation$_T$ initializer_string) $\Rightarrow$* `true` *iff the initializer_string is a well formed initializer for instances of data type* `T`

The `TextIn` parameter names the function in the run-time library to be used to build instances of the data type from user-specified initializer strings. The `TextIn` function will construct an instance of the data type from any text string acceptable to the validation function. Initializers are arbitrary strings from of the machine-dependent characters C and are denoted as C^* (zero or more characters from C). The `TextOut` function can be used to print a readable description of the object to aid debugging. Note: The output of `TextOut` need not match the input format for `TextIn`. The two text processing functions are optional and, if they are not specified, that capability is lacking for the data type. Axiom 3 specifies that the `TextIn` function must accept any initializer string which passes the LISP validation function.

Axiom 3 *If LISP(Validation str) $\Rightarrow$* `true` *then TextIn(str) is defined* $\forall str \in C^*$

4.4.2 Data Movement Axioms

The input values for agent parameters can either be constant values specified at compile time or the output of another agent. Consider a particular agent a which is an instance of the primitive $p(L)$ where L is the list of parameters. If $l \in L$ is a particular input parameter and if a is parameterized such that l is assigned a constant value as in

$$a\,(l = \{\text{initializer string}\})$$

then the input value is always available and there is no need to wait for another agent to generate it. Axiom 4 defines a function to allow checking if an input has been assigned a constant value.

Axiom 4 *IsConstantInput* $(l) =$ `'t` *iff* l *is assigned to a constant*

Now consider the other case where the input parameter $l \in L$ is assigned to receive the output of another agent. This generator agent could be instantiated inline as an anonymous agent or referenced by name if it was previously instantiated. The following example demonstrates both possibilities. The notation $b(\ldots)$ is used to indicate that b is properly parameterized, but the detail is unimportant for the current discussion.

$$\texttt{instAgent}\ n\ \texttt{from}\ b\,(\ldots)\,;a\,(l_1 \ = \ b\,(\ldots)\,,\ l_2 \ = \ n)$$

It is necessary for the outputs generated by b and n to be available through the parameter links l_1 and l_2. Axiom 5 formalizes the common sense notion that specifying an agent as the input for a parameter means the output of that agent will be the value of the parameter.

Axiom 5 *If* $a\,(l \ = \ v)$ *where* v *references an agent anonymously or by name, for each execution of* a, l *gets the last value generated by* v

4.4.3 Execution Axioms

From the execution point of view there are two classes of agents, active and inactive. Active agents are generating new output values at some rate while inactive agents are not executing. This distinction is made to allow ensuring that the input values being used by an active agent are all constant values or are, in turn, being generated by active agents.

A particular agent a which is an instance of the primitive $p\,(L)$ can not become active until all inputs in the list of parameters L are available. Parameters which are constant values are always available but those generated by other agents are only available if the source agent is also active. Axiom 6 formalizes the requirements that a particular agent is allowed to execute only when all of its input values are available.

Axiom 6 $\neg \mathit{IsActive}\,(a)$ *if* $\exists\, l \in L$ *where* $\neg\ \mathit{IsActive}\,(\mathit{GeneratorAgent}\,(l))\ \wedge\ \neg\ \mathit{IsConstantInput}\,(l)$

The notation $\{P\}S\{Q\}$ will be used to describe activation of an agent S, where P is a logical statement of the conditions existing before activation of S, and Q is a logical statement describing the results of the activation. In Axiom 7, ¬IsActive(a) is substituted for P as the precondition, $A\,(L)$ is substituted for S as the action, and IsActive(a) is substituted for Q as the post condition. This formalizes the meaning of the IsActive predicate.

Axiom 7 $\{\neg \mathit{IsActive}(a)\}\ A(L)\ \{\mathit{IsActive}(a)\}$

4.5 Summary

The Configuration Description Language was developed as an architectural specification for the **Societal Agent** theory presented in Chapter 3. CDL captures the important recursive nature of the **Societal Agent** theory in an architecture- and robot-independent language. The uniform representation of components at all levels of abstraction simplifies exchanging portions of configurations and facilitates reuse of existing designs. The ability to describe complicated structures in a compact language eliminates unexpected interactions, increases reliability, and reduces the design time required to construct mission specifications.

The context-sensitive syntax of CDL was formally defined using an attribute grammar. This specification goes beyond the traditional *Backus-Naur Form* (BNF) grammars by attaching attributes and conditional tests to productions. These conditionals are used to prune the set of strings accepted by the productions to just those meeting the context-sensitive requirements of the language.

The semantics of CDL were formally defined using an axiomatic definition. This technique uses a collection of logical statements to formally define the results of executing programs written in the language. These definitions serve as design specifications for implementers by specifying **what** programs are to do when executed. This nicely supplements the syntax definitions which specify only which programs are legal, and not what they actually do.

CDL strictly partitions specification of a configuration from the implementation of the underlying primitives. Further, the ability of CDL to rise above particular robot run-time architectures vastly increases its utility. It is now possible, using CDL, to specify configurations independent of constraints imposed by particular robots and architectures. Only after the mission has been developed do hardware binding issues need to be addressed. These contributions of generic configurations and explicit hardware bindings allow the construction of toolsets based on this architecture which provide the correct depths of presentation to various classes of users. This separation in structure supports the separation of presentation required to empower non-programmers with the ability to specify complex robot missions.

Chapter 5

Implementation: MissionLab

The *MissionLab* toolset has been developed based on the Configuration Description Language. Figure 5.1 shows a block diagram of the *MissionLab* system. The user interface centers around the graphical Configuration Editor (*CfgEdit*). From within that environment the user can develop configurations, bind them to specific robots, and generate executables. The CDL compiler generates either CNL or SAUSAGES code based on the robot to which the configuration has been bound. Built-in support for the AuRA architecture allows deploying and monitoring configurations on the multiagent simulation system and/or robots, all from within *MissionLab*.

CfgEdit is used to create and maintain configurations and supports the recursive construction of reusable components at all levels, from primitive motor behaviors to societies of cooperating robots. *CfgEdit* supports this recursive design process by allowing creation of coordinated assemblages of components which are then treated as atomic higher-level components available for later reuse. It allows deferring commitment (binding) to a particular robot architecture or specific vehicles until the configuration has been developed. This explicit binding step simplifies development of a configuration which may be deployed on several vehicles each requiring use of a specific architecture. The process of retargeting a configuration to a different vehicle when requirements change is similarly eased.

The capability exists to generate code for either the ARPA Unmanned Ground Vehicle (UGV) architecture or for the AuRA architecture. The AuRA executables drive both simulated robots and several types of Denning vehicles (DRV-1, MRV-2, MRV-3). The binding process determines which compiler will be used to generate the final executable code as well as which libraries of behavior primitives will be available for placement within the editor.

5.1 Graphic Designer

Reactive behavior-based architectures[3, 9] decompose a robot's control program into a collection of behaviors and coordination mechanisms. This decomposition allows

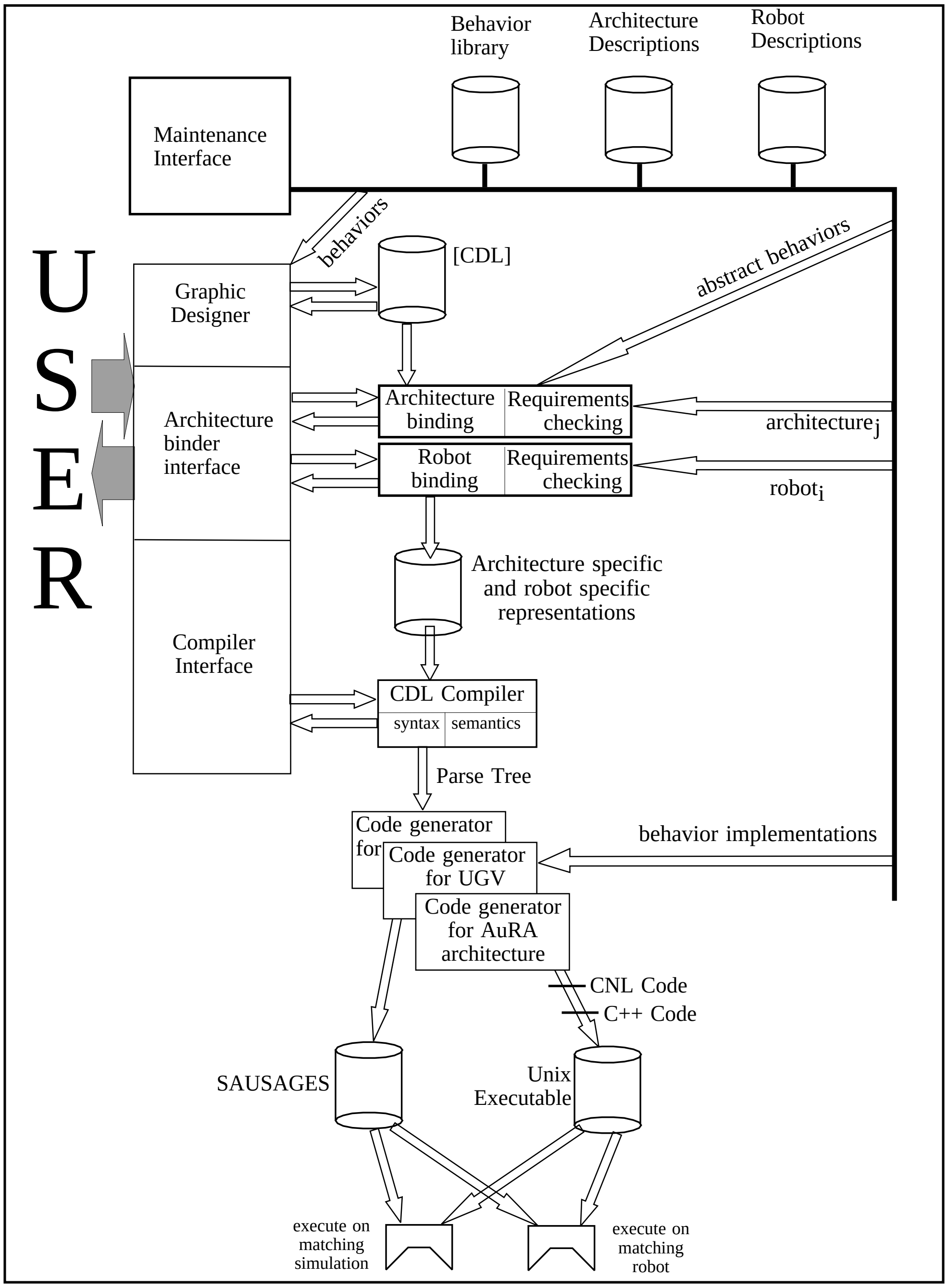


Figure 5.1: Block diagram of the *MissionLab* System.

construction of a library of reusable behaviors and assemblages of behaviors by designers skilled in low-level control issues. Subsequent developers using these components need only be concerned with their specified functionality. Creating a multiagent robot configuration using this paradigm involves three steps; determining an appropriate set of skills for each of the vehicles; translating those mission-oriented skills into sets of suitable behaviors (assemblages); and the construction/selection of suitable coordination mechanisms to ensure that the correct skill assemblages are deployed over the duration of the mission.

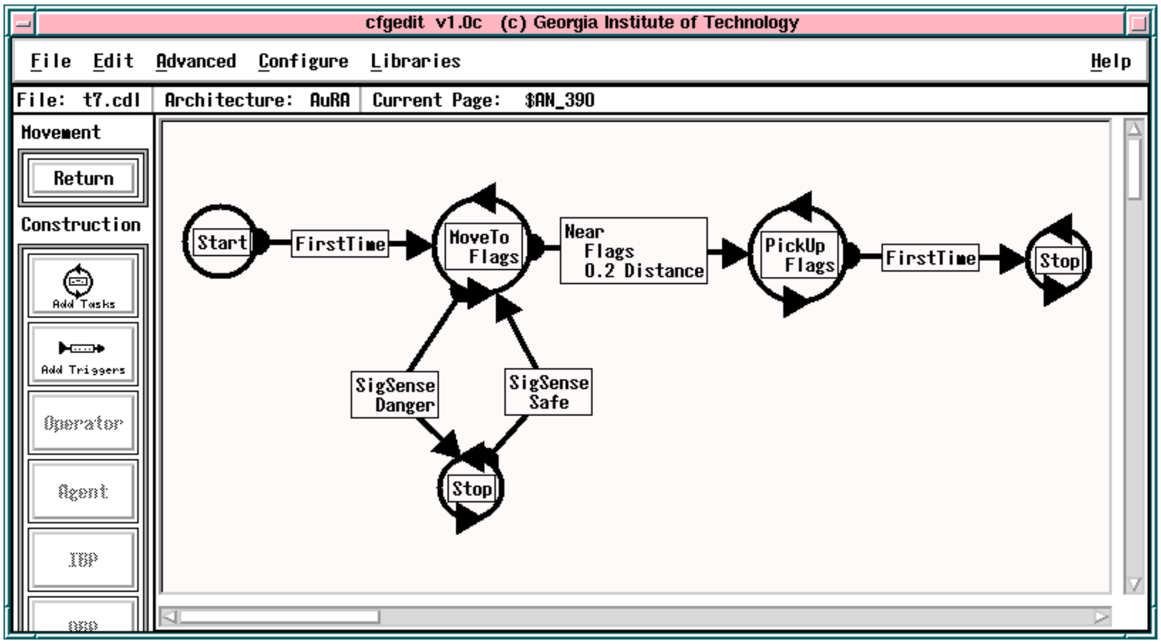


Figure 5.2: The Configuration Designer (*CfgEdit*) with an FSA displayed.

Mission specification is facilitated by encoding the steps in the mission using a Finite State Automaton (FSA). This makes explicit the various operating states and perceptual triggers causing transitions between states. Figure 5.2 shows *CfgEdit* with an illustrative FSA loaded. *CfgEdit* supports the graphical construction of such FSA's and the skill assemblages active in each state. The selection of archived library behaviors from pull-down menus supports use by personnel with minimal system training but who are skilled in the domain within which the robots are operating.

Support for users in the various stages of mission development (*e.g.*, behavior implementation, assemblage construction, and mission specification) are provided.

The primitive behavior implementor must be familiar with the particular robot architecture in use and a suitable programming language such as C++ or LISP. The assemblage constructor uses a library of behaviors to build skill assemblages using the graphical configuration editor. This allows visual placement and connection of behaviors without requiring programming knowledge. However, the construction of *useful* assemblages still requires knowledge of behavior-based robot control. Specifying a configuration for a robot team consists of selecting which of the available skills are useful for the targeted environments and missions.

5.1.1 Configuration Specification

The **Societal Agent** theory presents a uniform view of all robot configuration components as agents. These agents occur at various levels in the recursively constructed configuration and encapsulate skills of varying scope and complexity but are, in essence, equivalent from a representational viewpoint. Whether a *travel_along_road* agent consists of an assemblage of behaviors interacting in complex ways or is in fact a monolithic behavior is unimportant to the designer attempting to utilize that skill in a configuration. The whole purpose of the coordination operator is to **make** it unimportant!

Assemblages encapsulate a particular skill or ability which is a higher level construction than a behavior[52]. A simple skill such as *Wander* will include several behaviors: *Noise* to generate random motion, *Avoid_obstacles* to keep the robot from bumping into things, and perhaps *Follow_wall* to encourage exploration of the perimeter. A more complicated skill such as "follow road" might include several behaviors (*e.g.*, *stay_on_road*, *move_ahead*, and *avoid_static_obstacles*), other skill assemblages (*e.g.*, *avoid_dynamic_obstacles*) and perhaps even behaviors transcending the individual robot (*e.g.*, *maintain_formation*). In either case, the chosen coordination operator is responsible for ensuring that the group of behaviors can be treated as an integrated unit.

The ability to treat assemblages as coherent objects allows abstracting the details of the implementation and naturally suggests the following top-down design philosophy: Beginning with the top level, define the desired overt behavior, specify a collection of simpler, more focused, more concrete behaviors which could be used to generate the desired complex behavior, then recursively expand those new definitions until they bottom out in capabilities available from library components or behaviors which can be implemented as new primitives. Figure 5.3 enumerates these steps.

This top-down design process fosters reuse by supporting archival of components at all levels of detail. When a new component is created, it can be added to the library and made available for reuse in subsequent projects, independent of whether it is a primitive behavior or a configuration for a society of robots conducting a

```
cfg_design()
    If there exists a library component implementing the skill
      • Use the library component
      • Parameterize it
      • Return
    Else if the skill can be divided into temporally distinct operating states
      • Specify the operating states and perceptual triggers causing transitions between the states.
      • Decide on basis of design and select either an FSA or Mission Coordination operator (Section 5.1.4). Encode the state diagram in the selected format.
      • Specify the skill embodied in each state.
      • Recursively invoke cfg_design for each of these skills.
    Else if the skill can be decomposed into distinct behaviors
      • Specify the behaviors which combine to create the skill.
      • Create an assemblage using the cooperative coordination operator and these behaviors.
      • Recursively invoke cfg_design for each of these skills.
    Else
      • Implement a new primitive behavior to encompass the skill
      • Parameterize it
      • Return
End_Function
```

Figure 5.3: Pseudo code representation of the configuration design process.

search and rescue mission. Configuration design tools have been developed to reduce the workload on designers following this development process.

5.1.2 Use of the Graphic Designer

To demonstrate use of the graphic designer we will develop a configuration implementing a janitorial task for a team of robots (*e.g.*, 1994 AAAI mobile robot competition[5]). Reconsider the trash collecting state-transition diagram from Chapter 3 (Figure 3.12) reproduced in Figure 5.4. Each robot wanders around looking for empty soda cans, picks them up, wanders around looking for a recycling basket, and places the can into the basket. We will refer to the configuration fulfilling these design requirements as the *trashbot* configuration.

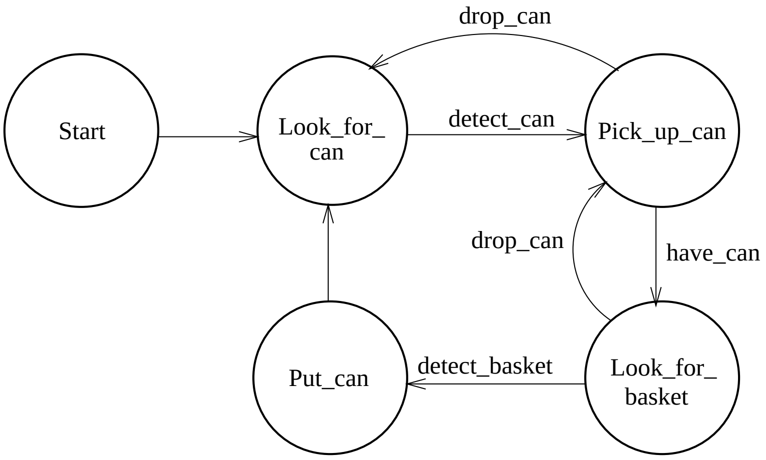


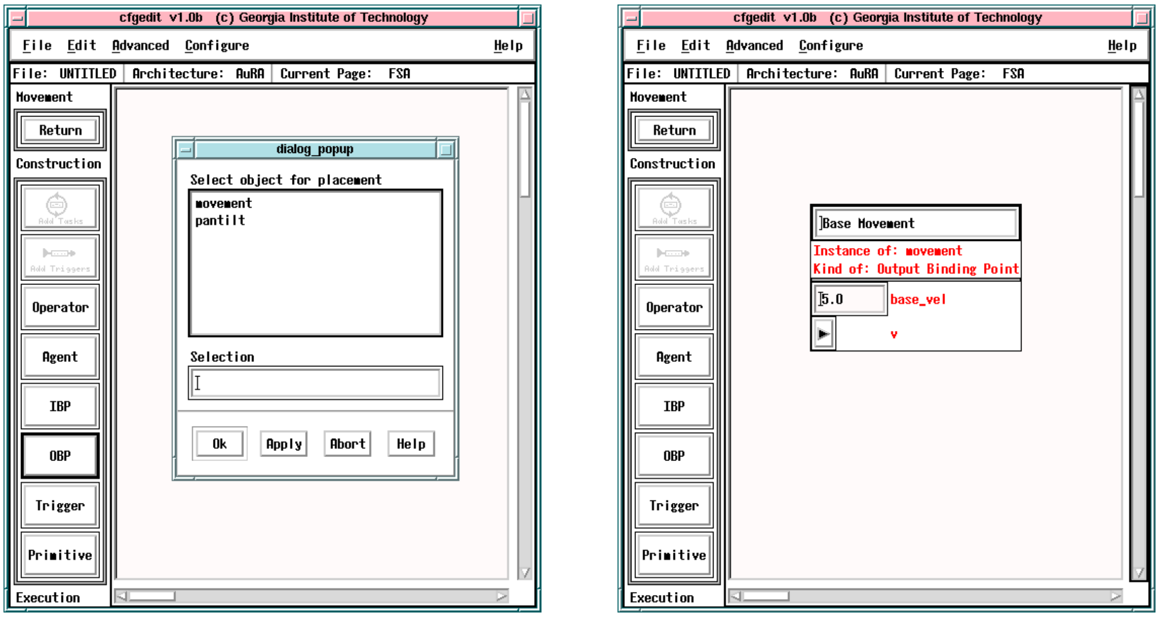
Figure 5.4: Reproduction of the trash collecting FSA.

To start construction an output binding point is placed in the workspace where the actuator to drive the wheels of the robot will later be attached. Figure 5.5a shows the Configuration Editor after the "OBP" button was pressed to bring up the list of possible output binding points. In this case, the movement binding point was selected. Figure 5.5b shows the workspace with the binding point in place.

During the design process it was determined that the top level of the *trashbot* configuration is temporally separable and best implemented using state-based coordination. Therefore, an FSA coordination operator will be used as the top level agent within the robot. The FSA operator is selected and placed in the workspace and then connected by clicking the left mouse button on the output and input arrows for the connection. Figure 5.5c shows the workspace after this connection is made.

The Configuration Editor supports the graphical construction of FSA's. In Figure 5.5d, the operator has moved into the FSA workspace and started to build the state diagram. The circles represent the two operating states within this simple FSA. The rectangle in the center of each circle lists the behavior which will be active when the robot is in that operating state. The arcs represent transitions between operating states, with the arrow heads denoting the direction of the transition. The icon near the center of the arc names the perceptual trigger activating the transition. Figure 5.6 shows the completed state machine implementing the trash collecting robot. Compare Figure 5.6 with the state diagram shown in Figure 5.4.

Clicking on states or transitions with the right button brings up the list of assemblages or perceptual triggers from which to choose. Figure 5.7a shows the list of assemblages available for use in states. The operation performed by a state is changed



Select object for placement

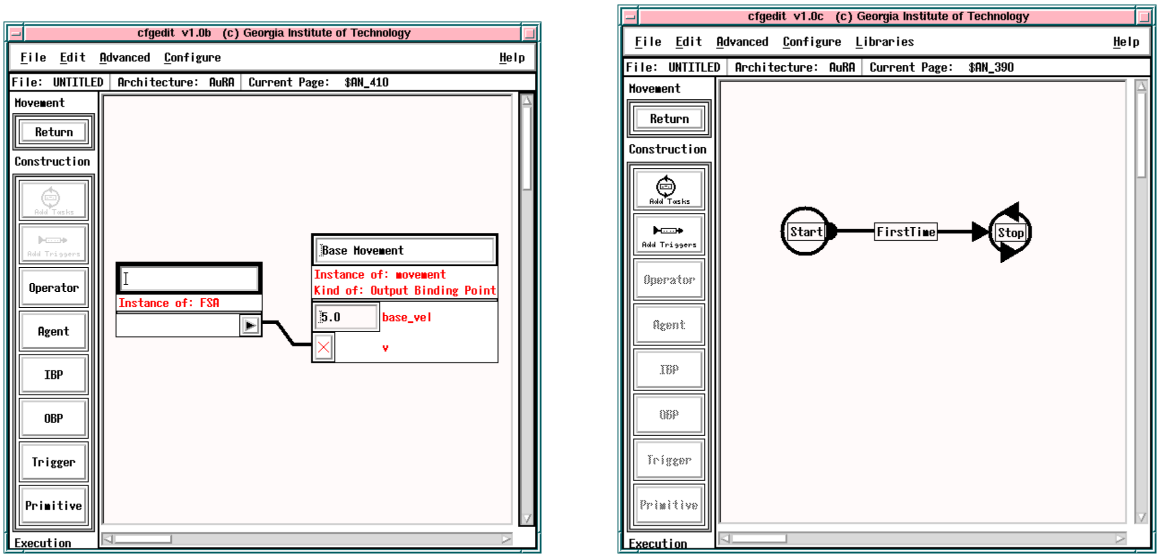
movement
pantilt

Selection

Base Movement
Instance of: movement
Kind of: Output Binding Point
5.0 base_vel

a. Output binding point menu used to choose base movement binding point.

b. Iconic representation of the output binding point placed in the workspace.



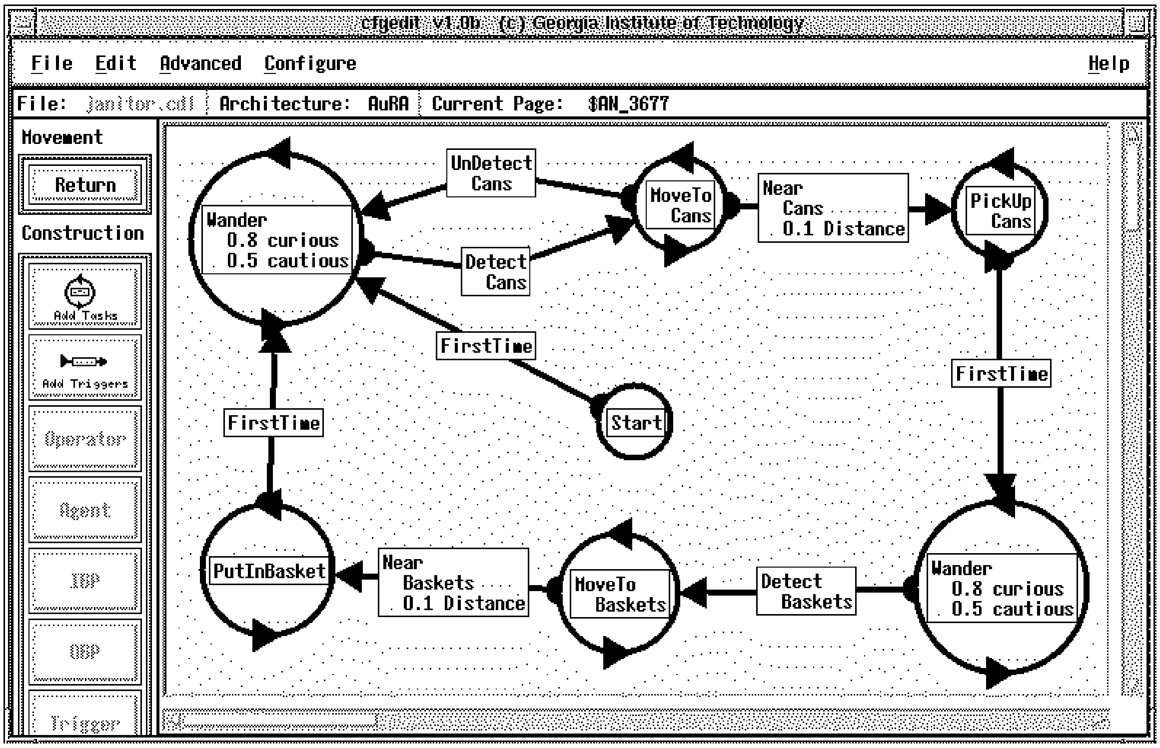
Instance of: FSA

Base Movement
Instance of: movement
Kind of: Output Binding Point
5.0 base_vel

c. Iconic representation of FSA coordination operator connected to drive wheels.

d. A simple two state FSA diagram.

Figure 5.5: Development of the trash collecting configuration.



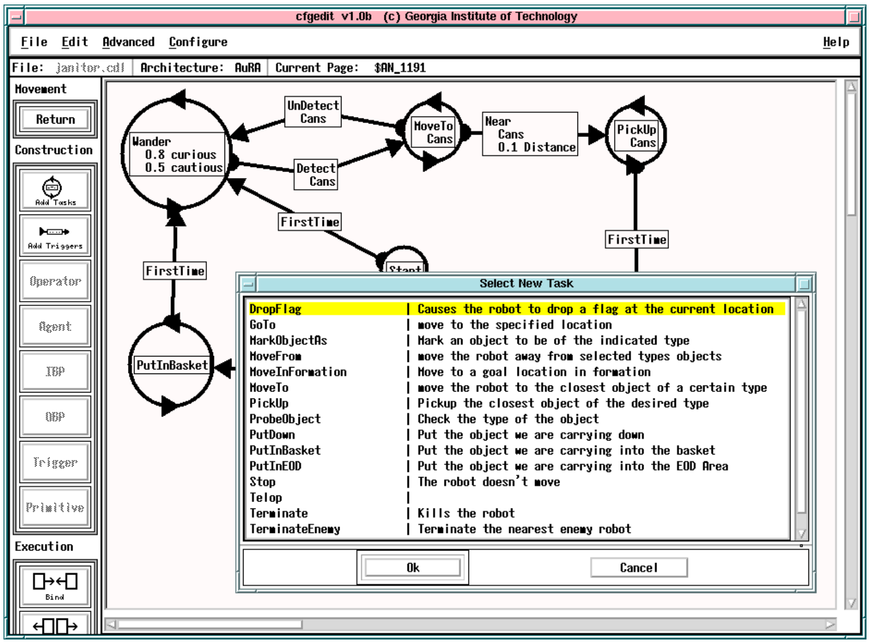
File: janitor.cdl | Architecture: AuRA | Current Page: $AN_3677

Figure 5.6: The completed FSA for a trash collecting robot.

using this popup menu. The trigger causing a particular transition is selected using a similar menu.

Clicking the middle button on an icon brings up the list of parameters for that particular assemblage or trigger, allowing parameterization. Figure 5.7b shows the parameter popup window for the *Near* assemblage. The types of objects to which the robot will react are selected by checking the box next to the perceptual class. In this case soda cans are the only object which will cause a trigger. The slider bar is then used to set how near the robot gets to the object before the transition occurs. Other assemblages and triggers will be parameterized using similar input mechanisms.

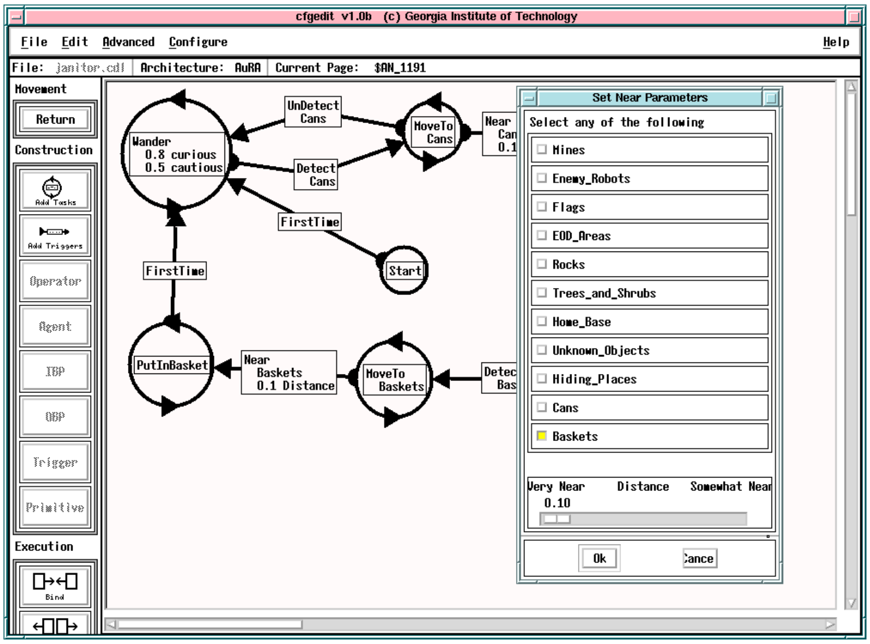
If the available assemblage or trigger choices are not sufficient, the designer can specify new constructions. These may in turn be state-based assemblages but are generally cooperative constructions. In this case, we will examine the wander assemblage. Notice it is used to both look for cans and home base. The only difference between the two states is the object being searched for, and detection of the target object is encapsulated in the termination perceptual trigger.



File: janitor.cdl | Architecture: AuRA | Current Page: $AN_1191

DropFlag	Causes the robot to drop a flag at the current location
GoTo	move to the specified location
MarkObjectAs	Mark an object to be of the indicated type
MoveFrom	move the robot away from selected types objects
MoveInFormation	Move to a goal location in formation
MoveTo	move the robot to the closest object of a certain type
PickUp	Pickup the closest object of the desired type
ProbeObject	Check the type of the object
PutDown	Put the object we are carrying down
PutInBasket	Put the object we are carrying into the basket
PutInEOD	Put the object we are carrying into the EOD Area
Stop	The robot doesn't move
Telop	
Terminate	Kills the robot
TerminateEnemy	Terminate the nearest enemy robot

a. The menu used to select new behaviors for operating states.



File: janitor.cdl | Architecture: AuRA | Current Page: $AN_1191

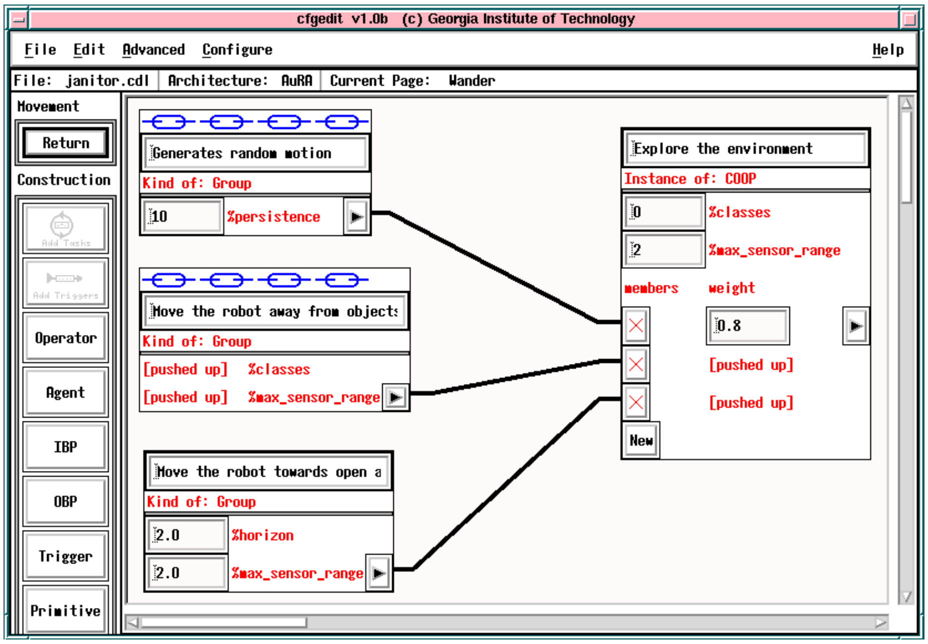
Select any of the following

☐ Mines
☐ Enemy_Robots
☐ Flags
☐ EOD_Areas
☐ Rocks
☐ Trees_and_Shrubs
☐ Home_Base
☐ Unknown_Objects
☐ Hiding_Places
☐ Cans
☑ Baskets

Very Near Distance Somewhat Near
0.10

b. Window used to select classes of objects to which the *Near* behavior is sensitive and the distance at which it triggers a transition.

Figure 5.7: Selecting and parameterizing assemblages.



File: janitor.cdl
Architecture: AuRA
Current Page: Wander

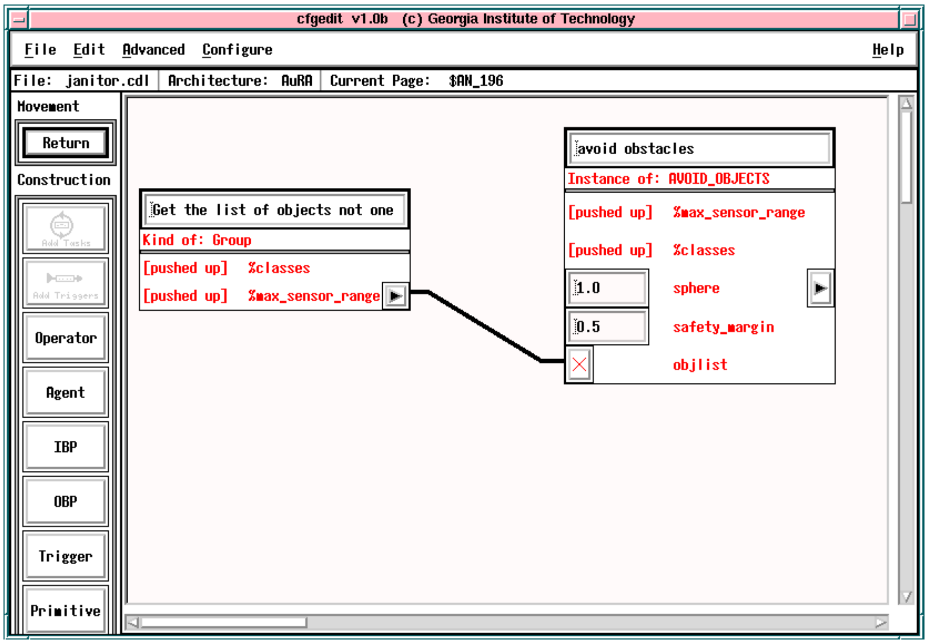
Generates random motion
Kind of: Group
10 %persistence

Move the robot away from objects
Kind of: Group
[pushed up] %classes
[pushed up] %max_sensor_range

Move the robot towards open a
Kind of: Group
2.0 %horizon
2.0 %max_sensor_range

Explore the environment
Instance of: COOP
0 %classes
2 %max_sensor_range
0.8
[pushed up]
[pushed up]

a. The behaviors and coordination operator making up the *Wander* assemblage.



File: janitor.cdl
Architecture: AuRA
Current Page: $AN_196

Get the list of objects not one
Kind of: Group
[pushed up] %classes
[pushed up] %max_sensor_range

avoid obstacles
Instance of: AVOID_OBJECTS
[pushed up] %max_sensor_range
[pushed up] %classes
1.0 sphere
0.5 safety_margin
objlist

b. The *avoid_static_obstacles* behavior coupled with a perceptual process.

Figure 5.8: The construction of the *Wander* behavior.

Figure 5.8a shows the *Wander* skill assemblage used in the *trashbot* configuration. This page is reached by shift middle clicking on either `Wander` state in the FSA. The large glyph on the right is an instance of the *Cooperative* coordination operator. This operator is responsible for creating a single output value for the group which merges contributions of the constituent behaviors. In the AuRA architecture, this operator calculates a weighted vector sum of its inputs. The three glyphs on the left of the figure are the iconic views of the three behaviors active within the wander assemblage, *noise*, *probe*, and *avoid_static_obstacles*. *Noise* induces randomness into the assemblage to increase coverage, *probe* is a free-space seeking behavior which keeps the robot from wasting large amounts of time in cluttered areas, and *avoid_static_obstacles* attempts to keep the robot a safe distance from objects in the environment. The outputs from these behaviors are weighted by the factors specified in the *Cooperative* glyph. In this case, noise has a weight of 0.8 while the weights for *probe* and *avoid_static_obstacles* are deferred by pushing them up to the next higher level. This allows these values to be specified at the FSA level.

Each of these behaviors are library functions that require no further expansion; however, they consume perceptual inputs that must be specified. In Figure 5.8b the operator has moved into the *avoid_static_obstacles* behavior to parameterize the motor behavior and connect the object detector input binding point. The *sphere* and *safety_margin* parameters set the maximum distance where obstacles still have an effect on the robot and the minimum separation allowed between the robot and obstacles, respectively. Passing closer than the *safety_margin* to an obstacle may cause the robot to convert to a "cautious" mode where it slowly moves away from the offending obstacle. This completes definition of the trash collecting configuration.

5.1.3 Datatype validation using a LISP subsystem

It is desirable to perform strict type verification on parameter values entered in *CfgEdit*. That ideal is only partially implemented in Version 1.0 of the *MissionLab* toolset. Parameters which lend themselves to presentation using slider bars, check-boxes and radio-boxes are easily added by behavior designers. Using these constrained input devices allows the editor to implicitly guarantee correctness of the data entered by users. However, some types of data are textual and require presentation of an input area allowing users to enter free-form text. Currently, *CfgEdit* does not verify such textual inputs.

A mechanism to support validation of text assigned to user-defined data types was developed as part of the Configuration Description Language. CDL suggests a LISP subsystem be included in the editor which is used solely to verify correctness of such textual inputs. When a behavior is added to the CDL libraries, the developer is also responsible for providing a LISP function to verify that a text string is a properly formed initializer.

This functionality can be added to *CfgEdit* by including a suitable LISP subsystem (*CfgEdit* is written in C++). Such an extension would allow immediate feedback to users when they enter incorrectly formed initializers. Currently, such errors are not reported until the configuration is compiled into an executable. Reporting errors at this late stage in the development process, even with good error messages, makes it extremely difficult for users to decipher the actual problem.

5.1.4 Command Description Language (CMDL)

MissionLab supports an alternative method of specifying mission sequences using structured English. A mission coordination operator is used in place of an FSA in the design and the user constructs the mission sequence using a domain-specific language with high-level primitives and mnemonic names. Currently, the mission language interpreter is resident on the operator console and communicates with the robots to invoke the correct assemblages during the mission. In future versions the interpreter may be moved onto the robots to better mesh with our schema-based control paradigm[3].

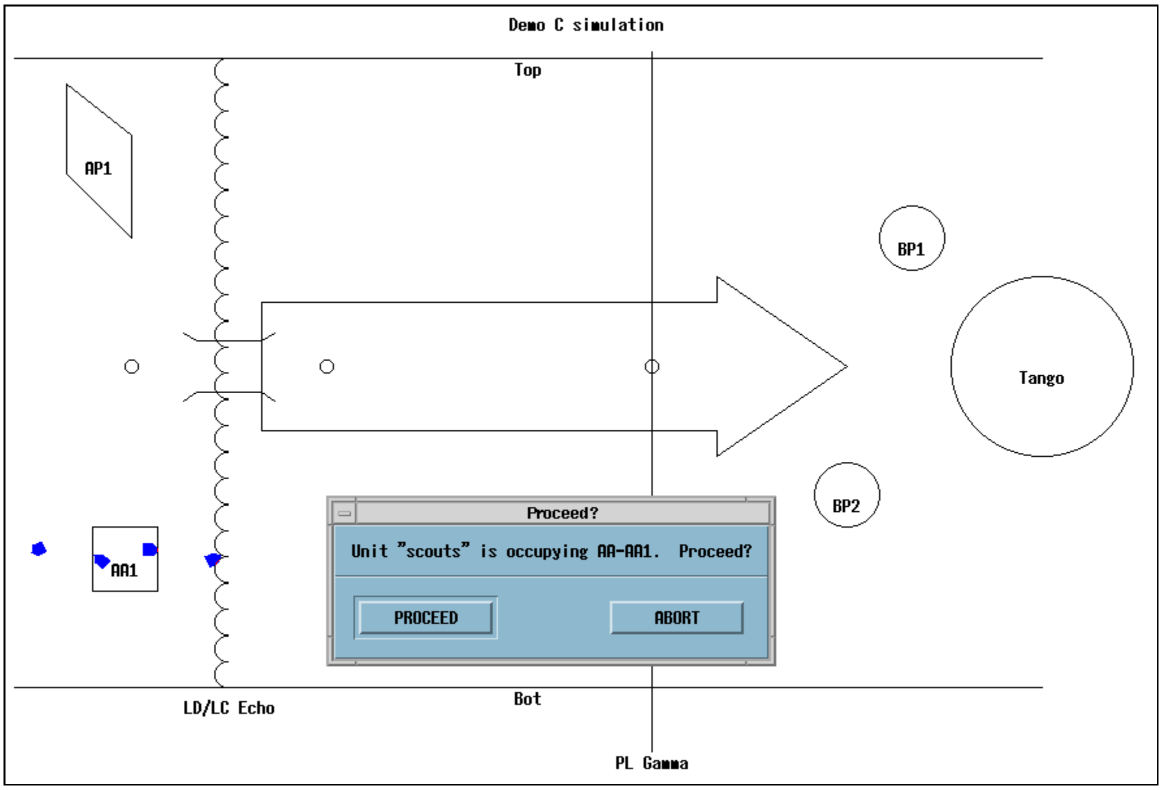
```
UNIT <scouts> (<scouts-1> ROBOT ROBOT)
              (<scouts-2> ROBOT ROBOT)
COMMAND LIST:
  0. UNIT scouts START AA-AA1 0 20
  1. UNIT scouts OCCUPY AA-AA1 FORMATION Column
  2. UNIT scouts MOVETO ATK-AP1 FORMATION Column
  3. UNIT scouts OCCUPY ATK-AP1 FORMATION Diamond
  4. UNIT scouts MOVETO PP-Charlie FORMATION Column
  5. UNIT scouts MOVETO PP-Delta1 FORMATION Column
  6. UNIT scouts MOVETO AXIS-Gamma1 FORMATION Diamond
  7. UNIT scouts-1 MOVETO ATK-BP1 AND
     UNIT scouts-2 MOVETO ATK-BP2
  8. UNIT scouts-1 OCCUPY ATK-BP1 AND
     UNIT scouts-2 OCCUPY ATK-BP2
  9. UNIT scouts MOVETO OBJ-Tango FORMATION Wedge
 10. UNIT scouts OCCUPY OBJ-Tango FORMATION Diamond
 11. UNIT scouts STOP
```

Figure 5.9: Example Mission Scenario Commands.

An example set of CMDL commands is shown in Figure 5.9. The UNIT command in the preamble (before the line containing `COMMAND LIST:`) defines the unit `scouts`. This unit is composed of two sub-units, `scouts-1` and `scouts-2`, each of which is composed of two generic robotic vehicles called `ROBOT`. The list of commands (below `COMMAND-LIST:`) is a series of steps making up the mission. The preliminary step, Step 0:

```
0. UNIT scouts START AA-AA1 0 20
```

initializes the `scouts` unit, starting execution of the robot binaries. Each robot program is also instructed where it is positioned (Assembly Area “AA1”). At this point, each robot program has no active assemblages.



Unit "scouts" is occupying AA-AA1. Proceed?

Figure 5.10: Screen snapshot showing the robot mission specified in Figure 5.9 with the robot occupying the assembly area. The four robots are the black rectangles in a horizontal line across the AA1 rectangle in the lower left.

The mission begins with Step 1:

```
1. UNIT scouts OCCUPY AA-AA1 FORMATION Column
```

This instructs the unit to occupy the starting location in column formation until it receives approval to continue. An assemblage is activated that knows how to "Occupy" and includes behaviors to maintain formations. Once the formation has been achieved in the correct location, each robot sends messages to the operator console indicating they have completed the command. The operator console pops up a "Proceed?" dialog box once all robots are in position, allowing the operator to give permission for the unit to proceed. Figure 5.10 shows the example scouting mission specified in Figure 5.9 after executing statement 1.

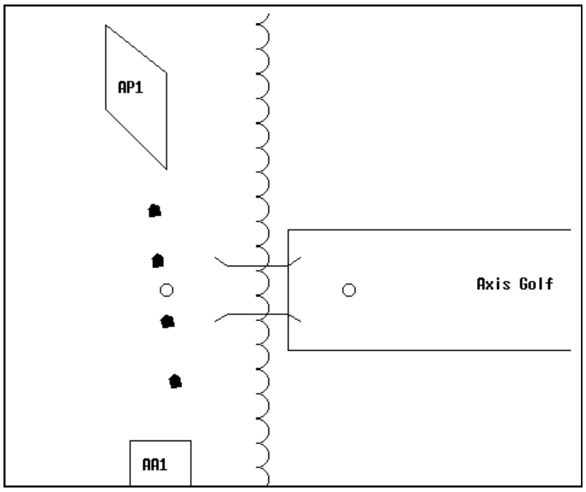


Figure 5.11: The robots have been released and are moving to position AP1.

Step 2 instructs the robots in unit `scouts` to move to position "AP1" in column formation:

```
2. UNIT scouts MOVETO ATK-AP1 FORMATION Column
```

As in the previous step, an appropriate assemblage is activated that knows how to `MOVETO` the specified location using column formation. Steps 3 through 6 move the robot through a series of way-points in various formations. The mission is shown while in mid-execution of statement 2 in Figure 5.11.

Notice that in Step 7,

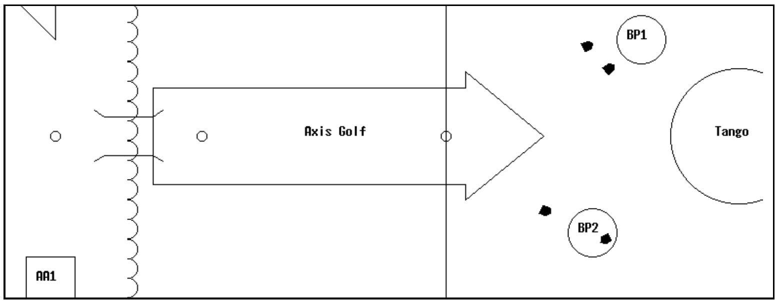


Figure 5.12: The unit has split into two sub-units so the two observation points may each be occupied.

```
 7. UNIT scouts-1 MOVETO ATK-BP1 AND
    UNIT scouts-2 MOVETO ATK-BP2
```

the unit `scouts` is divided into two sub-units `scouts-1` and `scouts-2`, each with its own command. In this case, both sub-units have `MOVETO` commands, although any command could have been given. The two robots in sub-unit `scouts-1` are moving towards BP1 and the two robots in `scouts-2` are independently moving towards BP2. Both sub-units must finish their commands before the step is complete. A screen snapshot showing the mission executing statement 7 is shown in Figure 5.12.

Once the objective has been achieved (in Step 10), the mission is terminated with Step 11:

```
11. UNIT scouts STOP
```

which instructs the robots the mission is complete and the executables are terminated. Complete definition of the command description file format can be found in the *MissionLab* manual[12].

5.2 Hardware Binding

CfgEdit supports automatic binding of configurations to robots. When the user clicks on the `bind` button, the system analyzes the configuration, matching output and input binding points to robot capabilities. It attempts to minimize the number of robots required to deploy a configuration and prompts for user input when choices are required. This vastly simplifies the binding process and promotes the creation of generic configurations.

Each architecture imposes restrictions on configurations. Some support only a small set of coordination operators or have an impoverished set of behaviors available for use by the designer while others are expansive in their scope. It may be necessary for designers to consider limits the target architectures will impose during development of the configuration to ensure that they can later be successfully bound.

Continuing with the trash collecting example, we will now bind the configuration. Clicking on the "Bind" button starts this process. First, a popup menu allows selecting the architecture to which the configuration will be bound. This determines the code generator and run-time system that will be used. In this case we will choose the AuRA architecture (the other currently supported choice is the UGV architecture).

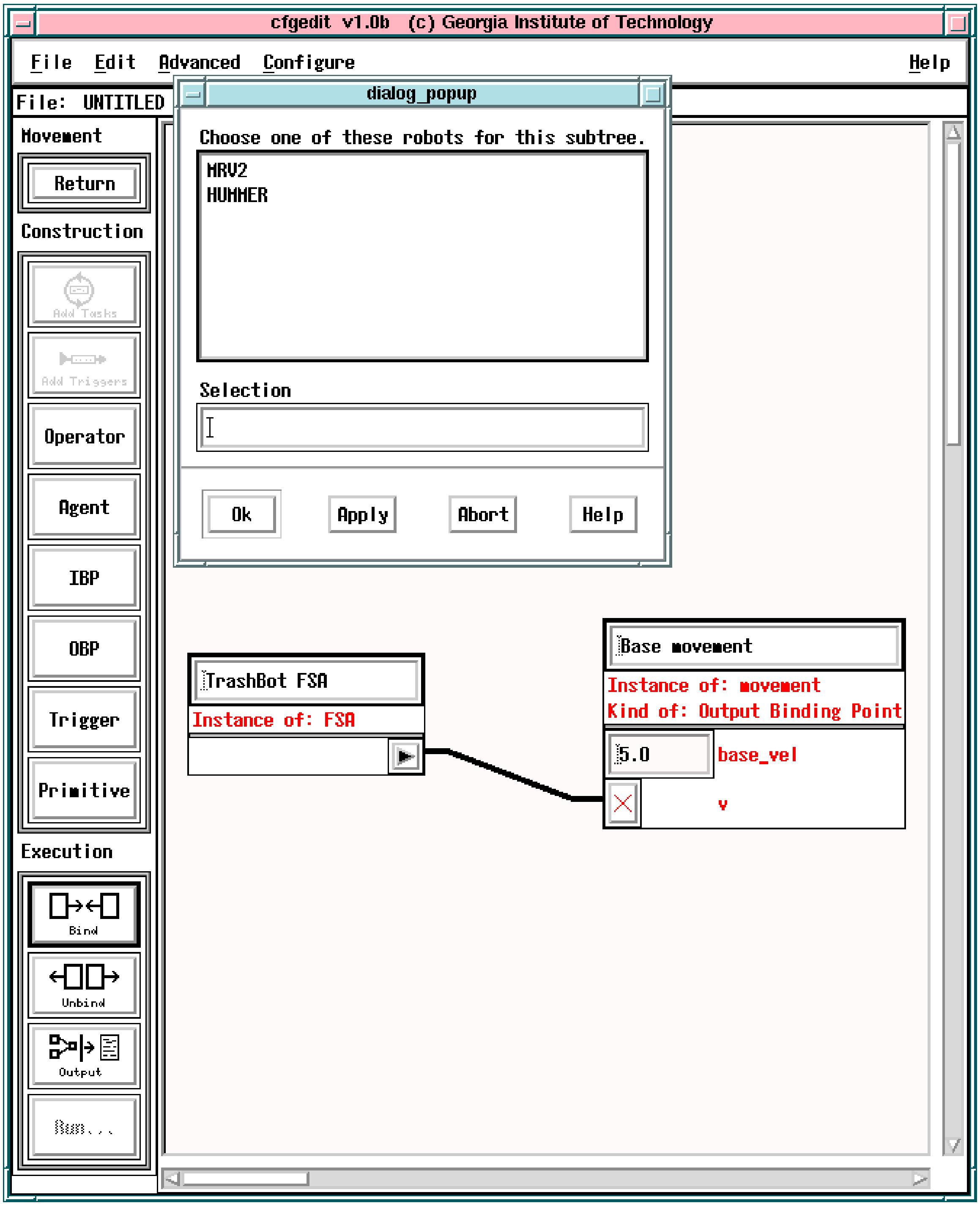
Next, the system prompts for selection of a robot to be bound to the assemblage (Figure 5.13). In this case we choose to bind this configuration to an MRV-2 Denning robot. This inserts the robot record above the displayed page, creating our recycling robot. If multiple robots are required, this robot can be replicated using the copy facilities in *cfgedit*. Figure 5.14 shows the resulting configuration with three robots specified.

This completes construction of the janitor configuration. The configuration is now ready for the code generators to create executables for each of the three robots. Once the executables are complete, the configuration can be deployed on the vehicles and executed. Although we have not shown every component of the trashbot configuration, construction of this representative subset has given an overview of the design techniques propounded and served to describe usage of the Configuration Editor. The next step in the development process is to generate a robot executable and begin evaluation of the configuration.

5.3 Code Generation

Code generators are used to convert configurations expressed using CDL into forms suitable for the targeted run-time system. Configurations bound to robots using the AuRA robot architecture are translated into a Configuration Network Language (CNL) description which is then compiled into C++ code. Robots supporting the UGV architecture require that configurations be translated into SAUSAGES (a LISP-based language) for execution.

Figure 5.15 shows a snapshot of the output logging window in *CfgEdit*. This window displays the progress as executables are built from configurations. Specifically, Figure 5.15 shows a build of a trash collecting configuration bound to three MRV-2 robots which use the AuRA architecture. The CDL compiler generates three CNL output files, one for each robot. The CNL compiler is invoked on each of these files to generate C++ code which is compiled into UNIX executables using the GNU C++ compiler (gcc).



Choose one of these robots for this subtree.

MRV2
HUMMER

Selection

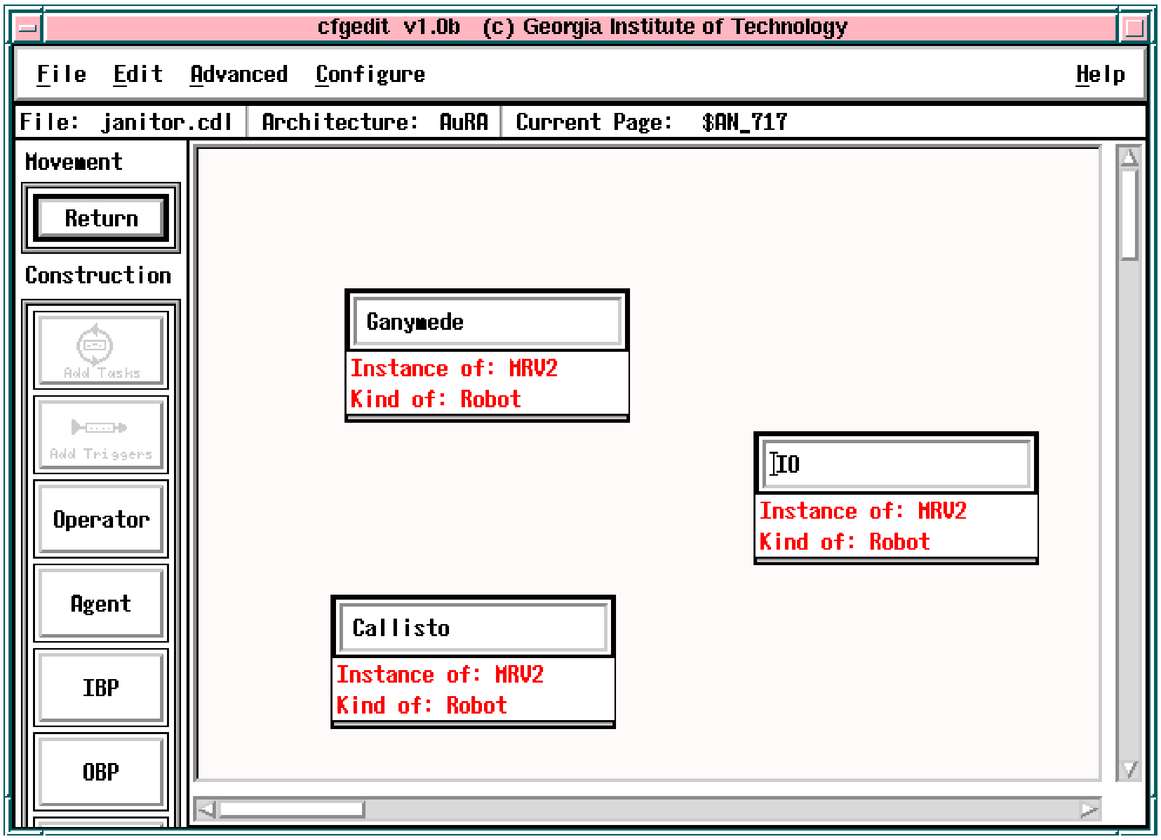
TrashBot FSA
Instance of: FSA

Base movement
Instance of: movement
Kind of: Output Binding Point

5.0 base_vel

v

Figure 5.13: The robot selection menu, presented during binding.

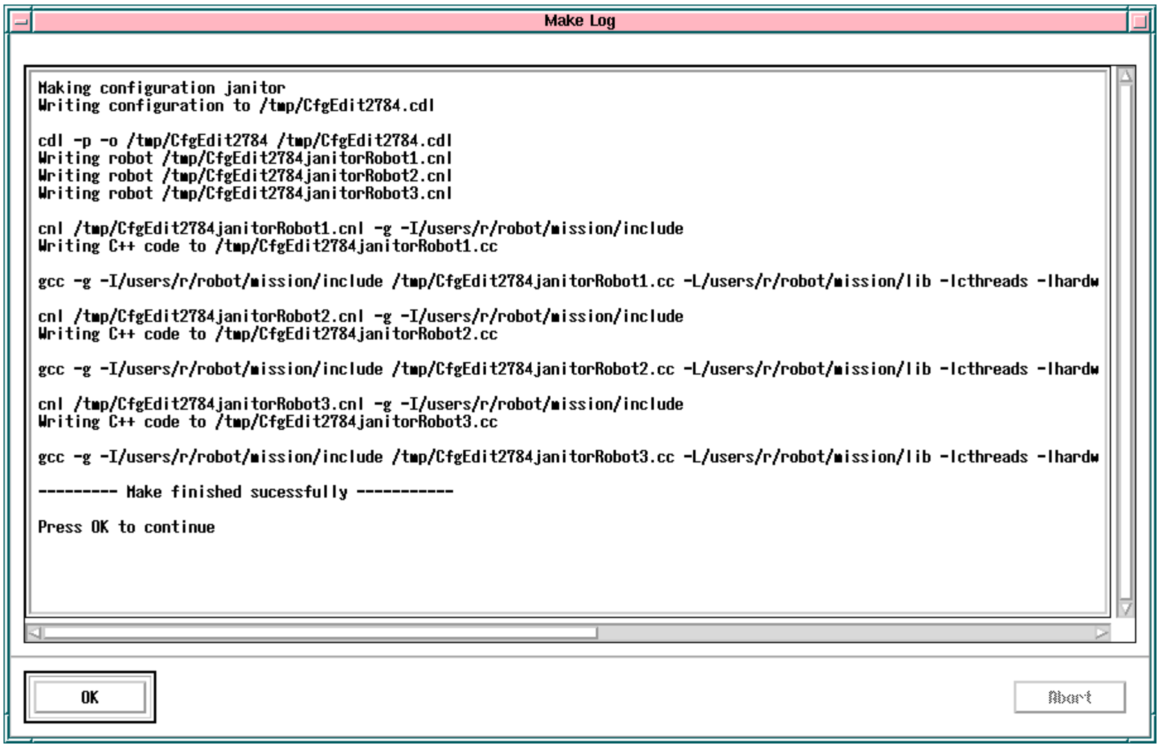


File: janitor.cdl

Architecture: AuRA

Current Page: $AN_717

Figure 5.14: *trashbot* configuration with three robots.



```
Making configuration janitor
Writing configuration to /tmp/CfgEdit2784.cdl

cdl -p -o /tmp/CfgEdit2784 /tmp/CfgEdit2784.cdl
Writing robot /tmp/CfgEdit2784janitorRobot1.cnl
Writing robot /tmp/CfgEdit2784janitorRobot2.cnl
Writing robot /tmp/CfgEdit2784janitorRobot3.cnl

cnl /tmp/CfgEdit2784janitorRobot1.cnl -g -I/users/r/robot/mission/include
Writing C++ code to /tmp/CfgEdit2784janitorRobot1.cc

gcc -g -I/users/r/robot/mission/include /tmp/CfgEdit2784janitorRobot1.cc -L/users/r/robot/mission/lib -lcthreads -lhardw

cnl /tmp/CfgEdit2784janitorRobot2.cnl -g -I/users/r/robot/mission/include
Writing C++ code to /tmp/CfgEdit2784janitorRobot2.cc

gcc -g -I/users/r/robot/mission/include /tmp/CfgEdit2784janitorRobot2.cc -L/users/r/robot/mission/lib -lcthreads -lhardw

cnl /tmp/CfgEdit2784janitorRobot3.cnl -g -I/users/r/robot/mission/include
Writing C++ code to /tmp/CfgEdit2784janitorRobot3.cc

gcc -g -I/users/r/robot/mission/include /tmp/CfgEdit2784janitorRobot3.cc -L/users/r/robot/mission/lib -lcthreads -lhardw

--------- Make finished sucessfully -----------

Press OK to continue
```

Figure 5.15: Logging window showing progress of *trashbot* configuration build (The configuration was bound to MRV-2 robots). Notice that three robot executables were generated.

5.3.1 Supported Robot Architectures

MissionLab uses the term **robot architecture** to describe a robot run-time system. Recall that CDL configurations describe when behaviors are instantiated, and how they are parameterized and coordinated. They do not specify how behavioral primitives are implemented or how run-time communication and scheduling issues are to be managed. The set of available primitives and the robot operating system which schedules behavior execution and manages data movement is in the domain of the robot architecture.

For *MissionLab* to support a particular robot architecture means it is available as a target in the binding process. This implies a suitable code generator is present to convert the CDL descriptions into a form usable by the architecture. Each architecture also has a CDL file to provide interface descriptions of each available primitive component to allow its use in the graphic editor *CfgEdit*. These descriptions are used to display the choices available to configuration designers when targeting a specific architecture.

The currently supported architectures are AuRA and UGV. The Configuration Network Language is used in the AuRA architecture and the SAUSAGES scripts are used in the ARPA UGV architecture. They are both data-flow languages which have similar expressive power. CNL is converted into C++ code with a lightweight thread scheduler while SAUSAGES is directly interpreted by a LISP function. Both will now be described.

5.3.2 The Configuration Network Language (CNL)

When a configuration is bound to the AuRA architecture, the CDL compiler generates a Configuration Network Language (CNL)[51] specification of the configuration as its output. CNL is a hybrid data-flow language[43] using large grain parallelism, where the atomic units are arbitrary C++ functions. CNL adds data-flow extensions to C++ which eliminate the need for explicit communication code. A compiled extension to C++ was chosen to allow verification and meaningful error messages to assist casual C++ programmers in constructing behaviors. The separation of the code generator from the CDL compiler permits incremental development and testing of the design tools as well as simplifying retargeting.

The use of communicating processing elements is similar to the Robot Schemas (RS)[50] architecture, based on the port automata model. The major differences are that RS uses synchronous communication and CNL is asynchronous to support multiprocessing; and RS is an abstract language while a CNL compiler has been developed. Both use the notion of functions processing data arriving at input ports to compute an output value, which is then available for use as inputs in other functions.

CNL provides a separation between procedure implementations and specification of the data-flow graph's topography. CNL encourages this separation by allowing the procedures to be coded, tested and archived in standard Unix libraries. Instances of these procedures are specified in a separate step, thereby creating the data-flow network. The output of the CNL compiler is a C++ file created by merging the user's procedures and the compiler's data movement code specified by the node definitions. This output file can be compiled using any suitable ANSI C++ compiler targeted for the desired machine architecture. Using a standard programming language for the procedures simplifies converting existing code to CNL.

Figure 5.16 presents the CNL procedure definition for the *avoid_static_obstacles* behavior used in the trash collecting configuration. There are three places that C++ code can occur in CNL: Bracketed by an `init/iend` pair, bracketed by a `header/body` pair within a procedure definition, or bracketed by a `body/pend` pair within a procedure definition. Code within `init` blocks is not specific to a particular procedure. Code within `header` blocks is emitted before the body loop and therefore executes once on instantiation. Code within `body/pend` blocks is surrounded with a while loop and executes once for each new set of input parameters. The predefined output parameter for the procedure is named `output`.

Figure 5.17 shows a portion of the CNL code generated from the *trashbot* configuration. A CNL configuration can be viewed as a directed graph, where nodes are threads of execution and edges indicate data-flow connections between producer nodes and consumer nodes. Each node in the configuration is an instantiation of a C++ function, forked as a lightweight thread using the C-Threads package[71] developed at Georgia Tech. UNIX processes are examples of heavyweight threads which use the operating system for scheduling. Lightweight threads are generally non-preemptive and scheduled by code linked into the user's program. All lightweight threads execute in the same address space and can share global variables. The advantage of lightweight threads is that a task switch takes place *much* faster than between heavyweight threads, allowing large scale parallelism. Current robot configurations are using around 100 threads with little overhead, while that many UNIX processes is not feasible. Code for thread control and communication synchronization is explicitly generated by the CNL compiler and need not be specified by the user.

5.3.3 SAUSAGES

When the configuration is bound to the UGV architecture, the SAUSAGES code generator is used. The System for AUtonomous Specification, Acquisition, Generation, and Execution of Schemata (SAUSAGES)[27, 26] is a behavior configuration specification language as well as run-time executive. At run time, SAUSAGES executes the scripts by instantiating behaviors, monitoring for failures, and interacting

```
procedure Vector AVOID_STATIC_OBSTACLES with
    double sphere;
    double safety_margin;
    obs_array readings;
header
body
    VECTOR_CLEAR(output);

    for(int i=0; i<readings.size; i++)
    {
        double c_to_c_dist = len_2d(readings.val[i].center);
        double radius = readings.val[i].r + safety_margin;
        double mag = 0;

        if (c_to_c_dist ≤ radius )
        {
            // if within safety margin generate big vector
            mag = INFINITY;
        }
        else if (c_to_c_dist ≤ radius + sphere )
        {
            // generate fraction (0...1) how far are in linear zone.
            mag = (sphere - (c_to_c_dist - radius)) / sphere;
        }
        // otherwise, outside obstacle's sphere of influence, so ignore it

        if (mag ≠ 0)
        {
            // create a unit vector along the direction of repulsion
            Vector repuls;
            repuls.x = -readings.val[i].center.x;
            repuls.y = -readings.val[i].center.y;
            unit_2d(repuls);

            // Set its strength to the magnitude selected
            mult_2d(repuls, mag);

            // Add it to the running sum
            plus_2d(output, repuls);
        }
    }
pend
```

Figure 5.16: CNL code for avoid static obstacles behavior.

```
// Define the node named $AN_3582 as an instance of the procedure IS_AN_OBJECT.//
// The procedure requires a single input with the list of objects to check. //
node $AN_3582 is IS_AN_OBJECT with
  object_list = $AN_3577;
nend

// Pass only objects of class ``can'' //
node $AN_3577 is FILTER_OBJECTS_BY_CLASS with
  remove_these = {false};
  classes = {Cans};
  full_list = $AN_3573;
nend

// Get list of objects from the sensor //
node $AN_3573 is DETECT_OBJECTS with
  max_sensor_range = {0.1};
nend

// Compute the relative location of the object from the robot //
node $AN_3605 is OBJECT_LOCATION with
  object = $AN_3600;
nend

// select only the closest object //
node $AN_3600 is CLOSEST_OBJECT with
  object_list = $AN_3594;
nend

// Get list of objects from the sensor //
node $AN_3590 is DETECT_OBJECTS with
  max_sensor_range = {0.1};
nend

// |Is there one of the objects? //
node $AN_3625 is IS_AN_OBJECT with
  object_list = $AN_3620;
nend
```

Figure 5.17: Portion of CNL code generated for *trashbot* configuration. Each of the node definitions instantiates a thread of execution using the named procedure and input links.

with higher-level processes. A SAUSAGES program is a graph structure where the behaviors are operations which move between nodes in the graph.

SAUSAGES is a LISP-based script language tailored for specifying sequences of behaviors for large autonomous vehicles. There are currently four available skill assemblages; off road move to goal, follow road, pause, and teleoperate. Due to these limitations, our *trashbot* configuration cannot be mapped onto this architecture. Figure 5.18 presents SAUSAGES code for a simple configuration which traverses through two waypoints, allows the operator to teleoperate the vehicle, and then returns the robot to the starting area.

5.4 *MissionLab* Maintenance

The *MissionLab* system requires that several data files be kept up-to-date as the toolset evolves at a site. The file `.cfgeditrc` is used to configure software functionality, user privileges, and locations of library files on the system. The library files are used to describe the primitive behaviors, which architectures are supported, and what robots are available.

5.4.1 The .cfgeditrc Resource File

Figure 5.19 shows an example `.cfgeditrc` file. Each of the executables making up the *MissionLab* toolset attempt to load this file on startup. They begin by looking in the current directory, then the user's home directory, and finally search the directories listed in the path statement. This file configures several operating characteristics of the system such as user privileges, whether backup files are created on writes, how values are presented in the graphic editor, *etc.* Directories where libraries and source files reside is also listed. Architecture specific sections are used by the code generators to configure how they generate executables. A section devoted to configuring the actual robots allows users to pass architecture-specific configuration information to the robots on startup.

5.4.2 Adding New AuRA Primitives

The addition of a new CNL primitive such as the one shown in Figure 5.16 requires some care. First, the behavior itself must be coded in CNL and debugged to verify that it functions as desired. The source file containing the CNL primitive is structured with one procedure per file and named with the name of the procedure and a `.cnl` extension.

For example, the `AVOID_STATIC_OBSTACLES` procedure in Figure 5.16 should be saved in the file `AVOID_STATIC_OBSTACLES.cnl` in a directory with similar cnl files.

```
(defplan the-plan ()
  (sequence
    (link user-wait
          (plan-id 0)
          (position '(1425.0 675.0))
          (message "Waiting for proceed message")
    )
    (link xcountry
          (plan-id 1)
          (initial-speed 40.0)
          (points (vector
            '(1425.0 675.0)
            '(1050.000000 525.000000)))
    )
    (link xcountry
          (plan-id 2)
          (initial-speed 40.0)
          (points (vector
            '(1050.000000 525.000000)
            '(600.000000 525.000000)))
    )
    (link teleoperate
          (plan-id 3)
          (initial-speed 40.0)
          (start '(600.000000 525.000000))
          (end '(600.000000 525.000000))
    )
    (link xcountry
          (plan-id 4)
          (initial-speed 40.0)
          (points (vector
            '(600.000000 525.000000)
            '(1350.000000 600.000000)))
    )
    (link user-wait
          (plan-id 5)
          (position '(1350.000000 600.000000))
          (message "Waiting for proceed message")
    )))
```

Figure 5.18: SAUSAGES code generated by CfgEdit for a simple mission which moves the robot through two waypoints, allows the operator to teleoperate the vehicle, and then returns it back to the starting area.

```
# Make backup CDL files from the editor on writes (true or false).
backup_files = True

# Show the values of the slider bars instead of the symbolic names
ShowSliderValues = True

# Set the capabilities of the user.
user_privileges = Execute, Modify, Edit, Create, Library, RealRobots

# List architectures here to restrict primitives shown to only those occurring in all listed.
#restrict_names = UGV

# List of comma separated directories and root names of CDL libraries to load.
# Will try to load xxx.gen, xxx.AuRA, and xxx.UGV
CDL_libraries = /users/mission/lib/default, /users/mission/lib/agents

# Optional: Configuration to load as the empty configuration.
DefaultConfiguration = /users/r/robot/mission/lib/FSA.cdl

# Where to find the map overlays
MapOverlays = /users/r/robot/mission/overlays

# Directory to write the event logs.  Logging is off when this is empty.
# EventLogDir = .

# *********** CNL Architecture configuration ********************
# List of comma and white space separated directories for link libraries.
lib_paths = /users/r/robot/mission/lib

# Directories and root names of the # CNL source files and libraries.
# Will try to include xxx.inc as the cnl header file and link libxxx.a
# The editor will look for yyy.cnl in these locations to show AuRA primitive yyy
CNL_libraries = /users/r/robot/mission/lib/cnl

# directories to look in for CNL source files to display in the editor.
CNL_sources = /users/r/robot/mission/src/libcnl

# cflags parm passed to C++ compiler
cflags = -g, -I/users/r/robot/mission/include

# ldflags parm passed to C++ compiler. CNL_LIBS is replaced with library list
ldflags = -L/users/r/robot/mission/lib, -lcthreads, -lhardware_drivers,
              CNL_LIBS, -lhardware_drivers, -lipt, -lg++, -lstdc++, -lm

# *********** Real AuRA robot configuration flags ********************
#The list of real AuRA-based robots we know about
robots = ren, stimpy, george

# Any misc robot settings that will be dumped to the script file
MiscRobotSettings = "set show-trails on", "set scale-robots on"

# Any list of strings attached to the robot name will be appended to the
# startup parameters for that robot.  The robot names are case sensitive.
ren    = "ignore_sensor23 = 1", "ignore_sensor1 = 1", "tty_num= 1"
stimpy = "ignore_sensor0 = 1", "ignore_sensor3 = 1", "tty_num= 0"
george = "robot_type = DRV1", "tty_num= 2"
```

Figure 5.19: Example `.cfgeditrc` file.

The CNL compiler is used to compile the procedure into C++ code, which is compiled into an object file using a C++ compiler (*e.g.*, gcc). The object files for each of the primitives are then archived into a library file and left in the same directory.

```
external procedure Vector AVOID_STATIC_OBSTACLES with
    double sphere;
    double safety_margin;
    obs_array readings;
pend
```

Figure 5.20: CNL prototype for avoid static obstacles behavior.

The library should be named `libxxx.a` where `xxx` is the descriptive name for the collection of primitives. A CNL header file must also exist in the directory with the name `xxx.inc` which provides CNL prototypes for each of the procedures in the library. This allows the separate compilation of CNL procedures while ensuring tight type-checking between input and output parameters. An example prototype for the `AVOID_STATIC_OBSTACLES` procedure is shown in Figure 5.20. The directory path is appended to the `CNL_libraries` list in the appropriate `.cfgeditrc` file so *MissionLab* can find the files. For example, if we make a library of navigation primitives in the directory `/behaviors/navigation` and call it `nav` then the library file will be called `libnav.a`, the include file `nav.inc` and the directory path assigned to the `CNL_libraries` variable will be:

```
CNL_libraries = /behaviors/navigation/nav
```

The graphic editor will try to load cnl files from the `/behaviors/navigation` directory when users attempt to display implementations of CNL primitives. Multiple library specifications assigned to variables in the `.cfgeditrc` file are separated by semicolon ";" characters.

5.4.3 Adding New CDL Primitives

The CDL-based tools use library files to describe the components available to users. Components can be primitives or assemblages constructed from other components. The `CDL_libraries` variable in the `.cfgeditrc` file lists the CDL libraries to load. Given the following `.cfgeditrc` file fragment

```
CDL_libraries = /robot/lib/navigation
```

the system will attempt to load the libraries `navigation.gen`, `navigation.AuRA`, and `navigation.UGV` from the directory `/robot/lib`.

```
defAgent displacement AVOID_STATIC_OBSTACLES(
    const double sphere,
    const double safety_margin,
    object_locations readings);
```

Figure 5.21: Generic CDL definition for avoid static obstacles behavior.

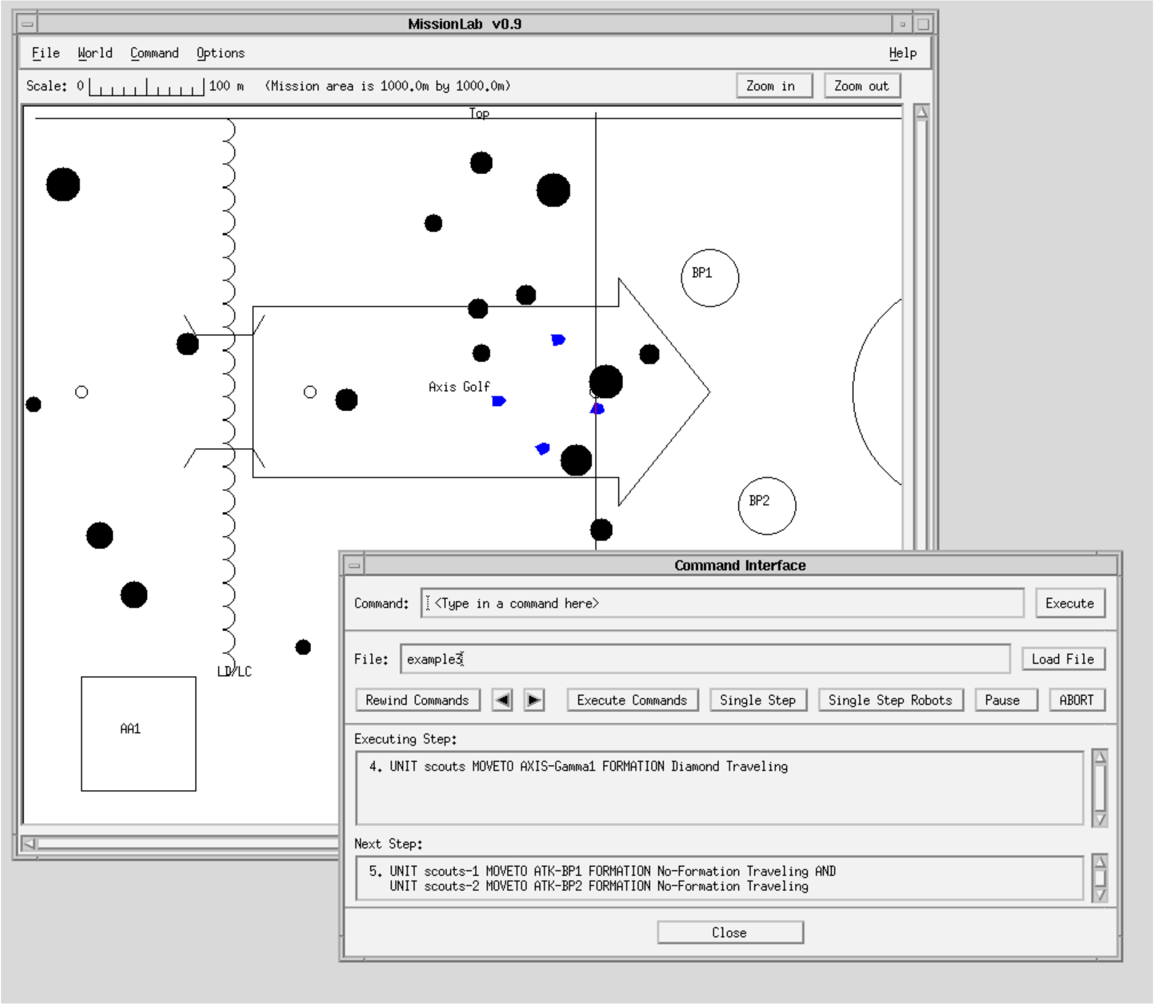
```
defAgent[AuRA] binds AVOID_STATIC_OBSTACLES Vector AVOID_STATIC_OBSTACLES(
    const double sphere = 3.0,
    const double safety_margin = 0.5,
    object_locations readings);
```

Figure 5.22: AuRA specific CDL definition for avoid static obstacles behavior.

Figure 5.21 shows a fragment of `navigation.gen` which defines a generic version of the CDL primitive `AVOID_STATIC_OBSTACLES`. Figure 5.22 shows a fragment of `navigation.AuRA` which defines the version of the `AVOID_STATIC_OBSTACLES` primitive specific to the AuRA architecture. With these two definitions in place, users are free to construct generic and AuRA-based configurations using the primitive `AVOID_STATIC_OBSTACLES` behavior.

5.5 MissionLab Command Console for AuRA

MissionLab includes an operator console supporting AuRA-based robots used to execute missions using simulated and/or real vehicles. The operator display shows the simulation environment, the locations of all simulated robots, and the reported positions of any real robots. Figure 5.23 shows a screen snapshot of the system simulating a scouting mission.



Command: <Type in a command here>

File: example3

Executing Step:

```
4. UNIT scouts MOVETO AXIS-Gamma1 FORMATION Diamond Traveling
```

Next Step:

```
5. UNIT scouts-1 MOVETO ATK-BP1 FORMATION No-Formation Traveling AND
   UNIT scouts-2 MOVETO ATK-BP2 FORMATION No-Formation Traveling
```

Figure 5.23: Example scenario being executed in *MissionLab*. The solid circles represent obstacles, the various control measures shown in the overlay allow the operator to symbolically specify and constrain the mission.

The main display area shows the robots in a diamond formation starting to cross the phase line. The solid black circles represent obstacles within the simulated environment. The command interface in the lower right of Figure 5.23 allows the operator to monitor and control execution of the mission. For more detail on the operation of *MissionLab*, see [12].

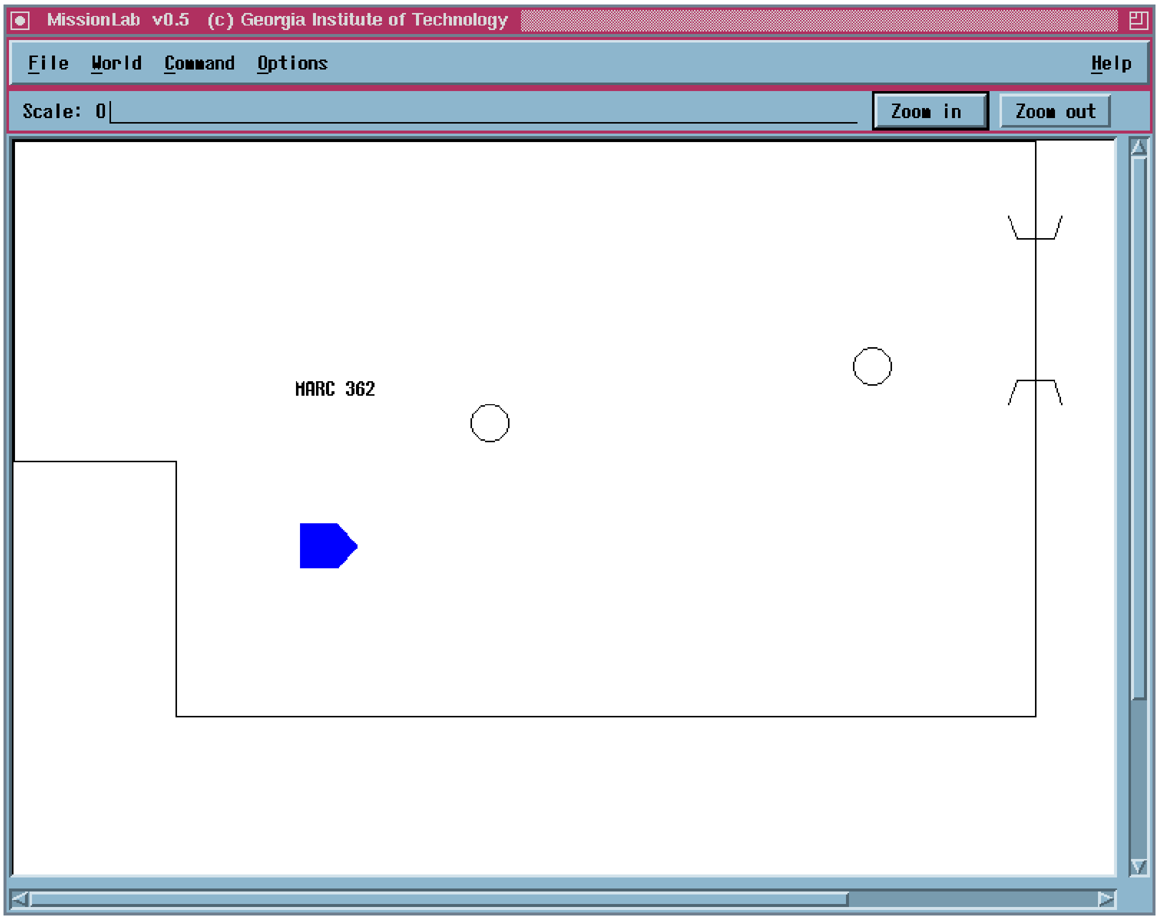


Figure 5.24: *MissionLab* with the Mobile Robot Lab overlay loaded.

The overlay files are constructed using standard text editors. Figure 5.24 shows *MissionLab* with an overlay representing the Georgia Tech Mobile Robot Lab. The robot is shown in its starting location, in the lower left. The overlay file specifying this environment is shown in Figure 5.25. The SCENARIO command names the environment. The SITE command provides a description of the environment. The

```
SCENARIO "example1"
SITE "Robot Lab, MARC 362"
CONTROL MEASURES:

Boundary "MARC 362" 0 0  10.7 0
   10.7 6.1  1.7 6.1  1.7 3.4
   0 3.4  0 0
Gap Door 10.5 1.8    10.9 1.8  1.5
PP DoorWay 9.0 2.4 0.4
PP Middle 5.0 3.0 0.4
```

Figure 5.25: The definition file specifying the overlay shown in Figure 5.24.

Boundary marks the walls of the Mobile Robot Lab (units in Meters). The Gap specifies the door to the lab. Two passage points (PP) (shown as circles in Figure 5.24) were chosen arbitrarily to use as targets for MOVETO commands in missions. Many other types of overlay symbols are available and are described in the *MissionLab* manual[12].

5.6 AuRA simulator

The *MissionLab* toolset includes a multiagent simulation system which supports robots using the AuRA architecture. The simulator is structured in a client/server arrangement and currently executes in the same process as the operator console. Each robot (simulated or real) is a separate Unix process which communicates with the simulation server using the IPT communications package developed at Carnegie-Mellon University. This allows running robots and the simulation server on different computers with only the names of the machines required for initialization.

Figure 5.26 shows the `Janitor` configuration executing in simulation using the AuRA run-time architecture. Within the main display area robots, obstacles, and other features are visible. The cans have all been gathered and returned to the circle labeled `basket`. The shaded and solid circles of various sizes represent simulated obstacles within the arena (vegetation and rocks, respectively). The three robots are actively gathering trash and the paths they have taken are shown as trails. For more details on the *MissionLab* simulation system, see [12].

The simulation system currently models only the vehicle kinematics. The vehicle maximum speed and steer angle is also limited in some cases. No attempt to introduce dynamics into the simulator has been undertaken. Currently, Denning MRV-2 robots are modeled as holonomic robots. Our HUMMER robot (a traditional car-type vehicle) is simulated by limiting the maximum steer angle and setting the size of the vehicle appropriately.

Using the client/server model allows the user to mix simulated and real robots within a single run. Robot executables driving simulated robots communicate with the simulation server to request movement of their robot and to get simulated sensor readings. Executables controlling real robots report their position to the simulator as they move and request simulated sensor readings of where other robots are located from the server. This allows the system to report positions of both real and simulated vehicles to the executables. This capability of mixing simulated and real robots on a single run allows testing the benefits of additional vehicles as well as the scalability of particular configurations without having to acquire additional hardware.

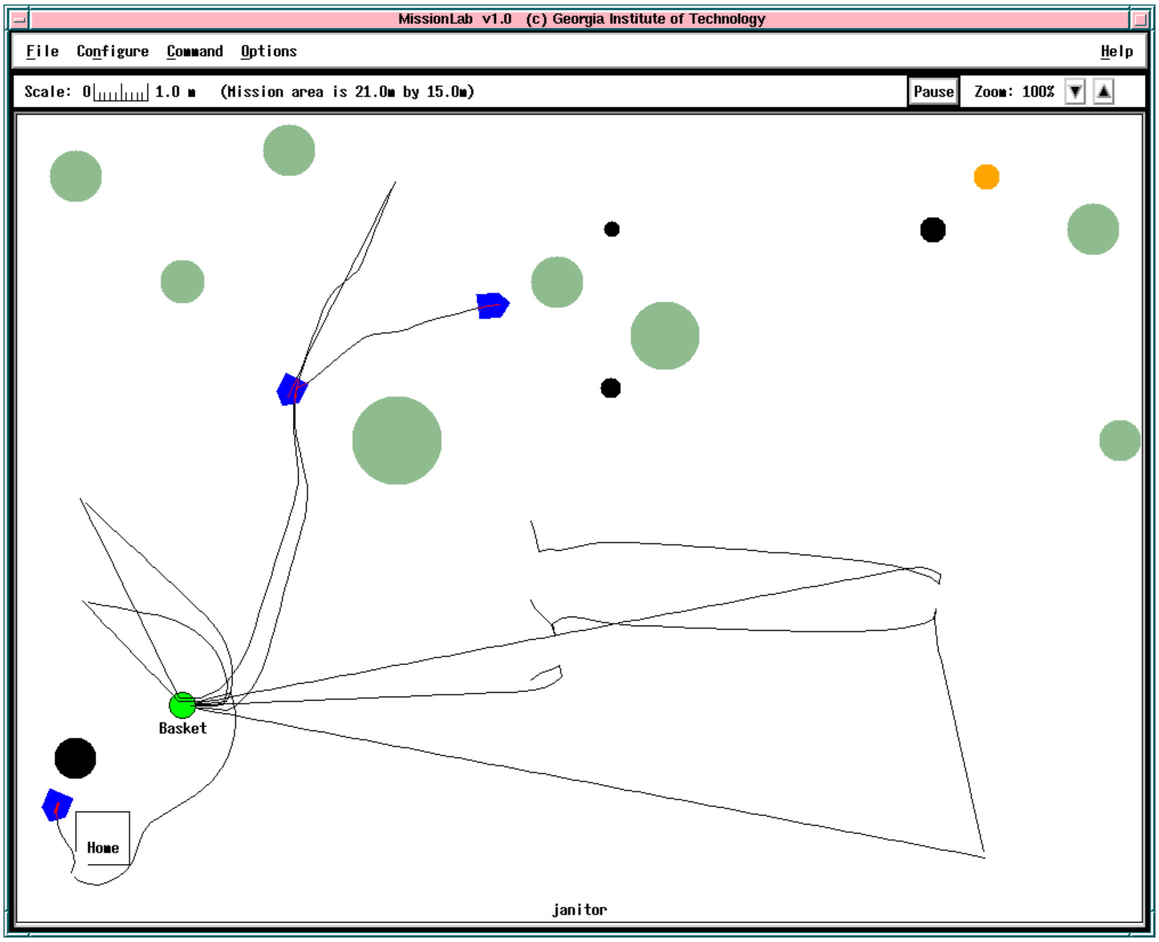


Figure 5.26: The *trashbot* configuration executing in simulation. The cans have all been gathered and returned to the circle labeled `basket`. The trails show the paths the robots took completing the mission. The shaded and solid circles of various sizes represent simulated obstacles within the arena (vegetation and rocks, respectively). The robots treated both classes of obstacles the same during this mission.

5.7 SAUSAGES simulator

A single robot simulation system using the SAUSAGES run-time system was provided by Jay Gowdy and Carnegie Mellon University. This allowed evaluating SAUSAGES code generated by *CfgEdit* on an independent system. Figure 5.27 is a screen snapshot of the SAUSAGES simulator after execution of a simple mission created with CfgEdit. The robot does not leave trails in this simulator, although the waypoints are connected by straight lines to show the projected route.

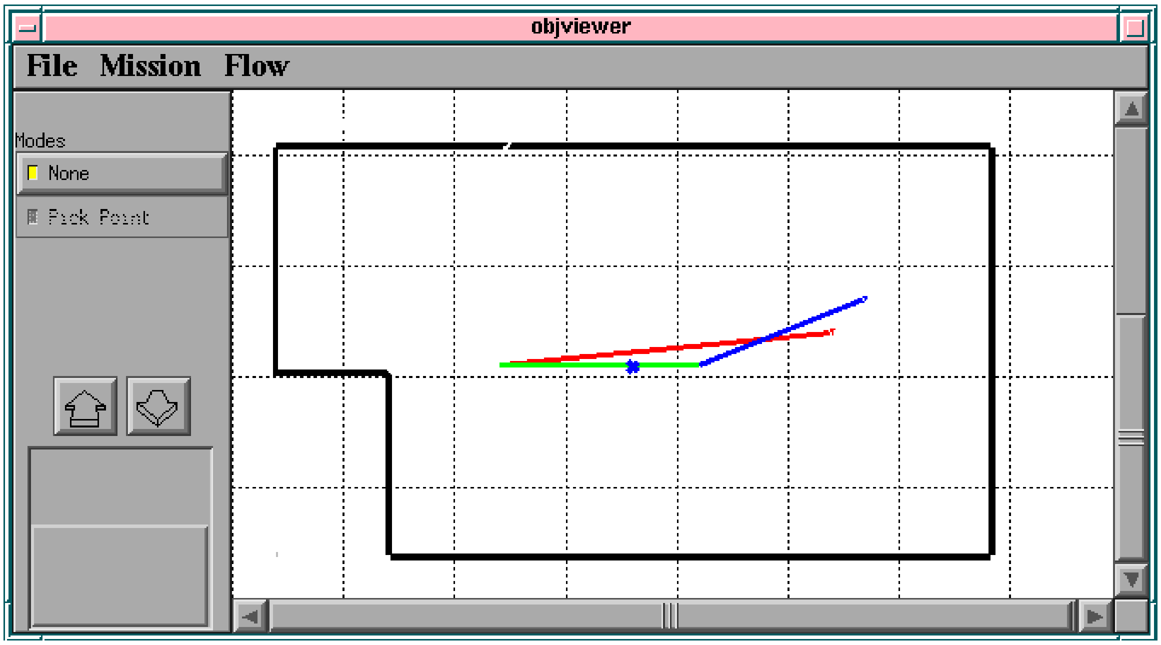


Figure 5.27: Example screen snapshot of SAUSAGES simulation display.

The simulator is constructed in LISP and the SAUSAGES code is loaded as a LISP source file. The simulator then begins executing the plan while simulating movement of a HUMMER robot. Several types of terrain are available when constructing the environment, including tar and gravel roads, rivers, and grasslands. There are currently three available behaviors in the simulator: move to goal, follow road, and teleoperate.

5.8 Demonstrations of Functionality

Several demonstrations were performed to highlight important facets and capabilities of the *MissionLab* toolset. The first demonstration in Section 5.8.1 shows how new components can be automatically added to the library file from within CfgEdit. Section 5.8.2 shows how complex assemblages, including FSA coordination, can be created and archived as library components.

Section 5.8.3 documents creation of a generic configuration, which is then bound to a Denning MRV-2 robot and executed using the *MissionLab* simulator. The mission is repeated driving a real Denning robot. Finally, the configuration is re-bound to a UGV robot and SAUSAGES code generated which is demonstrated using the CMU SAUSAGES simulator.

A four robot scouting mission was conducted in simulation to highlight the multiagent capabilities of the *MissionLab* system. Section 5.8.4 shows the robots moving through a series of areas in various formations while demonstrating a military-style scouting mission.

Finally, in Section 5.8.5, two Denning MRV-2 robots were used to demonstrate the multiagent capabilities of the system while driving real robots. A configuration was constructed to cause the robots to move through the Mobile Robot lab in two different formations.

5.8.1 Adding a new component to a library

The `Library` option on the CfgEdit menu-bar allows manipulating CDL component libraries from within the editor. The operator can add new components to libraries, delete components from a library, and import components into the workspace for modification. The process of adding a new component to a library will be presented in this section to introduce these library capabilities.

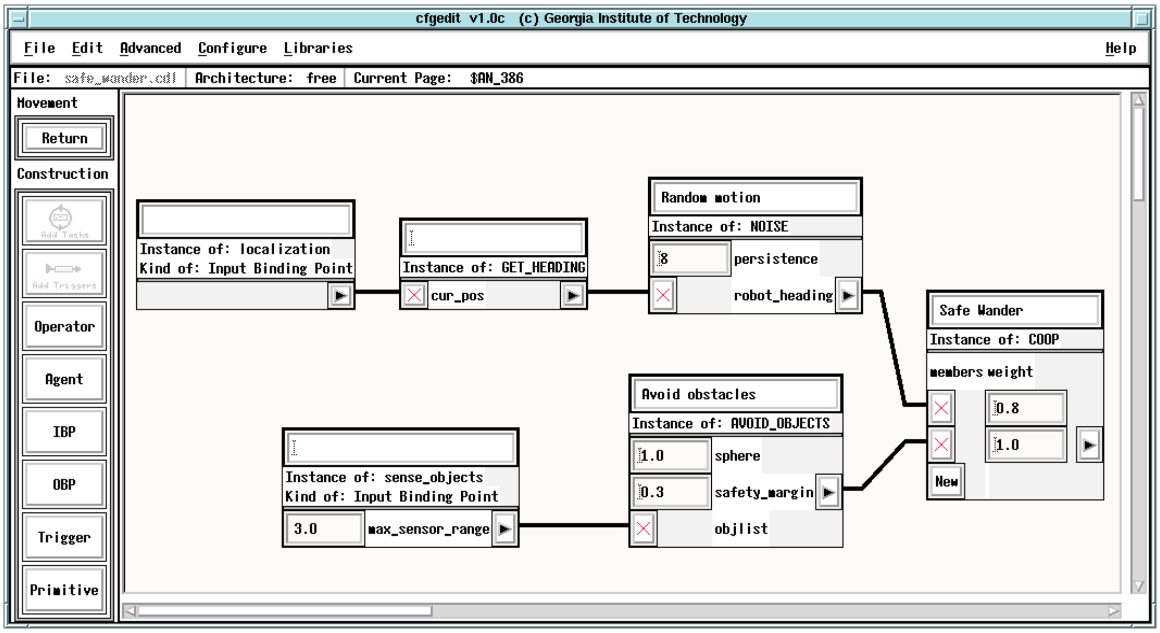
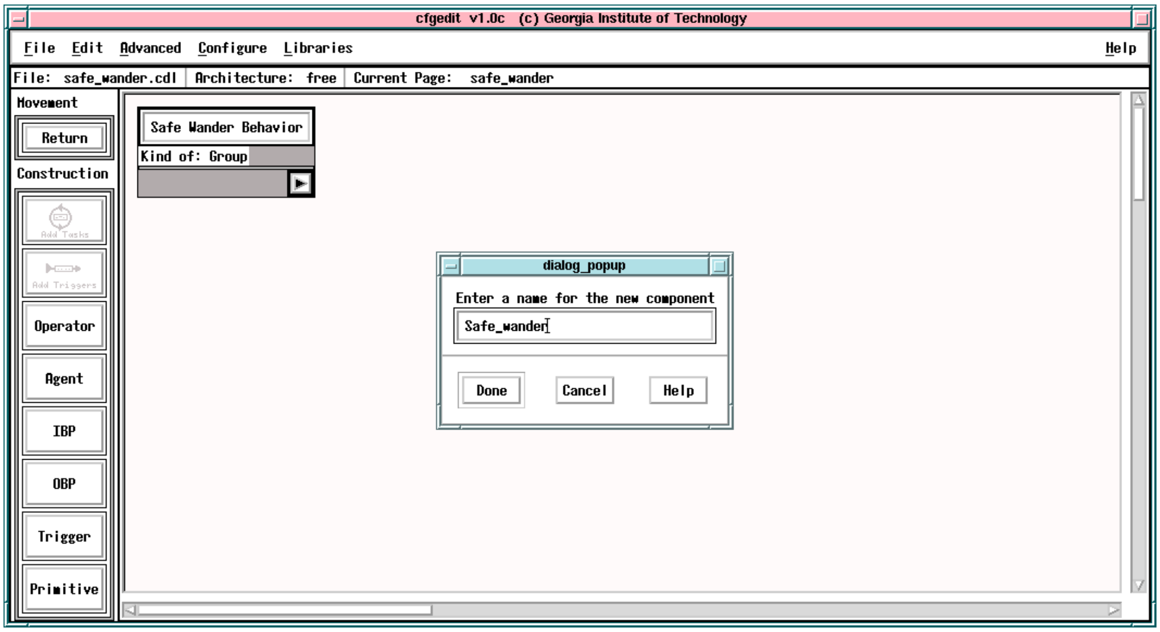


Figure 5.28: A new wander behavior is created which consists of random motion generated by the `Noise` behavior and an `Avoid_Objects` behavior to keep the robot from running into objects.

Figure 5.28 shows a new "Safe" wander behavior which was created by combining the `Noise` and `Avoid_Objects` behaviors with a cooperative coordination operator. To add this new component to a library, the operator moves up a level in the editor so that the behavior is represented as a single icon. The left mouse button is then used to select the entire behavior by clicking on this iconic representation. The "Add Selected Component to Library" option under the library pulldown menu is used to start the process.

Figure 5.29 shows the editor with the library addition process underway. The system is asking for a name to give the new component. This name will be used in



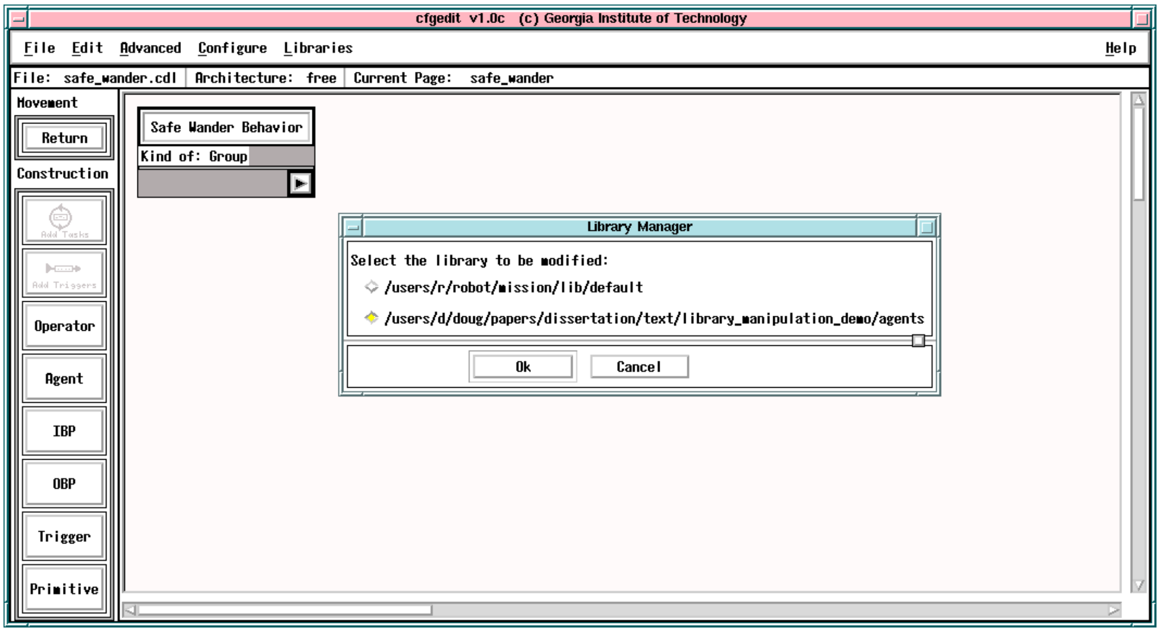
File: safe_wander.cdl | Architecture: free | Current Page: safe_wander

Safe Wander Behavior
Kind of: Group

Enter a name for the new component
Safe_wander

Figure 5.29: Moving up one level, the safe wander behavior is shown in its iconic form. The operator has selected the "Add selected to library" option from the library menu bar. The editor is now prompting for a name for the new component.

selection dialogs to identify this new behavior when users are selecting new behaviors for operating states. The string "Safe Wander Behavior" shown in the comment area of the new behavior's icon will be used as the descriptive comment, presented with the name. The name must be formatted as a traditional single word identifier using C conventions. However, free form text is allowed in the description. In this example the operator has entered `Safe_Wander` as the name of the new behavior.



File: safe_wander.cdl | Architecture: free | Current Page: safe_wander

Library Manager

Select the library to be modified:

◇ /users/r/robot/mission/lib/default

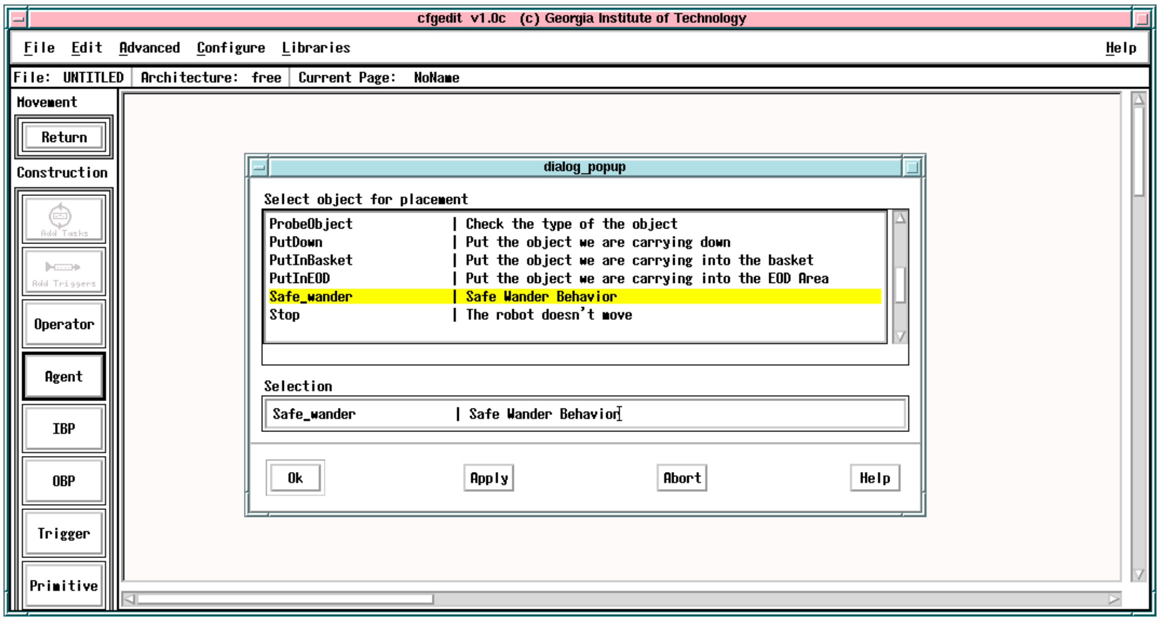
◆ /users/d/doug/papers/dissertation/text/library_manipulation_demo/agents

Figure 5.30: The list of available libraries are presented to allow the user to select the library to which the new component will be added.

After the name is entered the system prompts for the user to select to which of the libraries the behavior will be added. Figure 5.30 shows the dialog presented to the user with the list of libraries which were specified using the `CDL_libraries` variable in the `.cfgeditrc` file. The operator selects the particular library they wish to modify by clicking on the button to the left of the name and pressing `OK`. `MissionLab` now adds a copy of the component to the internal copy of the library and writes a new copy of the library file.

There are generic versions of each library file and one additional file for each supported robot architecture (*i.e.*, AuRA and UGV). The new component is added to one of these versions based on how it is bound. If the configuration contained in the editor is currently unbound (free) components are added to the generic version of the selected library. If the configuration is bound to one of the supported architectures the component is added to the corresponding library file. Note that it is necessary to add the generic version of a component before any architecture specific versions can be added.

To complete the demonstration, the editor was restarted with an empty workspace. In Figure 5.31 the operator has pressed the button to add a new agent to the workspace, causing the selection dialog to be presented with the list of available



File: UNTITLED | Architecture: free | Current Page: NoName

Select object for placement

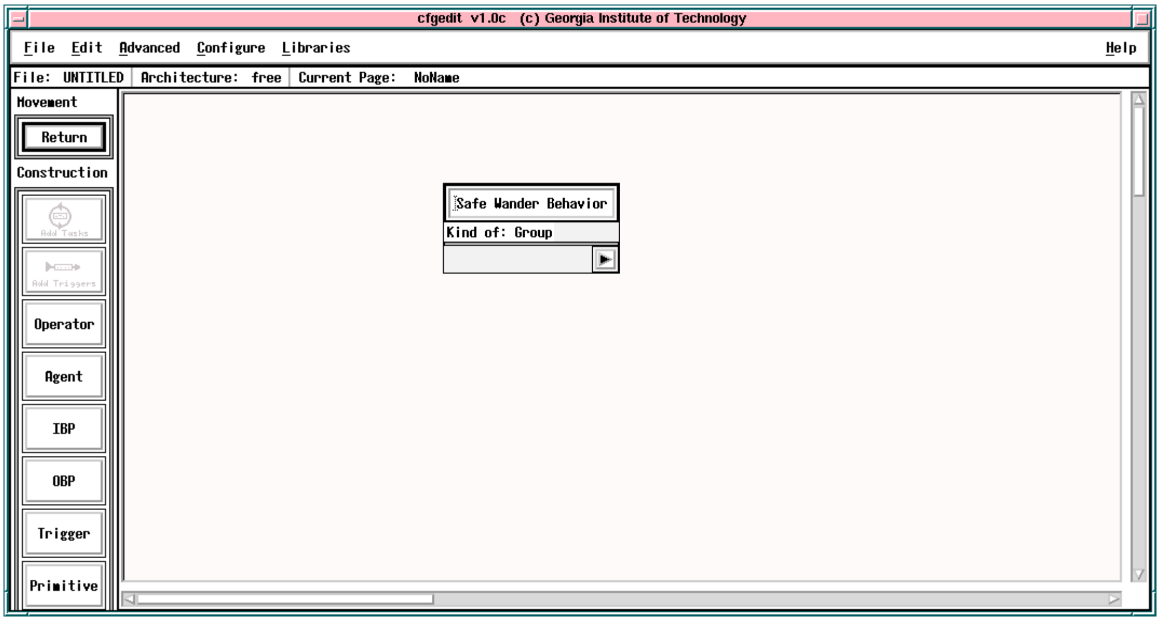
ProbeObject	Check the type of the object
PutDown	Put the object we are carrying down
PutInBasket	Put the object we are carrying into the basket
PutInEOD	Put the object we are carrying into the EOD Area
Safe_wander	Safe Wander Behavior
Stop	The robot doesn't move

Selection

Safe_wander | Safe Wander Behavior

Figure 5.31: The editor has been restarted to show that the component is now in the behavior list.

components. The highlighted entry for the new `Safe_wander` behavior shows that the component was correctly added to the library.

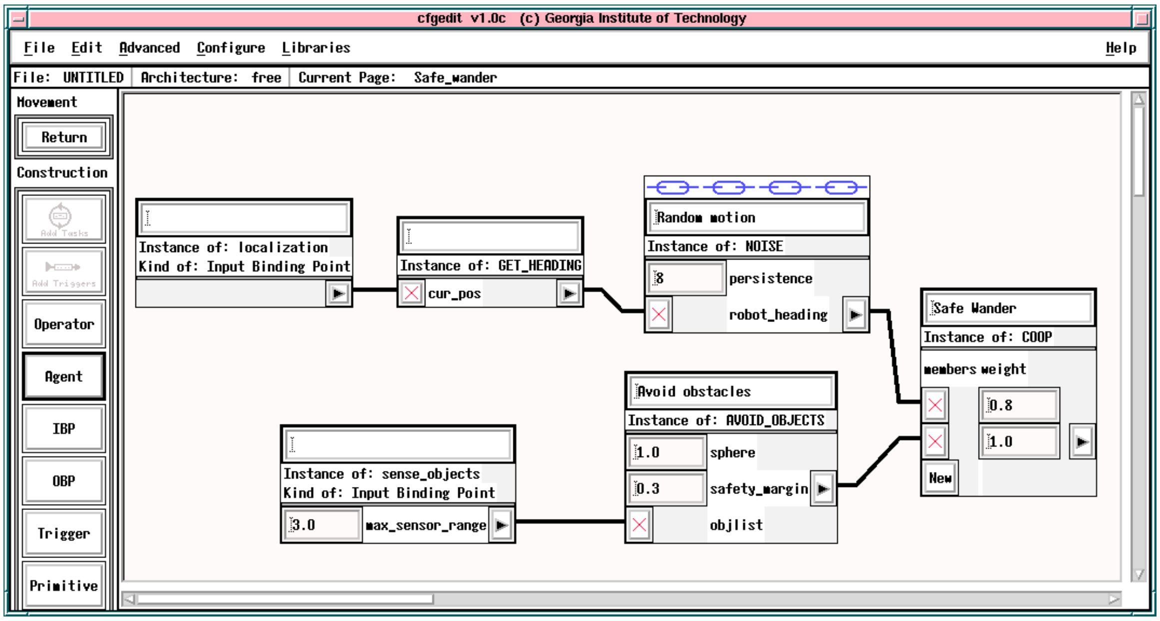


File: UNTITLED
Architecture: free
Current Page: NoName

Safe Wander Behavior
Kind of: Group

Figure 5.32: The new *safe wander* behavior was selected and placed into an empty workspace.

In Figure 5.32 the new component was placed in the workspace to allow verification of its composition. Figure 5.33 shows the detail level of the component, allowing verification that the definition was correctly added to the library. This completes the demonstration of how components are added to libraries.



File: UNTITLED
Architecture: free
Current Page: Safe_wander

Instance of: localization
Kind of: Input Binding Point

Instance of: GET_HEADING
cur_pos

Random motion
Instance of: NOISE
8 persistence
robot_heading

Safe Wander
Instance of: COOP
members weight
0.8
1.0

Avoid obstacles
Instance of: AVOID_OBJECTS
1.0 sphere
0.3 safety_margin
objlist

Instance of: sense_objects
Kind of: Input Binding Point
3.0 max_sensor_range

Figure 5.33: Moving down into the library component, the original definition is displayed.

5.8.2 Creating components containing FSA's

Creating new components which use FSA's to generate their behavior is easily accomplished. Many times a design has been created to solve a specific task and it is later decided that a portion of it is suitable for inclusion as a generic component. For example, Figure 5.34 shows an FSA developed to perform a mine cleanup mission. The task is now to create a generic cleanup component from this configuration which can be parameterized as to the object to retrieve and the destination container.

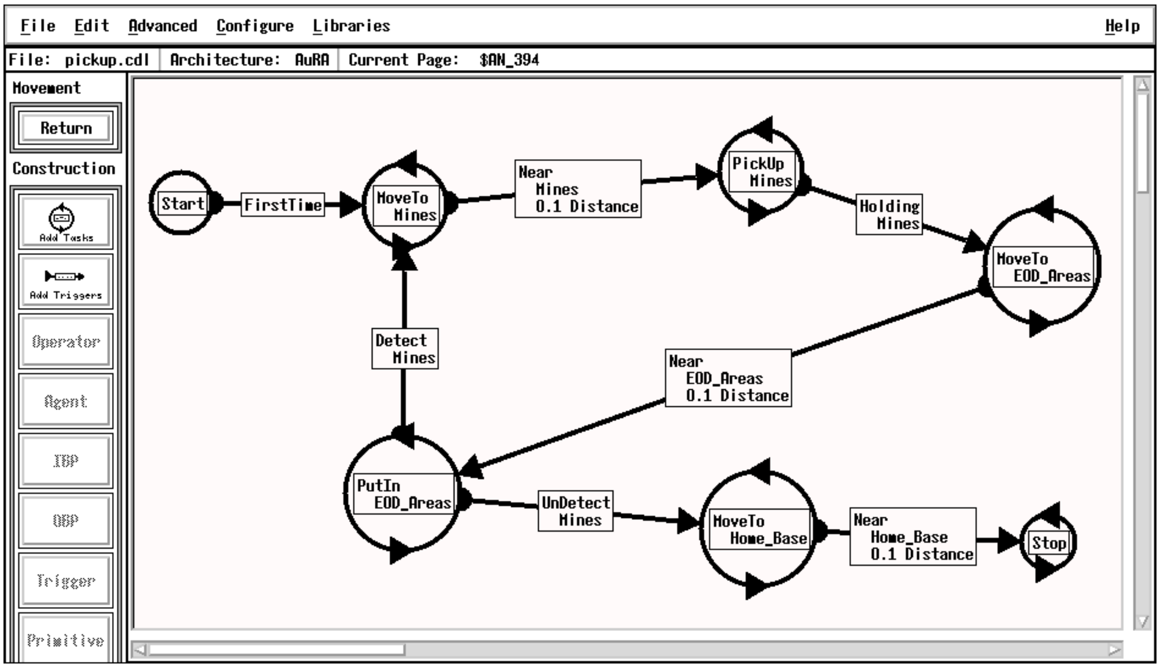


Figure 5.34: This FSA picks up mines and places them into the disposal area. When all the mines are collected, it returns to home base and halts.

The first step is to remove states extraneous to the desired component. Figure 5.35 shows the result of pruning the states in Figure 5.34 which caused the robot to return to home base after completing the cleanup operation. This leaves a loop which moves to the nearest mine, picks it up, moves to the nearest disposal area, and places the mine in the container.

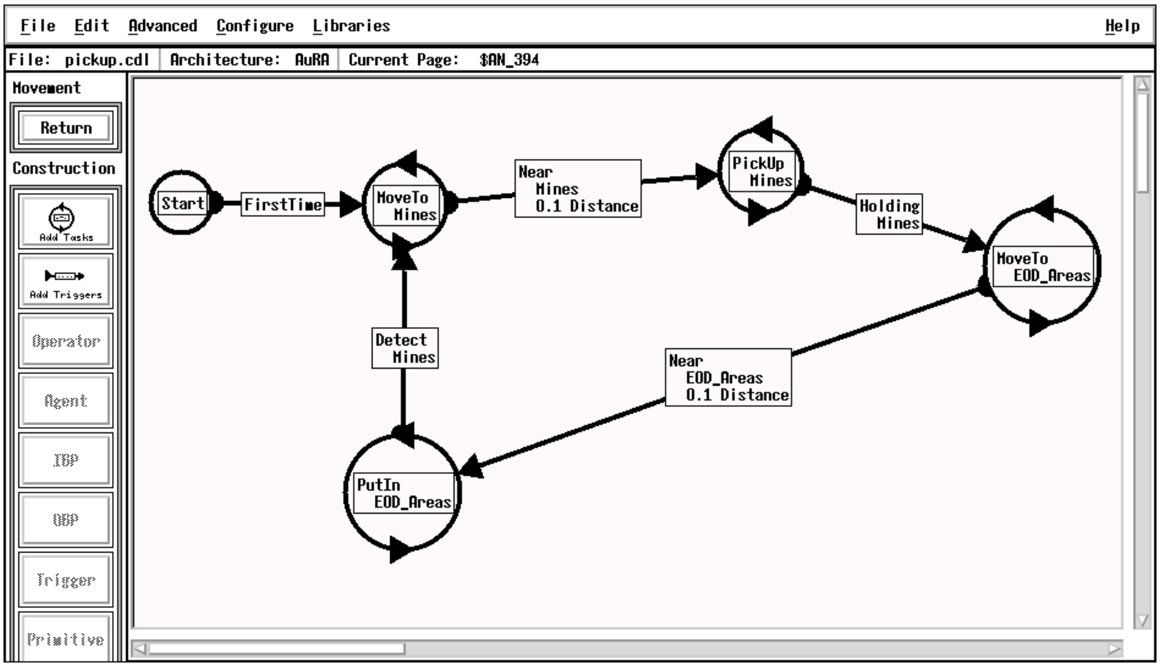
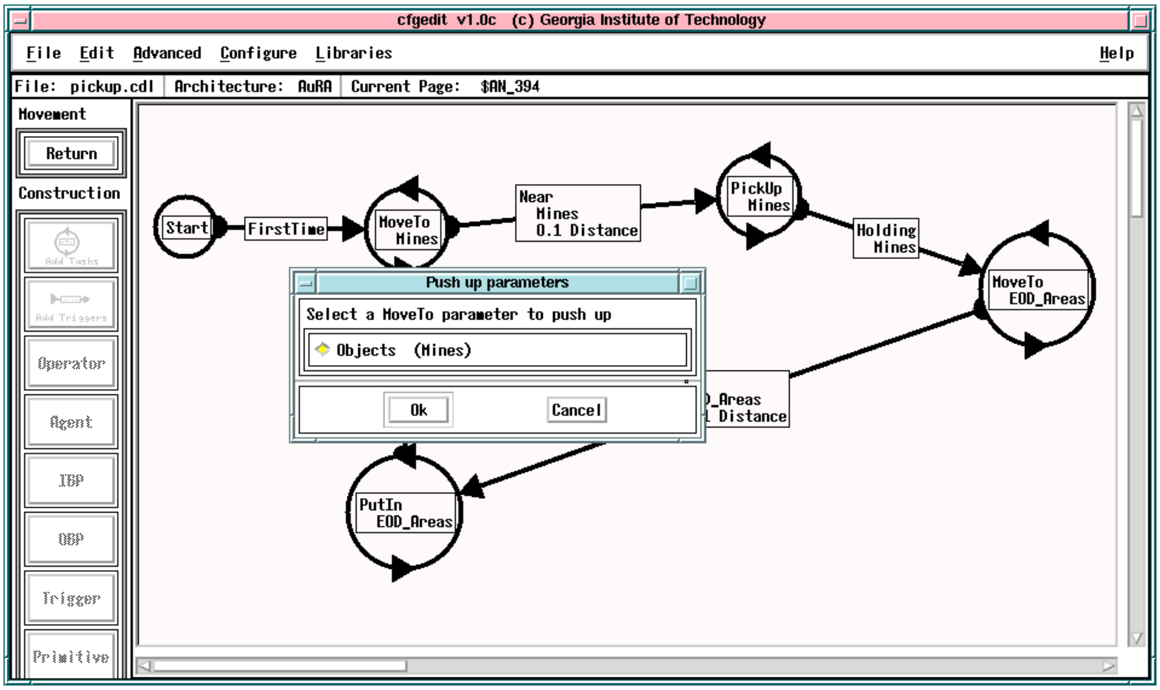


Figure 5.35: This portion of the FSA executes the mine cleanup task.

The design in Figure 5.35 is still specific to moving mines into disposal areas. Worse yet, the choice of objects to pick up (and destination containers) is duplicated in numerous places. To solve the problem of both hardcoded and duplicated object choices, we will push up the `Mines` parameter to the parent record (the iconic representation of the FSA).

In Figure 5.36 the operator has selected the "Push up input" option from the Advanced menu bar and then clicked on the `MoveTo` state. This opens a dialog box where the list of available parameters are displayed and the user can select the particular one to push up. In this case there is only one selection available (`Objects` with the value of `Mines`).



File: pickup.cdl | Architecture: AuRA | Current Page: $AN_394

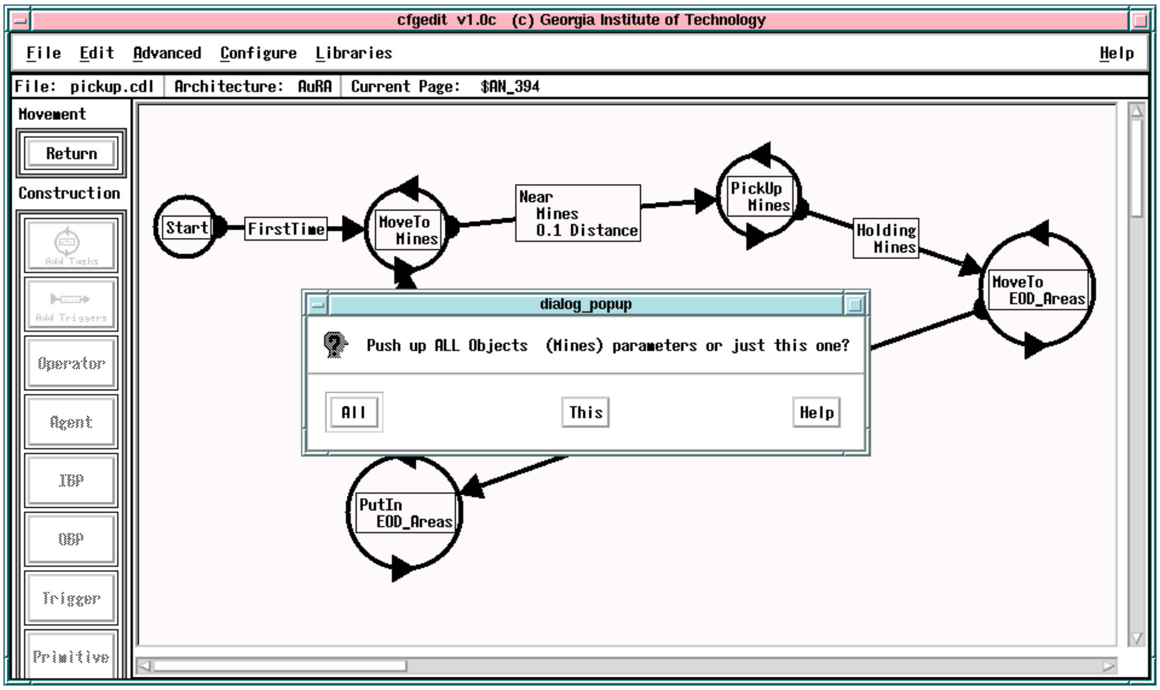
Push up parameters

Select a MoveTo parameter to push up

◆ Objects (Mines)

Figure 5.36: The user has selected the "Push up input" option from the Advanced menu bar and clicked on the `MoveTo` state. The system is now prompting for the user to select which parameter to push up into the parent.

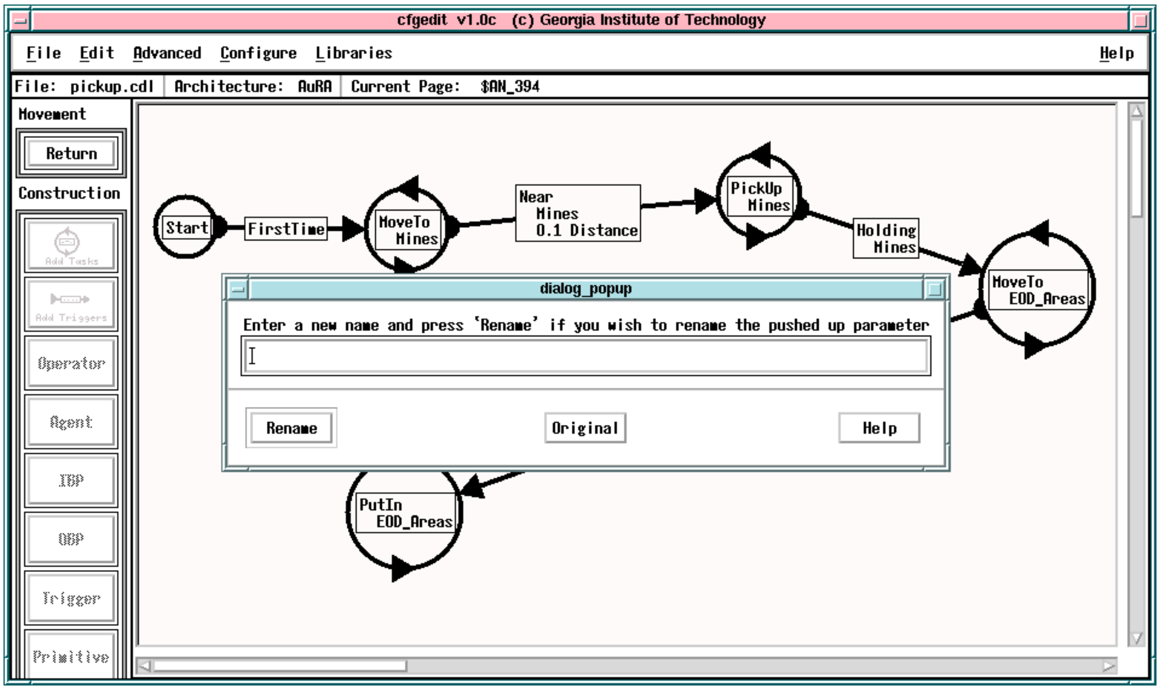
After a paramter has been selected for pushing up, the system checks to see if other instances of that parameter occur in the state diagram. In Figure 5.37 the system has determined there are multiple instances of the `Objects` parameter having the value of `Mines`. A dialog box is presented to the user to allow all of the parameters with this value or just the one selected to be pushed up.


File: pickup.cdl Architecture: AuRA Current Page: $AN_394

Push up ALL Objects (Mines) parameters or just this one?

Figure 5.37: The system has determined there are several other instances of the *Mines* parameter and is asking if all should be pushed up.

Finally, the user is presented with the option of using a new name for the parameter in the parent record. This aliasing allows descriptive names in components and elimination of name collisions. In Figure 5.38 the user leaves the parameter name as `Objects` by pressing `Original`.

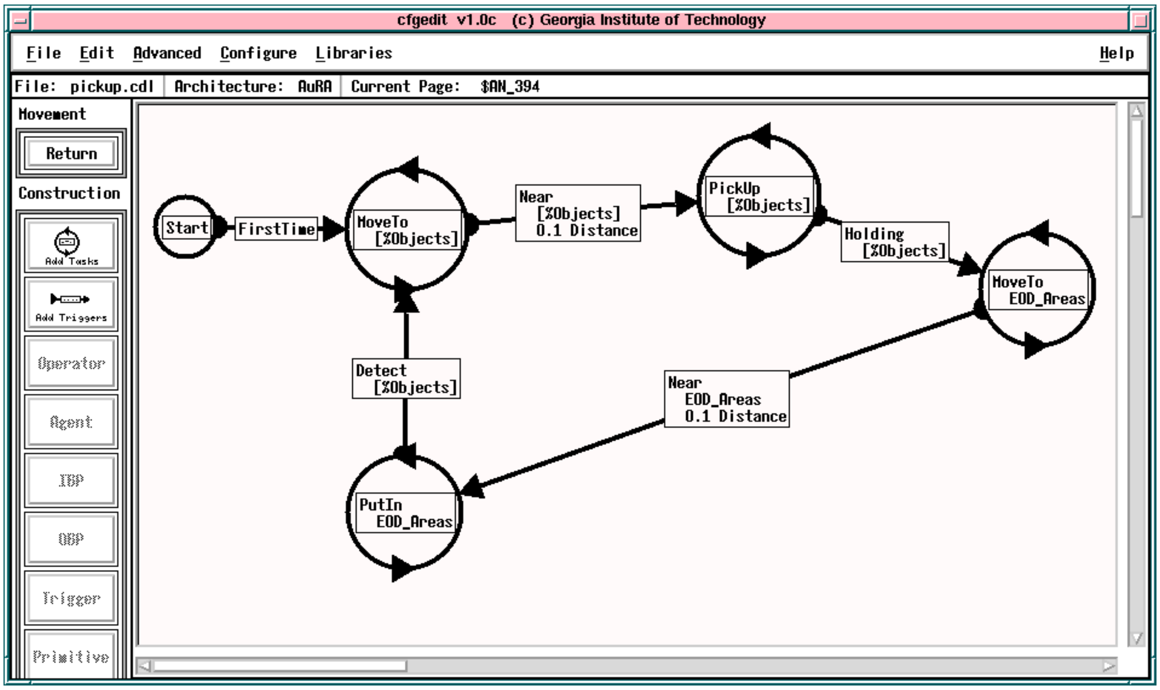


File: pickup.cdl | Architecture: AuRA | Current Page: $AN_394

Enter a new name and press 'Rename' if you wish to rename the pushed up parameter

Figure 5.38: Sometimes it is clearer if the parameter has a different name at the component level than in the definition. This dialog box allows specifying a new name for the pushed up parameter.

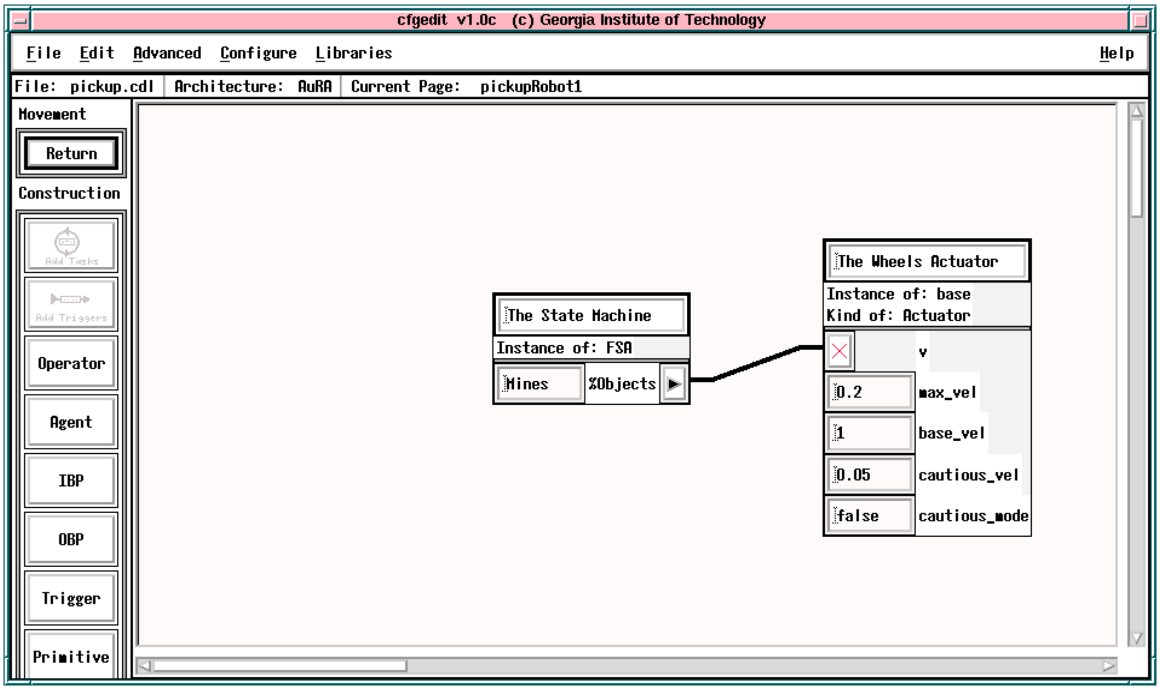
Figure 5.39 shows the FSA after the `Mines` parameter has been pushed up. Notice that in each place the parameter was referenced it now is displayed as `[%Objects]`. This signifies that the value of the parameter is deferred to the parameter in the parent record with the name `%Objects`. The percent sign "`%`" is automatically prepended to parameter names during the push up action and is used by *MissionLab* to distinguish deferred parameters from normal usages.



File: pickup.cdl
Architecture: AuRA
Current Page: $AN_394

Figure 5.39: The user pushed up all instances of the mine objects parameter.

Figure 5.40 shows the iconic representation of the FSA with the new pushed up `%Objects` parameter. Notice that it still has the value `Mines`. The definition is functionally equivalent before and after pushing up the parameter. The location where the value is entered is the only thing that has changed.



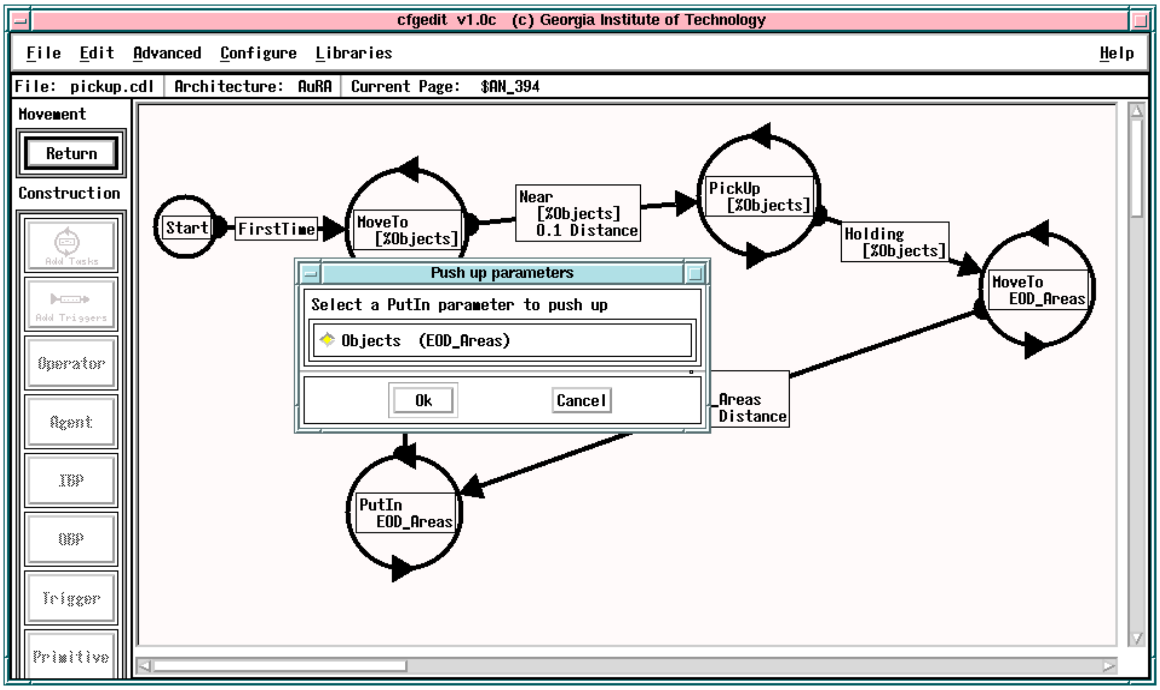
File: pickup.cdl
Architecture: AuRA
Current Page: pickupRobot1

The State Machine
Instance of: FSA
Mines %Objects

The Wheels Actuator
Instance of: base
Kind of: Actuator
v
0.2 max_vel
1 base_vel
0.05 cautious_vel
false cautious_mode

Figure 5.40: The iconic representation of the FSA now includes the `%Objects` parameter which determines which class of objects the behavior picks up.

The parameter selecting the container where the mines are to be placed is pushed up in Figure 5.41. The same process is used to select the parameter to defer as before.



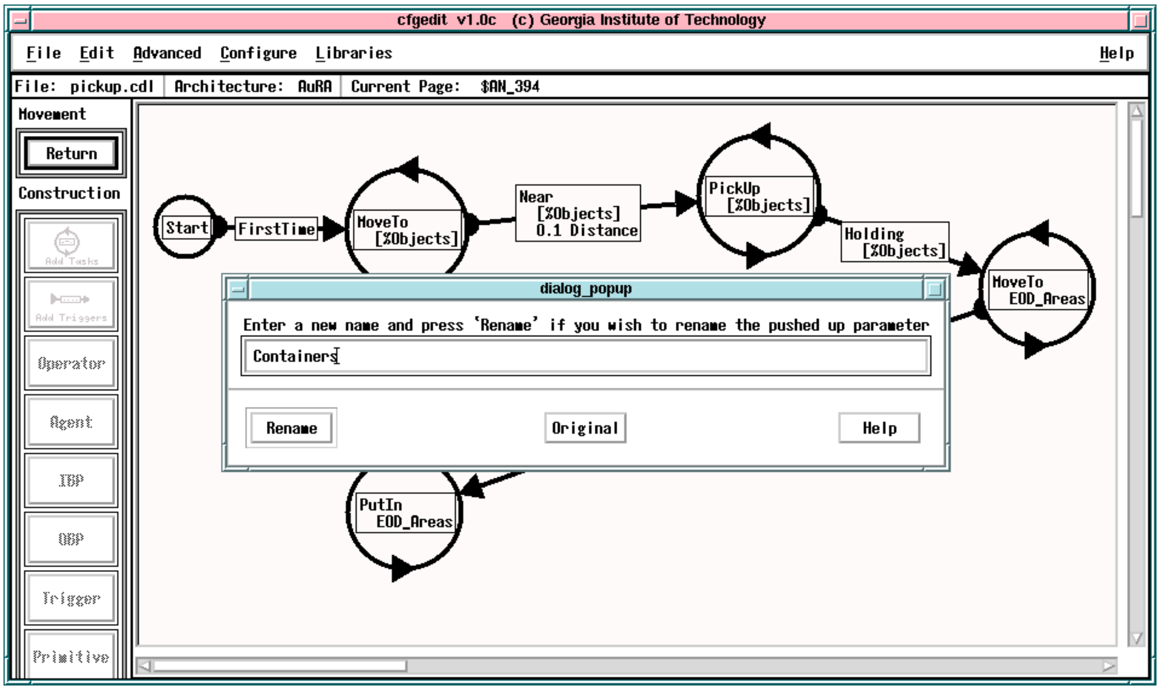
Push up parameters

Select a PutIn parameter to push up

◆ Objects (EOD_Areas)

Figure 5.41: The user now pushes up the parameter selecting the container in which to put the mines.

Notice in Figure 5.41 the parameter is also named `%Objects`. It is renamed to `Containers` in Figure 5.42 to eliminate duplication and provide a meaningful parameter at the component level.



File: pickup.cdl | Architecture: AuRA | Current Page: $AN_394

Enter a new name and press 'Rename' if you wish to rename the pushed up parameter

Containers

Figure 5.42: The parameter is aliased to `Containers` at the component level.

The final state diagram after the parameters are pushed up is shown in Figure 5.43. Notice all instances of the `Mines` and `EOD_Area` parameters have been pushed up. This FSA diagram now defers selection of these two important criteria to the parent record.

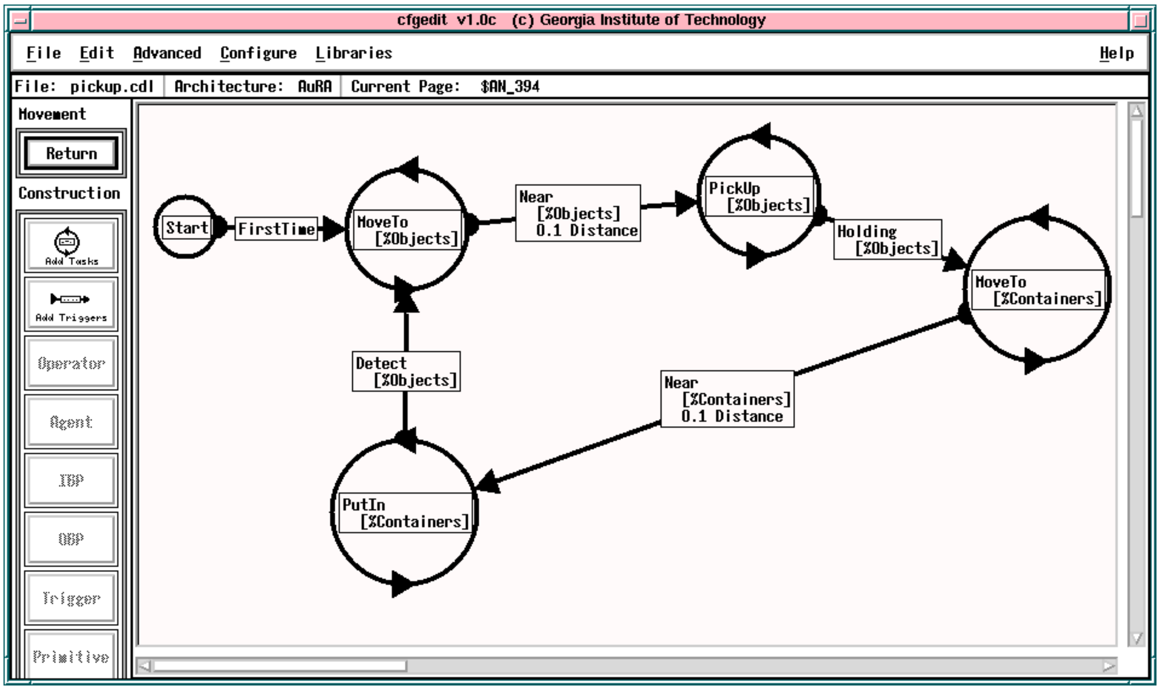
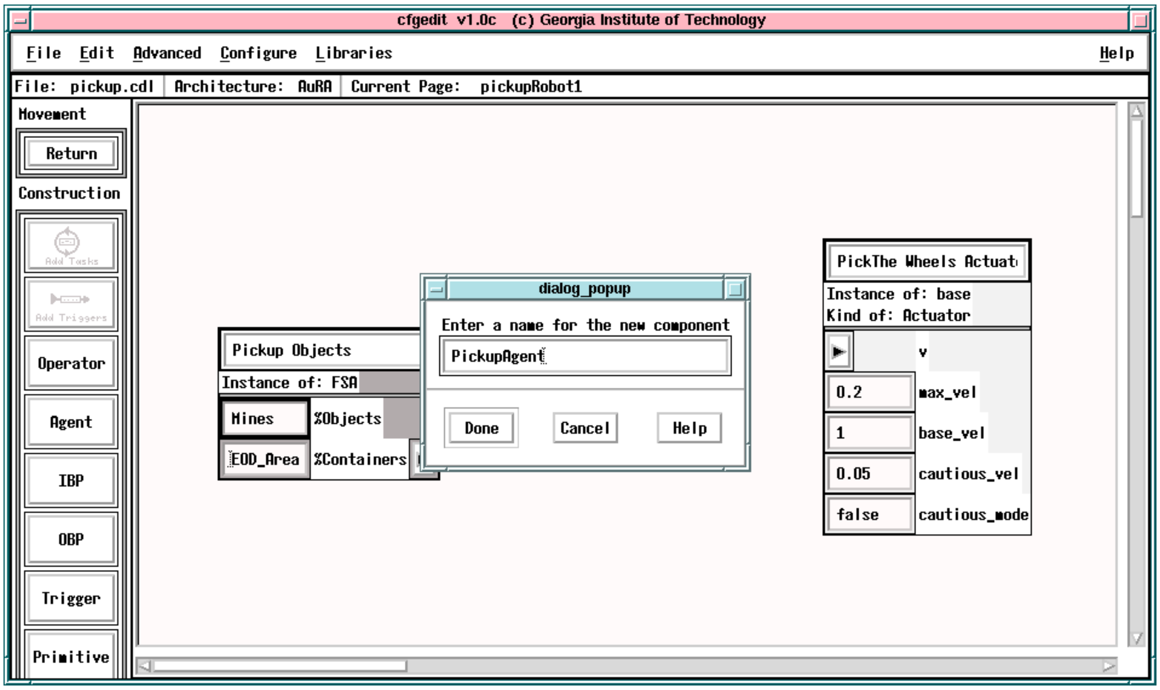


Figure 5.43: The FSA after the two parameters have been pushed up.

The iconic representation of the FSA is shown in Figure 5.44. Notice that the goal of combining the multiple references to the two parameters has been achieved. There is only one instance of each of the two object classes. There are multiple links to each parameter from the state diagram. This one to many relationship provides information hiding and simplification as part of the recursive construction of components.

Figure 5.44 also shows the operator starting to add the new generic `PickupAgent` component to the library. The process of adding components to libraries was documented in Section 5.8.1.



File: pickup.cdl
Architecture: AuRA
Current Page: pickupRobot1

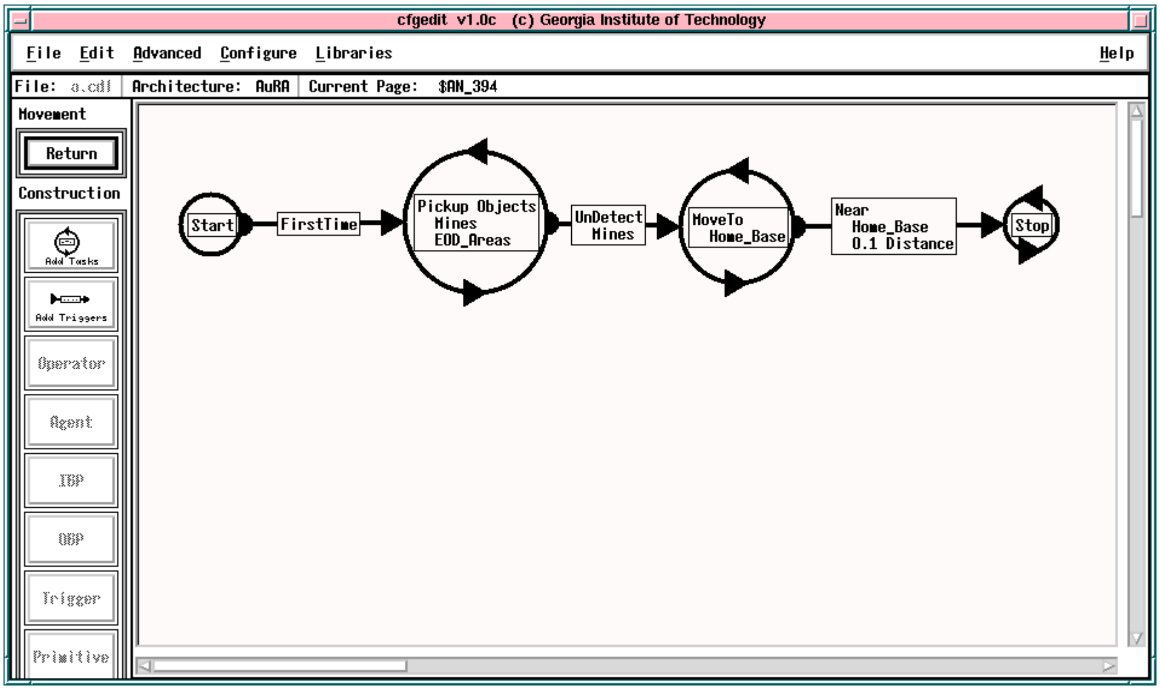
Enter a name for the new component
PickupAgent

Pickup Objects
Instance of: FSA
Mines %Objects
EOD_Area %Containers

PickThe Wheels Actuat
Instance of: base
Kind of: Actuator
v
0.2 max_vel
1 base_vel
0.05 cautious_vel
false cautious_mode

Figure 5.44: The completed FSA component with the two pushed up parameters and default values specified. The operator has started to add the component to the library and given it the name `PickupAgent`.

Finally, in Figure 5.45, the new component has been used to recreate a simplified version of the original mine cleanup configuration. This new configuration uses the `PickupAgent` as a coherent component to provide the operator with a greatly simplfied design over the original in Figure 5.34.

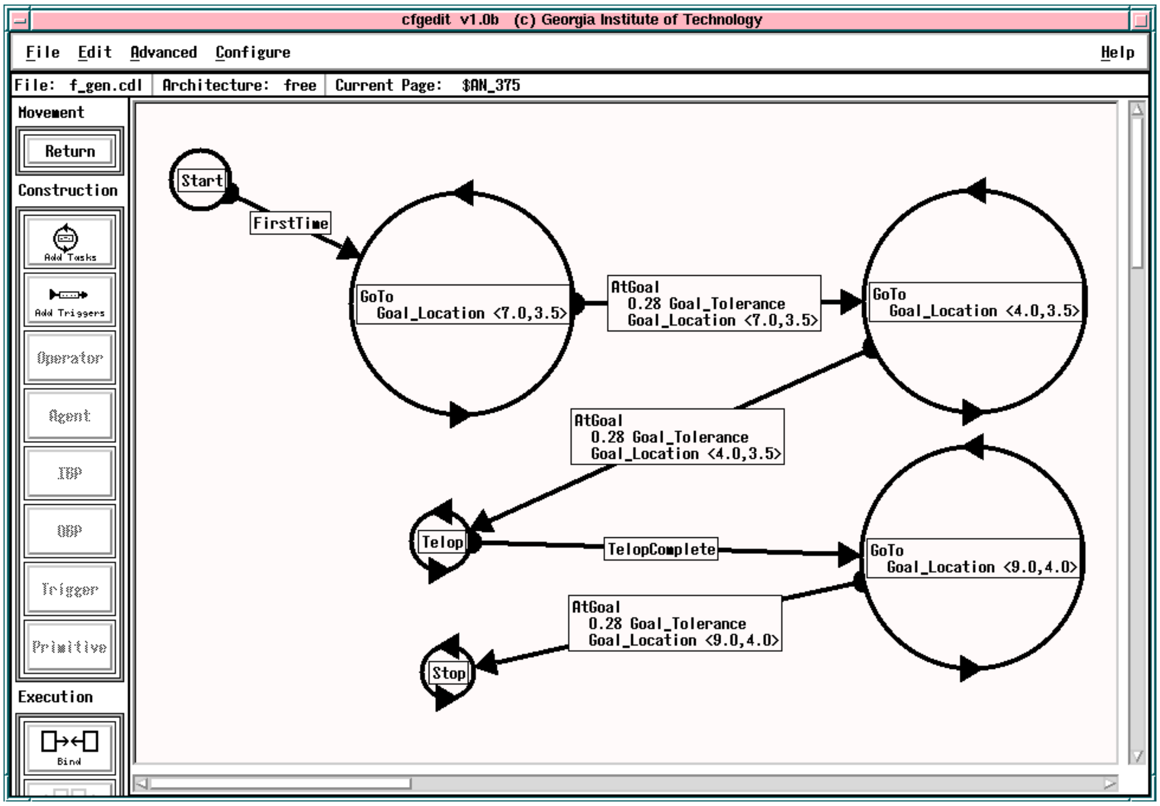


File: a.cdl Architecture: AuRA Current Page: $AN_394

Figure 5.45: The new component has been used to create a simplified version of the original FSA shown in Figure 5.34. Recall that the pickup objects state nests the FSA from Figure 5.43.

5.8.3 Retargeting a configuration

Configurations properly constrained to use only the available behaviors can be bound to the UGV architecture. In this case the SAUSAGES code generator is used. There are currently three available behaviors; move to goal, follow road, and teleoperate. SAUSAGES is a LISP-based script language tailored for specifying sequences of behaviors for large autonomous vehicles.



File: f_gen.cdl | Architecture: free | Current Page: $AN_375

Figure 5.46: Generic configuration suitable for binding to either the AuRA or SAUSAGES architecture. It causes the robot to move through two waypoints to an area where it is teleoperated, then returns to the starting area before halting.

Figure 5.46 shows the state transition diagram for a mission constructed within these limits. The robot moves through two waypoints to an area where it is teleoperated, then returns to the starting area before halting. First the configuration is

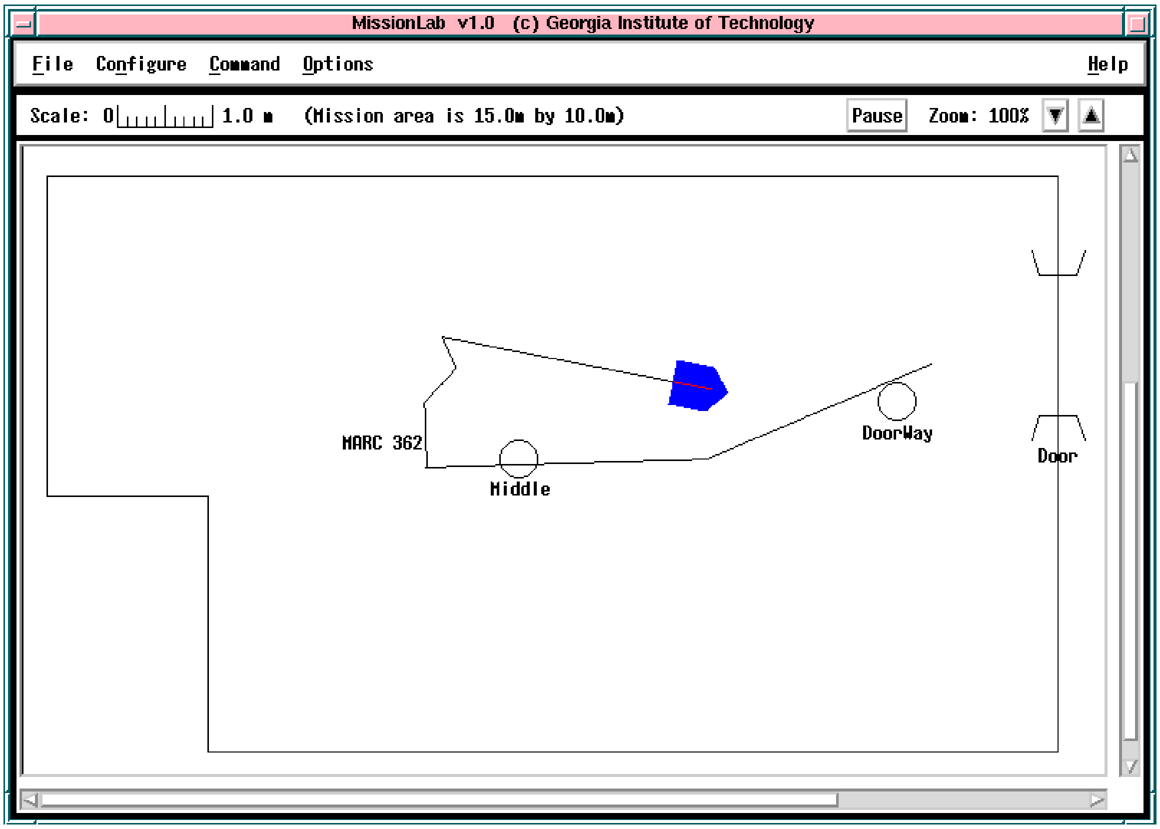


Figure 5.47: The configuration from Figure 5.46 executing in the *MissionLab* simulator. The two circles are landmarks in the map overlay not used during this mission.

bound to the AuRA architecture and deployed on the MRV-2 robots. Figure 5.47 shows the configuration executing in the *MissionLab* simulation system. Figure 5.48 shows the same executable controlling one of our Denning MRV-2 robots. Note that the same operator console is used to control simulated and real robots, so Figure 5.48 appears very similar to that in Figure 5.47 even though the first reflects a simulated execution and the second shows a real robot run. Figures 5.49 and 5.50 show the robot during the mission.

As a final demonstration, the configuration is unbound and rebound to the UGV architecture. The code generator now emits LISP-based SAUSAGES code suitable for use by the SAUSAGES simulator developed at Carnegie-Mellon University. Figure 5.51 is a screen snapshot of the SAUSAGES simulator after execution of the

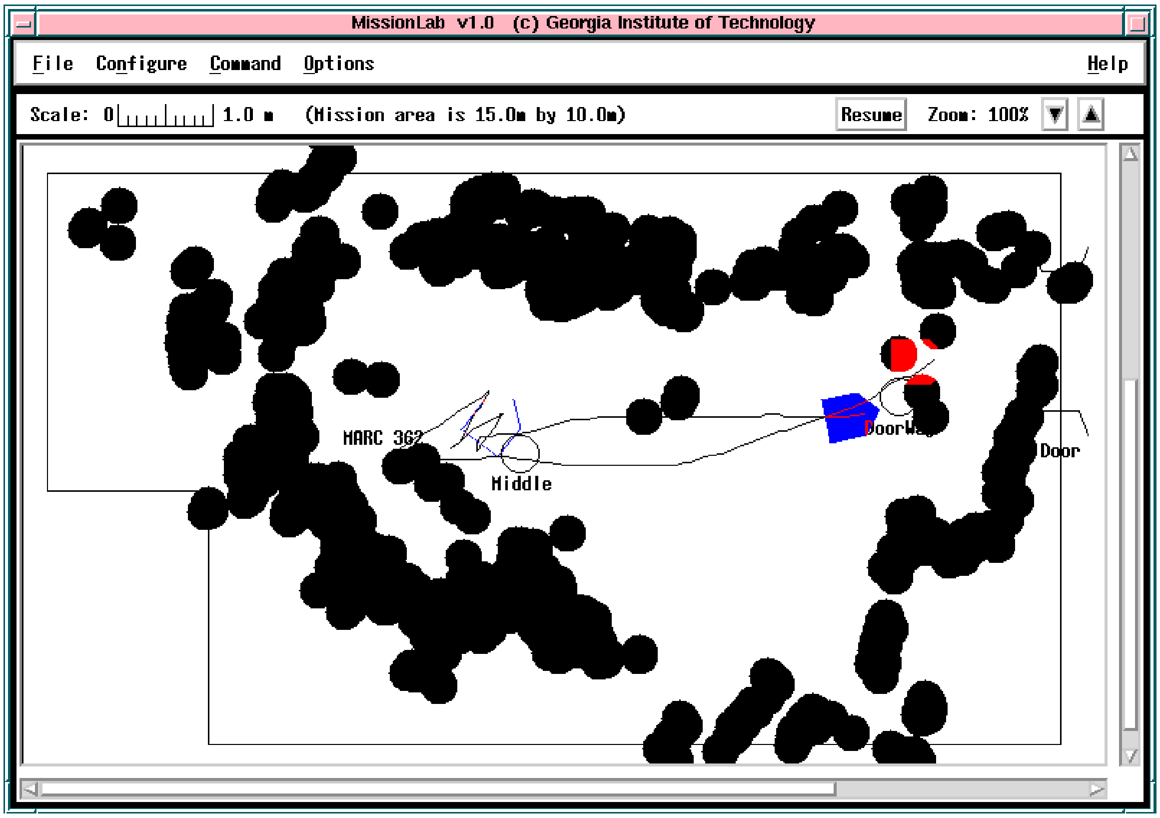


Figure 5.48: Snapshot of operator console after executing the mission shown in Figure 5.46 on a real MRV-2 Denning robot. The dark circles mark obstacle readings during the run in the Mobile Robot Lab. The same map overlay was used as in the simulated mission.

mission. The robot does not leave trails in this simulator, although the waypoints are connected by straight lines to show the projected route.

Figure 5.49: Photo of robot executing the mission in Figure 5.46 at start/finish location near the `doorway` landmark.

Figure 5.50: Photo of robot executing the mission in Figure 5.46 during the teleoperation portion. It is currently located near the `middle` landmark in the map overlay.

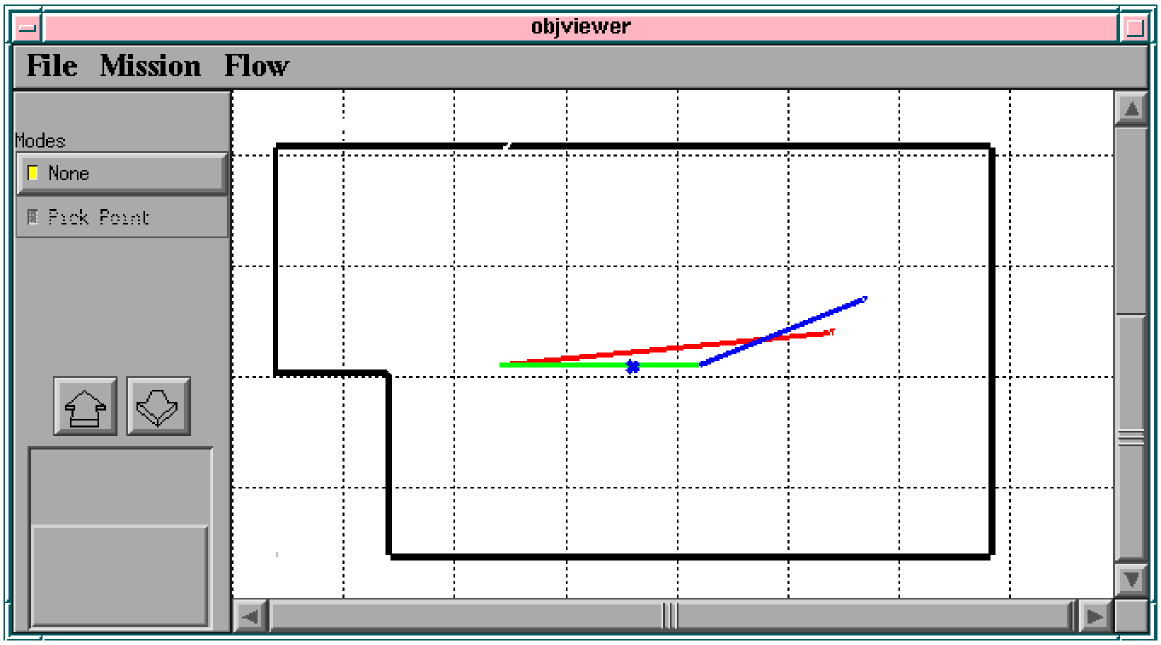


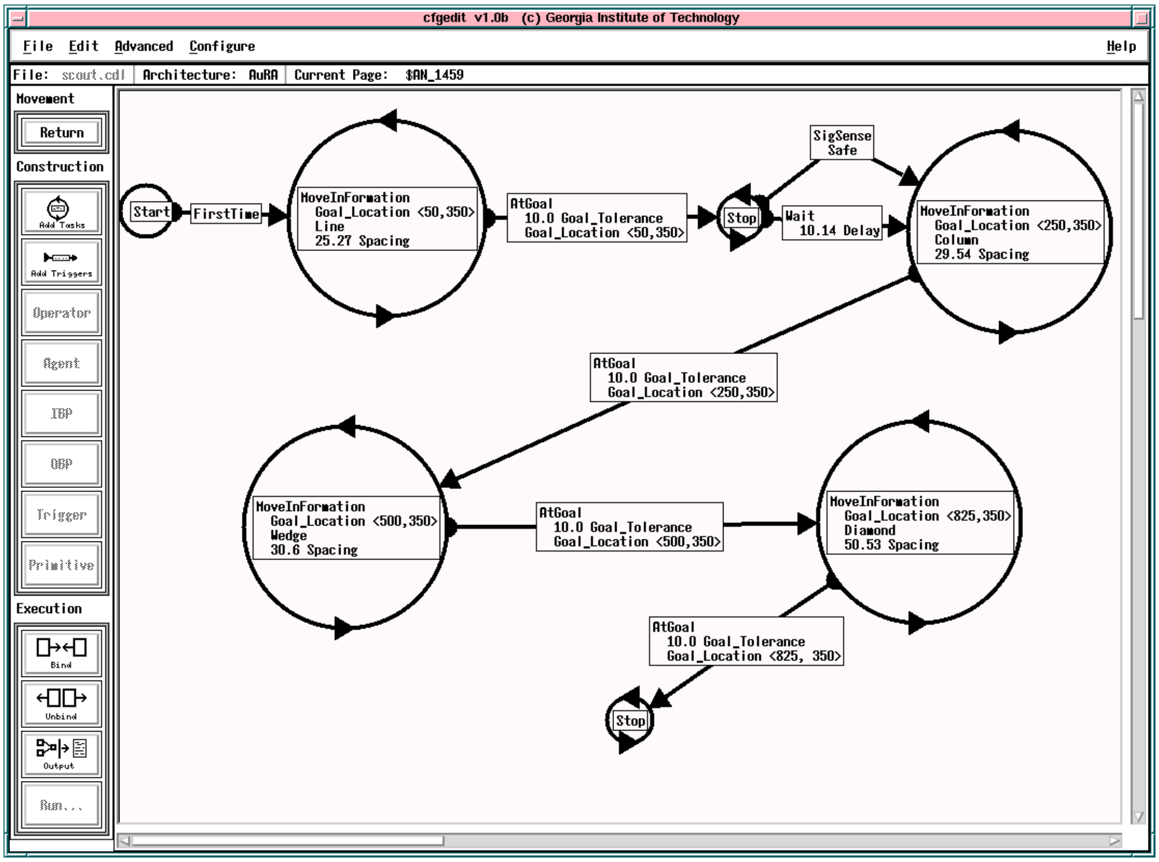
Figure 5.51: Snapshot of SAUSAGES simulation display after executing the mission shown in Figure 5.46. Notice the same general route was taken by the robot.

5.8.4 Simulated Robot Scouting Mission

A four-robot scouting mission has been constructed and evaluated in simulation. A behavior called `MoveInFormation` was created which causes the robot to move to a specified map location while maintaining formation with other robots [6]. The robots each have an assigned spot in the formation and know the relative locations of the other robots. Each robot computes where it should be located relative to the other robots, and the *Maintain_Formation* behavior tries to keep it in position as the formation moves. The choice of formation can be selected from Line, Wedge, Column, and Diamond. The separation between robots in the formation is also selectable at the FSA state level.

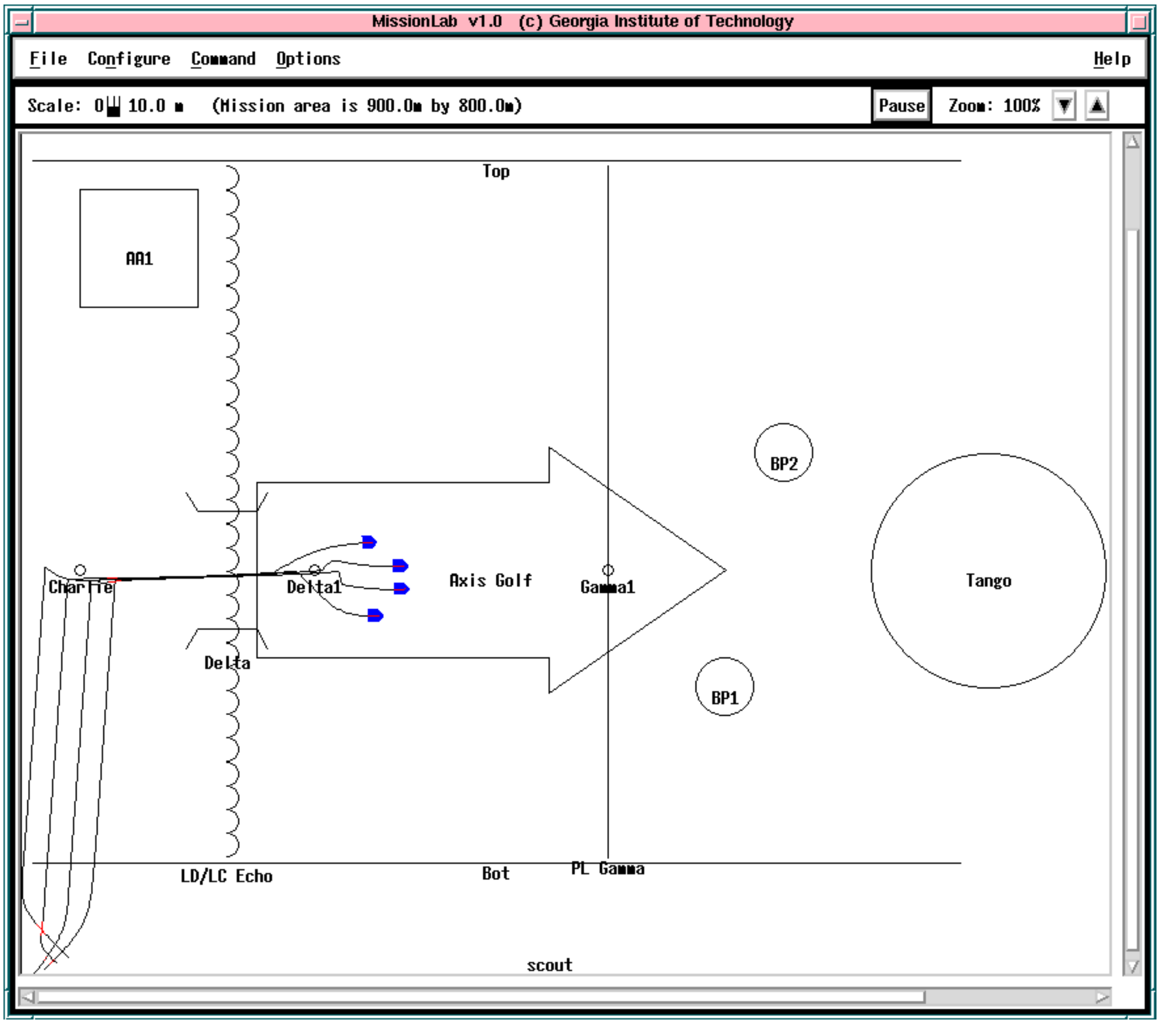
Figure 5.52 shows the state transition diagram used in the mission. In this case, explicit coordinates are used as destinations. Notice the robots begin moving in line formation. They then switch to column formation to traverse the gap in the forward lines (passage point). The robots travel along the axis of advance in wedge formation and finally occupy the objective in a diamond formation.

Figure 5.53 shows the robots during execution of the scout mission in the *MissionLab* simulator. The robots started in the bottom left corner moving up in line formation, then moved right in column formation, and are now moving to the right in a wedge formation. Figure 5.54 shows the completed mission with the robots occupying the objective in a diamond formation.



File: scout.cdl Architecture: AuRA Current Page: $AN_1459

Figure 5.52: The state transition diagram for the scouting mission.



Scale: 0 10.0 m (Mission area is 900.0m by 800.0m)

Zoom: 100%

Figure 5.53: The mission executing in the *MissionLab* simulator. The robots started in the bottom left corner moving up in line formation, then moved right in column formation, and are now moving to the right in a wedge formation.

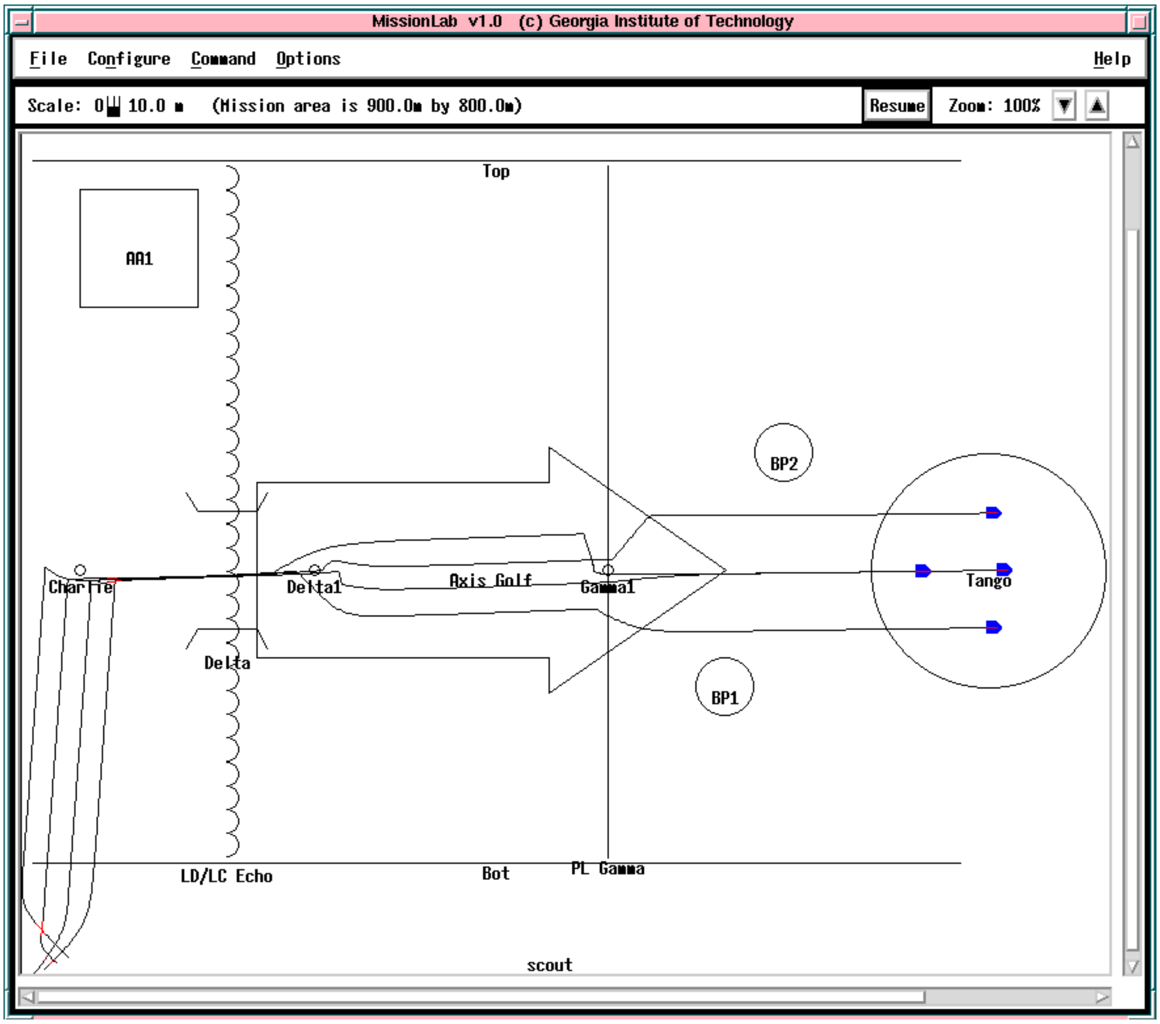


Figure 5.54: The completed scout mission with the robots occupying the objective in a diamond formation.

5.8.5 Indoor Navigation with Two Robots

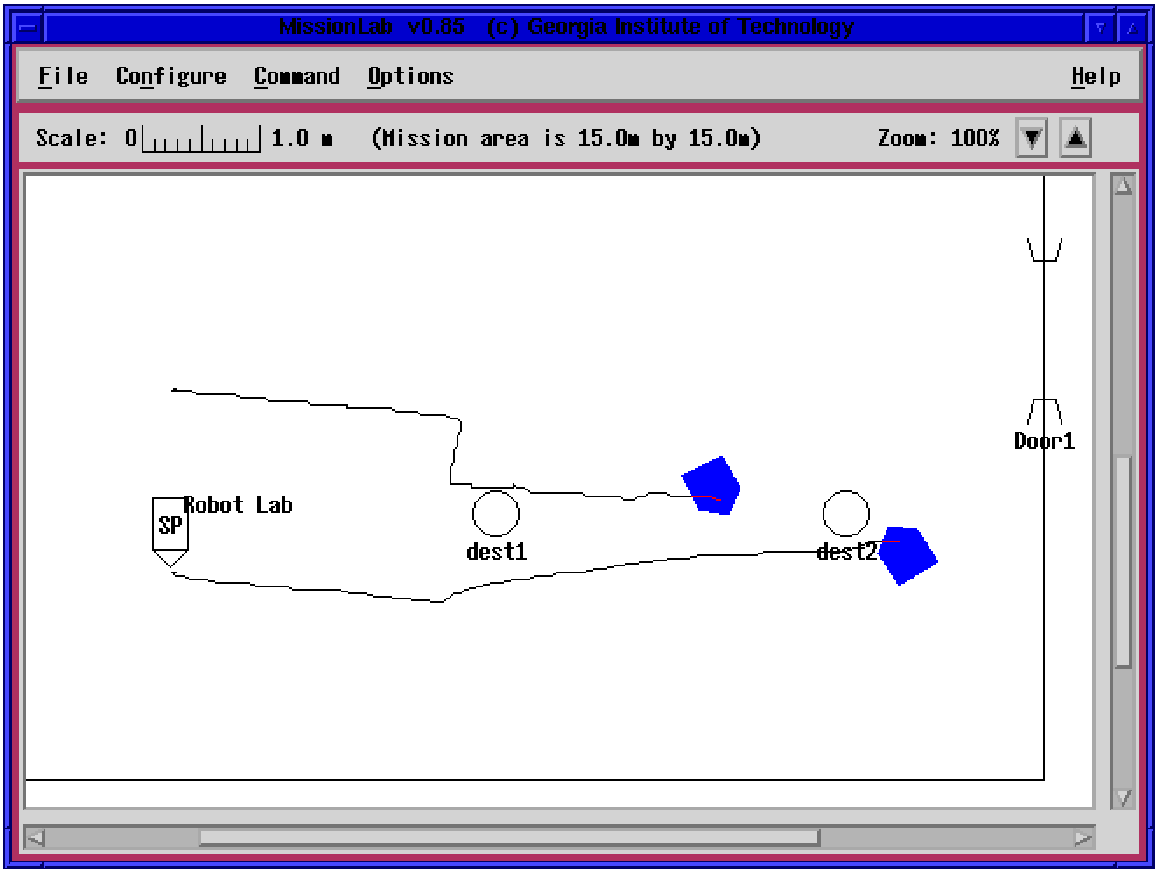
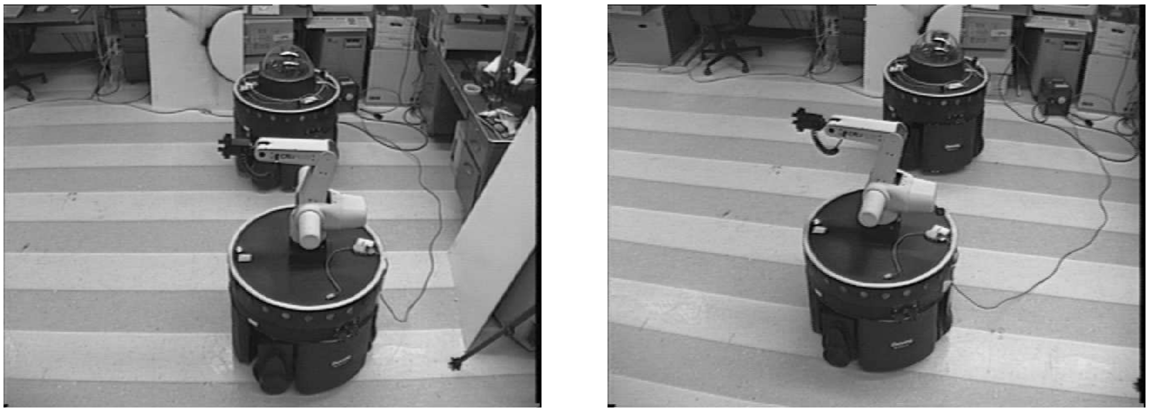


Figure 5.55: *MissionLab* showing the operator console after execution of a simple two-robot mission. The robots start on the left edge of the lab and proceed to the `dest1` point in line formation. They then continue to location `dest2` using column formation. They are shown in their final positions, with the trails marking the path each traversed.

Figure 5.55 shows *MissionLab* with the overlay representing the Georgia Tech Mobile Robot Lab loaded. The gap in the upper right represents the door to the laboratory. The goal circles were positioned arbitrarily to use as targets for the move-to-goal behavior in the mission. The pair of robots are shown in their final positions, after completion of the mission. The mission starts the robots on the left edge of the room and sends them to point *dest1* in line formation. Upon reaching this waypoint, they convert to column formation and move to point *dest2* on the right side of the room. The trails taken by the robots are shown, as are their final positions. Figure 5.56 shows a sequence of photographs of the robots executing this mission.

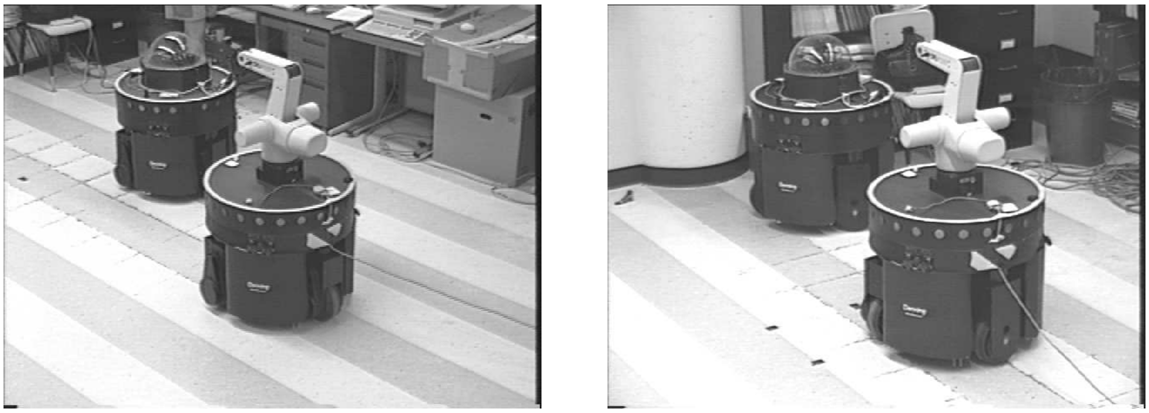


1. Robots in start location

2. Moving towards `dest1`

3. Robots at location `dest1`

4. Moving towards `dest2`



5. Robots nearing location `dest2`

6. Completed mission

Figure 5.56: Photos of the robots executing the two robot mission.

5.9 Availability of *MissionLab*

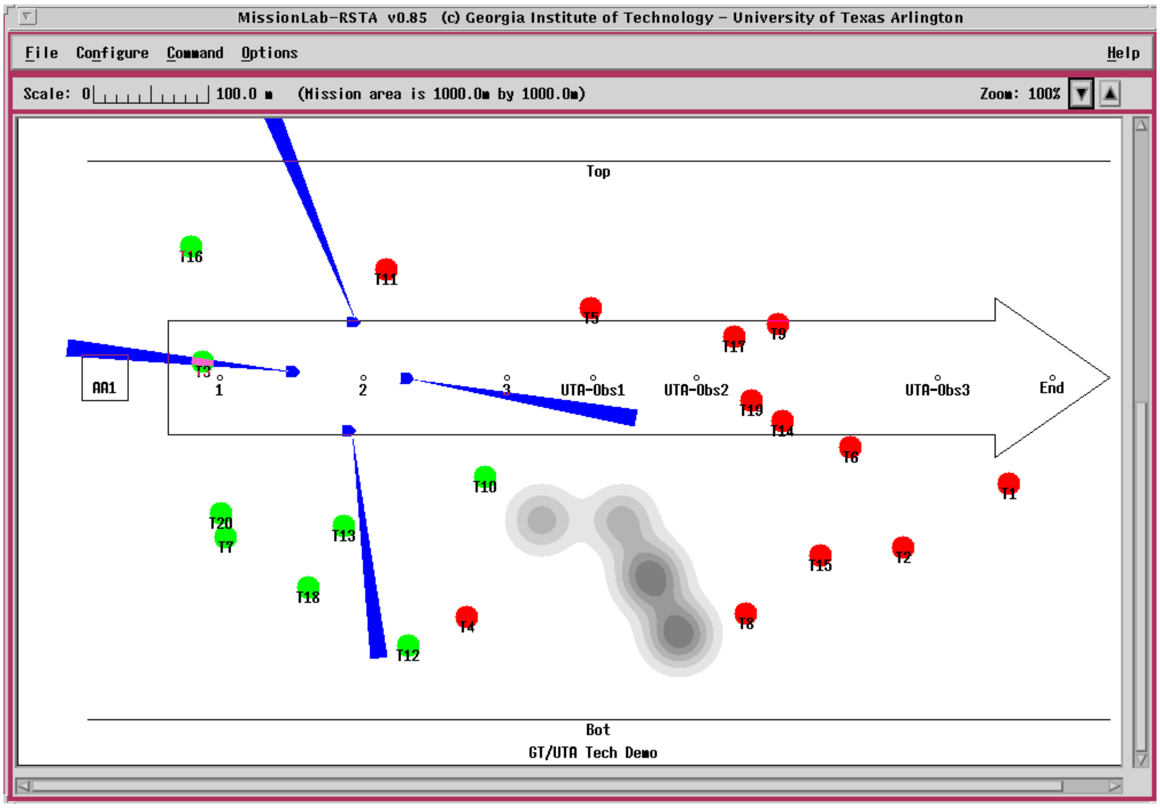
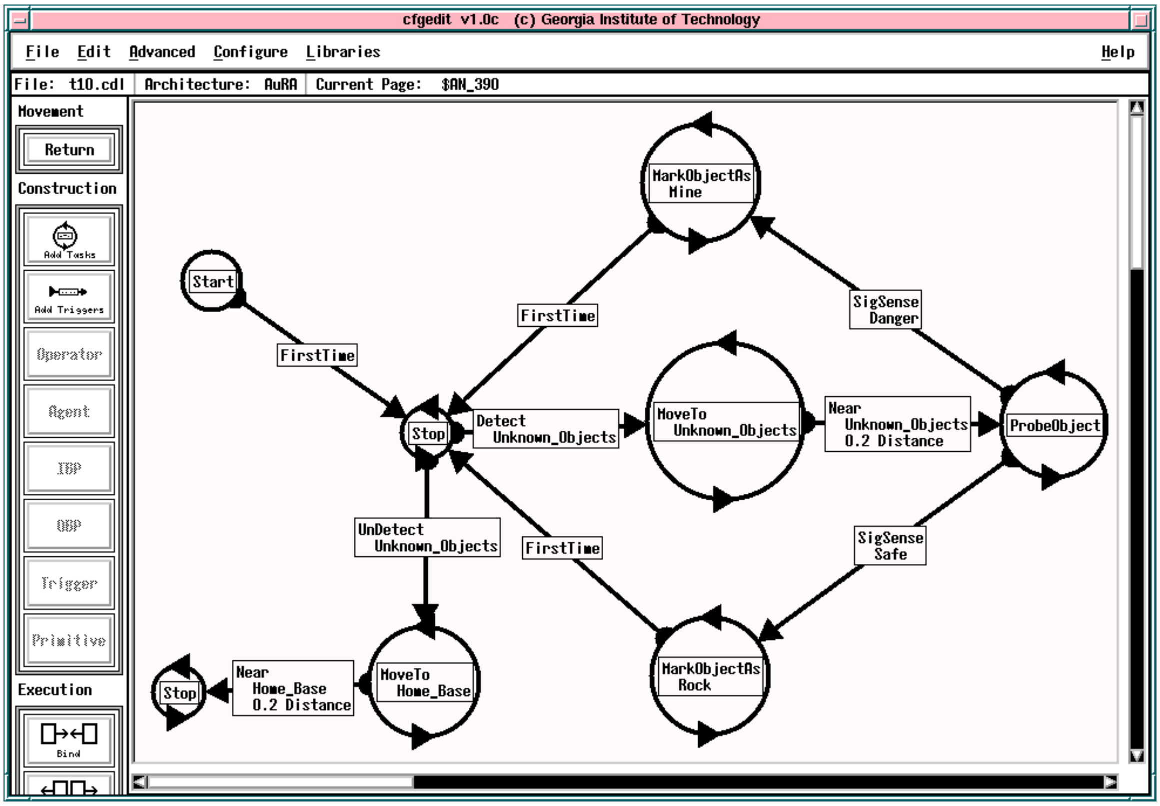


Figure 5.57: *MissionLab* with the UTA extensions for cooperative sensor pointing. The four robots are scouting an area and cooperatively looking for targets. The black wedges radiating from the robots represent areas currently being swept by the sensors. The large arrow shows the axis along which the robots are advancing. The targets the robots are searching for are shown as shaded circles labeled with the target number.

The *MissionLab* system is available in both source and binary form at
`http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab`
and has generated interest in the robotics community. The University of Texas at Arlington uses *MissionLab* in their research in cooperative sensor pointing. Figure 5.9 shows a screen snapshot of the *MissionLab* toolset with the UTA extensions.

5.10 The Frame Problem in *MissionLab*

During the usability experiments reported in Chapter 7 a so-called "race condition" manifested itself in *MissionLab*. Upon further investigation, the problem was found to be related to the general frame problem and raised the question as to how best to present an asynchronous world to novice robot operators. This section addresses these issues and suggests possible solutions.



cfgedit v1.0c (c) Georgia Institute of Technology

File: t10.cdl Architecture: AuRA Current Page: $AN_390

Figure 5.58: An FSA showing the *MissionLab* race condition. When a mine is labeled using the `MarkObject` action, the `Detect` perceptual primitive will still register the old form of the object for a small amount of time due to the asynchronous nature of the simulation. This gets the system stuck in the `MoveTo` action when the last object is marked, since there are no more objects to move towards.

Task 4 in Experiment 1 and Experiment 2 asked the participants to program the robot to map a mine field. The robot moved to `Brown` colored objects, invoked a `Probe` action, and then marked the objects as either safe or dangerous by "painting" them different colors. A solution which displays the problem is shown in Figure 5.10.

When a mine is labeled using the `MarkObject` action, the `Detect` perceptual primitive will still register the old form of the object for a small amount of time due to the asynchronous nature of the simulation. This corresponds to the relatively large amount of time required for actuators to make changes in the environment. However, the `MarkObject` action does not block until the action is finished, allowing the system to re-detect the object just marked and start moving towards it again. Once this old object changes color, the `MoveTo` action will swap to the closest remaining unknown object without any warning to the operator. This logic bug is only apparent to the user when the last object is marked. Since there are no remaining unknown objects, the `MoveTo` action stops generating motion and the robot just sits in place.

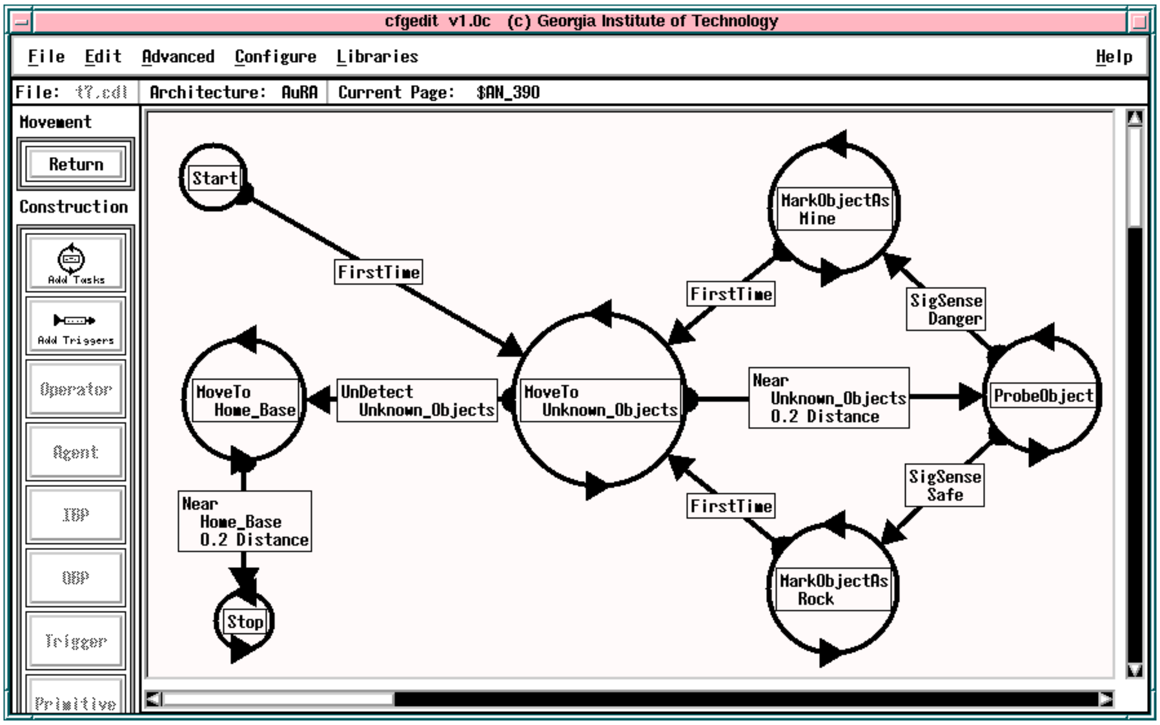


Figure 5.59: An FSA correctly avoiding the *MissionLab* race condition.

A solution which correctly handles this problem is shown in Figure 5.10. In this case the designer has handled the case where the `MoveTo` action fails due to lack of perceivable targets.

This problem is symptomatic of the frame problem, where the representation of the work no longer matches the physical environment. In this case, the designer's assumption that, once an object is perceived, it will remain visible is violated due to the asynchronous nature of the simuator. Since the problem would be even more likely to occur when driving real robots due to the slow dynamics of environmental changes, it must be addressed.

There are several avenues available for tackling this problem. Unfortunately, this problem is endemic to robotics and the "correct" choice must focus more on additional training than any comprehensive support in *MissionLab*.

1. Increase training. One can argue that this is a problem inherent to robotics and the designers need to guard sensorimotor behaviors to handle failure conditions. Figure 5.10 demonstrates such a solution.

2. Provide a termination condition for all actions. The problem in this case is caused by the lack of a test to block until the `MarkObject` action has completed. For example, the `MoveTo` action is generally terminated by a `Near` perceptual trigger. A similar `DoneMarking` test would allow forming a blocking action from the non-blocking `MarkObject`. This solution still requires the designer to remember to guard the `MarkObject` action.

3. Use only blocking actions. A more radical change would be to bundle all motor behaviors with termination perceptual triggers. `MoveTo` would have an extra parameter denoting how close to get to the object before terminating.

The best approach seems to be Choice 2. Choice 3 would expand the number of behaviors and destroy the FSA concept, and Choice 1 is insuffient by itself. In support of Choice 2, the configuration editor could be modified to require that all motor actions have termination conditions attached to them. This would relieve some of the cognitive load from designers by allowing the system to remind them that the `MarkObject` action is missing a terminating check. This, coupled with additional training targeted to the frame problem in robotics, should address the issues raised in the usability experiments.

5.11 Summary

The *MissionLab* toolset was presented as an implementation based on CDL. The graphical-based configuration editor allows the visual construction of configurations by users not familiar with standard programming languages. The compilers then translate these descriptions into executable programs targeted for either the ARPA UGV or AuRA architectures.

MissionLab uses a coherent graphical interface to allow novice users to rapidly gain proficiency while providing a powerful but easy to use environment for both novice and expert users. The interface is written to execute on any suitable UNIX machine. The CDL compiler uses multiple code generators to target disparate run-time architectures.

The Configuration Description Language (CDL) is used as the basis of the graphical editor. The uniform representation provided by CDL simplifies the implementation and allows the editor to use a recursive presentation of configurations.

The Configuration Network Language (CNL) has been developed to support the AuRA architecture. CNL programs are collections of nodes (threads of execution) connected with data flow links. The ARPA UGV architecture using SAUSAGES was chosen as a second target and the SAUSAGES code generator has been proven using the simulator provided by CMU.

The tight coupling provided by the inclusion of a simulation system for AuRA-based robots allows more feedback to the user at run time. The tighter interaction between editor and run-time console allows users to look at the graphic source for configurations while the robots are executing. The next step will be to provide true source level debugging by mapping run-time status information back into the editor.

Several demonstrations of capabilities were included to augment the presentation. These highlighted the retargeting capabilities of the system, the creation of new library components, and the construction and evaluation of multiagent configurations.

Chapter 6

Design of Experiments

When presented with a novel artifact it is natural to ponder its purpose and utility. What is the intended use and how much benefit does a user get from employing it? All change exacts a price, and for a new tool to be accepted it must either empower users to perform tasks heretofore inaccessible, or pronouncedly improve performance over the current state of the art. With this in mind we turn our attention to devising metrics which allow rating and comparing the utility and usability of the artifacts created as part of this research with existing strategies.

The foundation of this research is the **Societal Agent** theory presented in Chapter 3. Unfortunately, theories take the form of paradigms and design guidelines providing insight for how one should think about particular classes of problems and are a bit ethereal for direct experimental evaluation. However, the Configuration Description Language defined in Chapter 4 is a faithful transformation of the theory into a specification architecture. Therefore, instead of directly testing the **Societal Agent** theory, experiments are devised which can be tested using an available implementation of the architecture (presented in Chapter 5) while still reliably extracting those features relevant to the underlying theory itself. Clearly this will require that the experiments be carefully designed and executed so they illuminate the underlying architecture and, thus the theory, and not just facets of the particular implementation. This arms-length analysis may prove less satisfying than grabbing hold of the theory and wringing answers directly from it, but an experimental evaluation requires an implementation to run experiments against.

6.1 Establishing Usability Criteria

Usability experiments measure the performance of people completing certain tasks while using a particular tool. In this document we think of these tools as computer programs; however, there is no reason that such experiments wouldn't provide insight into the layout of paper forms, the design of a phone, and nearly all other tasks where people use tools. In this chapter we will use the term "product" to refer to computer programs presented to people for their use and we will refer to these users as the "customers", since they are the final determinants of the success of the product.

Early in the life of a development effort, metrics must be carefully crafted to capture the utility of the final product. These metrics must then be expounded until the development team understands and accepts them because, without such metrics, there is little hope they will choose a successful path through the myriad of design choices. It is important to have the evaluation process grounded in metrics which allow ranking various possible outcomes at the decision points in the development cycle. For example, the development staff believes that expending two person-months of effort rewriting database interface functions will reduce the time required to look up widgets in the XYZ application by 20%. The question of whether that effort is worth expending is difficult to answer without specific target levels on the user performance. Therefore, it is very important to list the performance metrics which will impact acceptance of the product, and to state minimum levels for them, below which the product will not be accepted. Table 6.1 is an example technique for presenting the usability metrics from [31].

Table 6.1: An example usability criteria specification table for some indeterminate task (After [31], page 223). Notice that `Usability Attributes` are vague high-level concepts while the `Values to be Measured` are concrete performance metrics. The `Current Level` shows the average user performance on existing systems. The `Worst Acceptable Level`, the `Target Level`, and the `Best Possible Level` are predictions of the performance of users on the new system.

Example Usability Specification Table					
Usability Attribute	Value to be Measured	Current Level	Worst Acceptable Level	Target Level	Best Possible Level
Novice performance	Time to perform action A	Hours	30 minutes	20 minutes	5 minutes
Novice performance	Time to perform action B	60 minutes	5 minutes	1 minutes	15 seconds

Notice that each line in the table lists a unique tuple combining an attribute and measurable value ⟨`usability attribute`, `value to be measured`⟩ and specifies target values for that feature. Using a table such as this, the development team can focus their efforts on improving performance in areas that are important, instead of wasting time on improving insignificant aspects. This table also provides criteria to objectively determine when the development process is complete. Once a product achieves all of the minimum acceptable values, it can be considered complete. We now define each of the columns appearing in Table 6.1.

6.1.1 Usability Attributes

The `Usability Attributes` are high level concepts that are deemed important to the customers, such as the performance of new users. Consider our XYZ database example. In this case, a usability attribute might be "The performance of novice users". This high-level concept will reflect things like how easy it is to enter the required color for the widget, database access times, and other factors. Also, what constitutes a "novice user" must be carefully defined. For example, "A novice user has never used the product before but has watched the training video".

The careful selection of attributes is necessary to ensure that all important facets of the human-computer interface are covered. For example, though a lot of attention is normally placed on improving the performance for users familiar with the system, all users begin as novices. Therefore, if the system is too painful for new users to learn, there will be no expert users to consider.

6.1.2 Value to be Measured

The `Value to be Measured` selects a particular aspect of the attribute for which we will specify performance figures. Continuing with the XYZ example, a `Value to be Measured` for the "the performance of novice users" attribute might be the "Time required to look up part numbers".

A particular attribute may have several relevant values which can be used to measure aspects of it. For example, given an attribute such as "novice user performance" there are many values which can be measured to illuminate aspects of the attribute. A small subset includes "time to perform a benchmark task", "time to perform a particular action", and "number of errors while performing a benchmark task". The idea is to take a high-level concept like "novice user performance" and develop concrete metrics that can be experimentally verified and which provide insight into the attribute itself. Of course, a particular value may be relevant to multiple usability attributes.

6.1.3 Current Level

The `Current Level` represents the average performance achieved by the target class of participants using the current state of the art. In cases where users of the existing systems are unable to perform the proposed task, a value of `not possible` can be entered. It is important to list these values to set the threshold the new product must compete with. There is little hope for acceptance if a new product is worse than what the customers are currently using. For our XYZ example, the existing system could be tested with novice users to determine their average performance for the chosen `Value to be Measured`, perhaps 31 seconds on the "Time required to look up part numbers" task.

6.1.4 Worst Acceptable Level

The worst acceptable level sets the minimums for the design process. Any values which are, on average, below this threshold require further refinement before the product can be considered finished. These values are normally close to the current levels since customers won't switch to something clearly worse than what they currently have. These are the best estimates of the levels below which the customers will not use the product.

For the XYZ example, marketing may predict that a performance by novices on the "Time required to look up part numbers" which, on average, exceeds 45 seconds will not be acceptable to the user community.

6.1.5 Best Possible Level

The upper bound on the level of performance that could reasonably be expected is called the `Best Possible Level`. This knowledge is useful to aid understanding of the significance of the performance values. The value should be set to the highest level that could reasonably be expected from users of the system. A useful method to determine these maximums is to base them on the performance of members of the development team using the system. It is unlikely that a user will ever be as familiar with the system as its designers and, therefore, their performance is likely to be less. For our XYZ example, the time required by experienced users would likely be a good estimate of the upper bound on the performance of novice users. Similarly, the performance of the design team would be a good model for the upper bound on the experienced users.

6.1.6 Target Level

The target levels define what the designers should be striving towards. These goals can be set based on market surveys, predicted customer needs, and other relevant information. Normally this value would be set last, after the `Best` and `Current` values are available to provide guidance. It is important that the development team has some input into these levels to ensure they are realistic and achievable with the available level of personnel and technology. It does little good to set targets that are out of reach.

Considering our XYZ example, assume that the `Best Possible Value` for the "Time required to look up part numbers" by novice users is 7 seconds and the current performance level is 31 seconds. Using this information we would be looking to set the target level in the 7 to 31 second range. A final target value would be selected using market surveys, feedback from the development staff, and high-level design priority information. In this case, perhaps a value of 25 seconds would be deemed appropriate.

6.2 The *MissionLab* Usability Criteria

The meta-objective of the *MissionLab* evaluation is to answer the question "Does basing a toolset on the **Societal Agent** theory benefit the users?". This abstract question is far too vague and broad to support a direct answer. Instead it must be decomposed into concrete objectives which can be experimentally evaluated. The following specific objectives have been identified as relevant to this issue:

1. Show that it is significantly faster to create robot configurations using the *MissionLab* toolset than writing the corresponding C code.
2. Show that the *MissionLab* toolset is well suited to the configuration design task.

Given these objectives, the following usability criteria were developed to rate the usability of the *MissionLab* configuration editor for specifying robot missions.

1. `Time to add a mission step`
   The time required to add a new step to a mission is an important determiner in how long it takes to construct missions from task descriptions.
2. `Time to specialize a step`
   The time required to change the behavior of a step in a mission sequence is a predictor of the time required to modify existing configurations.
3. `Time to parameterize a step`
   The time required to change the parameters used by a mission step also impacts utility of the toolset.

4. `Time to add a mission transition`
   The time required to create a new transition between two operating states in a mission sequence gives an indication of how easily the user is able to manipulate the configurations.

5. `Time to specialize a transition`
   The time required to change the perceptual trigger causing a particular transition in a mission sequence is a predictor of the time required to modify existing configurations.

6. `Time to parameterize a transition`
   The time required to change the parameters used by a perceptual trigger also impacts utility of the toolset.

7. `Number of compiles required to create a simple configuration`
   The number of edit/compilation cycles required to create benchmark configurations measures the level of understanding of the users.

8. `Time to create a simple configuration`
   The time required to create benchmark configurations serves as a yardstick metric, giving a handle on the overall performance of the test participants using the toolset.

9. `Ability to create a configuration`
   A binary metric which catalogs the ability of participants to successfully create configurations using the toolset.

10. `Creation time using` *`MissionLab`* `versus C`
    For the participants also constructing a C version of the configurations, how do times required to create the two configurations compare?

11. `General feeling after use`
    We want test participants to feel comfortable using the toolset and to enjoy using it. This metric attempts to determine how successfully that goal was achieved in practice.

Table 6.2 lists the usability criteria using the tabular form developed earlier. Notice that a large portion of the goals for this project deal with the performance of non-programmers using the system. This reflects empowerment of this group as a primary goal of this research project.

The `Current Level` values are *a priori* estimates based on participants using a traditional programming language. These predictions can be reevaluated using the data gathered from experiment 2 (presented in Section 6.5). The `Worst Acceptable`

`Levels` were picked arbitrarily by the author as estimates of the performance levels below which experienced programmers will avoid using the system. These levels are intended to be slightly lower than the performance of programmers using the C language. The system will be acceptable if experienced programmers suffer only a mild drop in productivity, since the system will also empower non-programmers, as reflected in Attribute 9. For this class of novice roboticists we are looking for a clear improvement, from not being able to specify missions, to the successful construction of robot configurations. The `Best Possible Levels` were determined based on the performance of the developer. These values are likely unapproachable by all but very experienced users. The `Target Levels` reflect the design goals of the project. These numbers were selected as targets for the development effort to provide a clear benefit to users over traditional programming languages.

6.3 Designing Usability Experiments

Once metrics have been specified and the various values selected, it is necessary to determine how data can be gathered to allow measuring the levels for the metrics. This is not an easy task and requires careful planning and execution to prevent bias and noise from swamping the underlying data.

Objective methods for data gathering generally involve test subjects using the system under controlled conditions[48]. Commonly, the software is instrumented to gather keystroke and timing information that will allow determining how the user performed certain tasks. The experiments are best if administered by a third party observer to remove bias and to keep the developers from interjecting knowledge not commonly available. This observer is responsible for logging interesting events in a journal of the experiment. The sessions are also videotaped to provide a method for closer and repeated examination of interesting details (and as a permanent record in case of disagreements with participants). Although these sterile test environments clearly impact participant performance, they do allow objective comparisons between competing techniques.

Gathering the data using objective methods is clearly preferable, but not always possible. Certain attributes (*i.e.*, initial impression, user comfort, *etc.*) are by nature subjective and best gathered via questionnaires and informal discussions. Of course, the questions must be carefully crafted to minimize sampling bias. The Questionnaire for User Interface Satisfaction (QUIS)[13] has been developed at the University of Maryland as a general purpose user interface evaluation tool and has undergone extensive testing and validation. The QUIS test can provide a starting point to creating a customized test to extract the desired information.

Table 6.2: The *MissionLab* usability criteria specification table.

***MissionLab* Usability Specification Table**						
	Usability Attribute	Value to be Measured	Current Level	Worst Acceptable Level	Target Level	Best Possible Level
1.	Novice user performance	Time to add a mission step	1 Min	30 sec	10 sec	1 sec
2.	Novice user performance	Time to specialize a step	2 min	1 min	30 sec	3 sec
3.	Novice user performance	Time to parameterize a step	1 min	1 min	30 sec	2 sec
4.	Novice user performance	Time to add a mission transition	1 min	30 sec	10 sec	2 sec
5.	Novice user performance	Time to specialize a transition	2 min	1 min	30 sec	3 sec
6.	Novice user performance	Time to parameterize a transition	1 min	1 min	30 sec	2 sec
7.	Novice user performance	Number of compiles to create a configuration	4	5	2	1
8.	Novice user performance	Time to create a simple configuration	20 min	20 min	15 min	5 min
9.	Non-programmer performance	Ability to create configurations	No	Yes	Yes	Yes
10.	User acceptance	General feeling after use	N/A	medium	good	great

We now present three usability experiments which allow evaluation of the attributes in Table 6.2. In Experiment 1 the participants construct a series of configurations to achieve written mission specifications using the graphical configuration editor. Experiment 2 repeats the process for the subset of subjects in Experiment 1 comfortable using a traditional programming language. Since participants conduct Experiment 2 using conventional text editors, it is necessary to exercise care in the experimental procedures to ensure that as many of the usability attributes as possible are being measured. Experiment 3 concentrates on evaluating the hardware binding features of the toolset and many of the usability attributes do not apply.

It is important to note that before conducting experiments such as these involving human subjects, it is necessary to gain approval at most institutions from an oversight organization. At Georgia Tech this is the "Human Subjects Board" and these experiments were approved for this project, by that board, contingent on participants reading and signing the informed consent form reproduced in Appendix A.

We now present the development of the experiments; the procedures to be followed in carrying them out, the nature and type of data generated, and the evaluation methods to be followed in analyzing the data.

Since Experiments 1 and 2 are intended to provide a direct comparison, participants able to program in traditional languages and willing to complete two experiments will be provide the needed comparison. These participants should first be assigned an ordering as to whether they will use the *MissionLab* toolset (Experiment 1) or the programming language first (Experiment 2). There should also be at least a one day break between performance of the two experiments to reduce fatigue.

6.4 Experiment 1: CfgEdit Mission Specification

6.4.1 Objective

Determine the performance of novice and expert users specifying benchmark robot missions using the Configuration Editor.

There are two target audiences for *MissionLab*: Non-programmers who are able to use the toolset, and expert programmers who can successfully utilize both *MissionLab* and traditional programming languages. Test participants are to be drawn from both participant pools for this experiment. This will allow testing both the hypothesis that skilled programmers can utilize the *MissionLab* system with little drop in productivity after minimal training, and that there exists a group of people who can create a *MissionLab* configuration but are unable to construct the corresponding code directly.

To evaluate this research project's military relevance, an attempt should be made to include a number of ROTC students as test participants. This will allow testing the claim that many will be able to modify a *MissionLab* configuration but unable to manipulate corresponding configurations written in traditional programming languages. If a significant number of the ROTC participants are able to use the *MissionLab* toolset, it will explicitly show the utility of this research for the military community.

6.4.2 Experimental Setup

An independent third party observer should conduct and monitor the experiments to ensure impartiality. The experimental procedures, the benchmark tasks the participants will implement, and much of the menuing structure of the configuration editor was developed in cooperation with Erica Sadun while she was a GRA on this project.

Test Environment

1. A small quiet room where participants can be observed unobtrusively.
2. Videotape equipment to record the session.
3. An X Window-based workstation (SUN SPARC 10).

Desired Test participants

The broadest spectrum of people possible should be run through this experiment. How these test participants are chosen as well as their numbers have the largest impact on the significance of the test data. Ideally, a random sample of potential users large enough to ensure statistical significance should be used as subjects.

Unfortunately, the number of test subjects necessary to ensure statistical significance is dependent on the expected variance in the data to be gathered. Therefore, as a first step, the test pool suggested below will provide a starting point to estimate the experimental parameters. The data gathered from this group cannot be assured to be statistically signification *a priori* but, even without those assurances, it should provide insight into whether the claims are supported at all by experimental evidence. The data will also allow refining these experiments and better selection of the subject pool for similar studies undertaken by other researchers.

1. 3 – 6 ROTC students
2. 3 – 6 CS students familiar with C
3. 3 – 6 individuals familiar with the MissionLab toolset
4. 3 – 6 participants with random skill levels

The time requirement for each participant is 2 hours.

Software

1. GNU C compiler version 2.7 or newer
2. MissionLab Toolset version 1.0 with logging enabled

Tasks

1. Deploy a robot to move to a flag, return to home base, and stop
2. Deploy a robot to retrieve mines one by one, returning each to the Explosive Ordinance Disposal (EOD) area. When all the mines are safely collected, the robot should return home and stop.
3. Deploy a robot to retrieve the flag while avoiding surveillance. Allow the robot to move only when the mission commander signals it is safe.
4. Deploy a robot to explore a mine field. Each possible mine must be probed. If it is dangerous mark it as a mine; if it is safe, mark it as a rock. The robot should return home when all unknown objects are marked.
5. Deploy a robot for sentry duty. Chase and terminate any enemy robots, then return to guarding home base.

Programming Model

1. All of the configurations created by the participants are executed in simulation. Since we are gathering metrics concerning the development process and not concentrating on the final configurations, it is felt that little would be gained by imposing the additional complexity required to deploy each configuration on real robots. (Experiment 3 deals with this issue.) This also allows the simulated hardware to be idealized to reduce complexity.
2. The simulated robots possess a complete and perfect sensor model, allowing determination of the identity, color, and relative location of all objects in their environment with a single sensor reading.
3. The environmental objects are partitioned into four physical classes: Fixed, movable, containers, and robots. Each object can be any color although, for these experiments a classification based on color is created and enforced. Mines are orange, enemy robots are red, flags are purple, EOD areas (containers where mines can be placed) are green, rocks are black, trees and shrubs are dark green, home base is a white rectangle, and unknown objects (either a mine or a rock) are brown.

4. When a mine is being carried by a robot or residing within one of the EOD areas it is not visible to any of the robot's sensors. This includes the robot carrying the object and any other robots operating within the environment.

5. To simplify the control software, the robots in this study are idealized holonomic vehicles. This means that they can move in any direction and need not deal with turning radius issues. The system does not simulate vehicle dynamics; only the maximum robot velocity is restricted.

These idealizations and simplifications result in a straightforward programming model presented to the test participants. It becomes easier to explain and for them to understand the requirements without detracting from the validity of the experiments themselves. Since the modifications apply equally to each participant, any resulting bias is eliminated in the comparisons.

6.4.3 Experimental Procedure

The participants are given oral and written specifications for a series of five tasks, one at a time, and instructed to create robot configurations which fulfill those mission requirements.

1. Participants are to read and sign whatever informed consent form is required by the testing institute.

2. Participants are given a tutorial introduction to the *MissionLab* graphical configuration editor. This provides an introductory overview of the *MissionLab* toolset and helps the participants become familiar with the use of the system. A reasonable method is to have the participants construct a simple configuration in cooperation with the person monitoring the experiments. An example task is to cause a robot to move around picking up mines. The observer can assist the participants in completing this task (or something similar), using it to demonstrate features of the toolset.

3. Repeat for each of the 5 tasks:

   (a) Give the participants the next task description and ask them to construct a configuration which achieves it.

   (b) At this point, the observer should leave the room and only offer assistance when the test participants ask for aid. This policy allows the test participants to decide for themselves when they have reached a stumbling block, and keeps the observer from interjecting help when it may not be required. This also allows all help to be logged and attempts to prevent bias from creeping into the experiments from unequal amounts of help being given to certain participants.

   (c) The test participants will use the configuration editor to construct a configuration which performs the desired task, compiling and testing their solutions using the *MissionLab* compilation facilities and simulation system.

   (d) When the user-created configuration correctly completes the task, or the participants are not finished within 20 minutes, the testing observer reenters the room. At this point any questions are answered and, if the participants' solutions are incomplete, they should be corrected and missing portions explained before the next task is introduced.

4. After completing as many of the tasks as possible within 2 hours, the session concludes with a survey.

6.4.4 Nature and Type of Data Generated

Metrics measuring the performance of each participant are gathered by instrumenting the *MissionLab* system, by the experiment observer, via video tape, and through participant surveys. The results are used to determine how the *MissionLab* system performs against the results gathered in Experiment 2, and to evaluate its usability in general. For both Experiment 1 and Experiment 2, at least the following data values are generated for each subject completing one of the 5 tasks:

1. Time expended creating each configuration until first compile.
2. Log of the durations of compilations.
3. Log of the length of intervals between compilations.
4. Number of compilations before running the simulator.
5. Number of compilations after first simulation until each task is completed.
6. Time required to finish each task. If the participant fails to complete the task, the observer estimates their progress towards a solution ($0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}$).

6.4.5 Data Analysis Procedures

The logging data gathered while the participants complete each task can be used to determine values for the usability criteria established in Section 6.2. Figure 6.1

```
          // Information to identify event file //
  0.000: start Session
  0.001: status StartTime "827354547.598"
  0.002: status Task "4"
  0.002: status Subject "0"
          // A new state was added to workspace //
 16.278: start PlaceState "State1"
 16.931: end PlaceState
          // A state was moved to a new location //
 19.905: start Move
 20.797: end Move
          // A transition was added to connect two states //
 21.796: start AddTransition Trans1
 22.859: status FirstState
 23.616: end AddTransition
          // State2 was changed to the MoveTo behavior //
 58.354: StartModify Agent State2 "Stop"
 61.832: EndModify Agent "MoveTo"
          // Unknown objects targeted for MoveTo //
 64.352: StartModify Parms State2 "MoveTo   None"
 67.198: EndModify Parms "MoveTo   Unknown objects"
          // Configuration compiled successfully //
538.415: event StartMake
605.778: event EndMake
          // Configuration executed in environment A //
607.246: start Run
610.132: start mlab
615.508: start PickMap
618.448: status PickMap "../World_A.ovl"
618.448: end PickMap
678.612: end mlab
678.730: end Run
          // The task was completed. //
824.233: end Session
```

Figure 6.1: An annotated portion of an event log from one of the usability experiments. The numbers are the time the event occurred in seconds, relative to the start of the experiment.

presents an annotated portion of an event log generated automatically by the *MissionLab* system while a user constructs a configuration. The log captures the time and duration of events such as adding states and transitions, selecting new tasks and triggers, compiling the configuration, and running the simulator. Gathering logs such as these during the usability experiments allows the determination of average values for the `Values to be Measured` metrics in the usability specification table. This includes the time required to place states and transitions, specialize states and transitions, and change their parameters.

Analysis of the logs will involve parsing them to extract the relevant data points. For example, the time to specialize a step occurs in Figure 6.1 from time 58.354 to time 61.832 in the log. This interval started when the user clicked the right mouse button on state 2 to choose a new task. The user selected `MoveTo` from the popup menu of tasks as the new task to be performed while this state is active, ending the event.

The total time to complete each task is also available from the logs. This will give a direct measure of the "Time to create a simple configuration" `Value to be Measured` and allow comparisons between Experiment 1 and Experiment 2 performance. The "User Acceptance" `Usability Attribute` will be extracted from the post-experiment surveys.

A statistical analysis of the variance in the measured parameters is necessary to determine the significance of the data gathered. Computing this variance will allow researchers to understand to what extent the data is predictive for future research. Comparisons between Experiment 1 and Experiment 2 should be made as paired samples when the same person performs both experiments since the performance of individual subjects likely has more variability than for the same person.

6.5 Experiment 2: Mission Specification using C

6.5.1 Objective

Determine the performance of participants on tasks similar to Experiment 1 when using a traditional programming language.

This experiment is intended to provide data allowing a direct comparison to the data gathered in Experiment 1. Ideally, the same subject pool should perform both this experiment and Experiment 1 in a random order. There should also be at least a one day break between the two experiments. Of course, participants who are not programmers will be unable to perform this experiment. Given the goal of duplicating as closely as possible conditions in Experiment 1, the procedures and tasks are the same as in Experiment 1 except for differences noted below.

6.5.2 Experimental Setup

Same as Experiment 1.

Test Environment

Same as Experiment 1.

Desired Test Participants

Same as Experiment 1, except for the additional restriction that they need to be fluent in the C programming language.

Software

1. GNU C compiler version 2.7 or newer
2. Current versions of `vi` and `emacs` editors
3. MissionLab simulation system Version 1.0

Programming Model

Same as Experiment 1.

6.5.3 Experimental Procedure

Same as Experiment 1, except that the tutorial also presents a library of behaviors and perceptual triggers that can be called from a traditional programming language. Instead of presenting the graphical editor, the mechanics of editing, compiling, and running programs must be presented.

The participants should be given the exact same task descriptions as Experiment 1 and asked to construct configurations by hand to achieve them. Test participants should be allowed to use their favorite text editor to construct the configurations, and they should evaluate their solutions using the same *MissionLab* simulation system as in Experiment 1.

An equivalent set of motor behaviors and perceptual triggers must be provided as callable functions. Effectively, the subject's job is to construct by hand the FSA that CfgEdit generates from the graphical descriptions. This is intended to give programmers every advantage in reproducing the *MissionLab* capabilities. Starting them out with less support would force them to take far longer to create a solution.

6.5.4 Nature and Type of Data Generated

Metrics measuring the performance of each participant are gathered by instrumenting the build and run scripts, by the experiment observer, via videotape, and through participant surveys. There are no event logging capabilities available during the editing process as in Experiment 1. Therefore, the data gathered during this experiment needs to center on logging when they start and stop editing, compiling, and running their configurations. As a minimum, the following data values are generated:

1. Time expended creating each configuration until first compile.
2. Log of durations of compilations.
3. Log of intervals between compilations.
4. Number of compilations before running the simulator.
5. Number of compilations after first simulation until each task is completed.
6. Time required to complete each task. If the participant fails to complete the task, the observer will estimate their progress towards a solution (0, $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$).

6.5.5 Data Analysis Procedures

The data gathered during this experiment will be used to compare the performance of subjects using the C programming language to their performance using the *MissionLab* toolset. The same evaluation techniques are used here as are used on the data gathered in Experiment 1.

6.6 Experiment 3: Configuration Synthesis

6.6.1 Objective

Show the *MissionLab* toolset is well suited to the configuration design task.

The final experiment is intended to demonstrate the multi-architecture support in the *MissionLab* toolset. One of the primary goals of *MissionLab* is to provide a mission specification tool which improves the design process. Reductions in development time and improved design robustness should be realized from the system's support for code reuse, the integrated simulation environment, and the ability to retarget existing configurations to different implementation architectures and physical hardware.

Existing technologies do not support explicit binding and retargeting of configurations to individual robots, so no direct comparison is possible. *MissionLab* is breaking new ground at this point and a rigorous statistical analysis is not needed. Therefore, the experiment will concentrate on exercising the capabilities of the *MissionLab* system for these tasks. Data gathered during the experiments will provide insights into the strengths and weaknesses of the system.

In this experiment the test participants create, from scratch, a configuration using the *MissionLab* toolset The mission is for a single robot to move through a series of waypoints to a goal, where it allows the operator to control it using teleoperation before sending it on to its ultimate objective. A generic configuration is constructed, then bound to the UGV architecture and the corresponding SAUSAGES code is generated. The SAUSAGES simulation system from CMU is used to evaluate the results. The same configuration is then unbound and subsequently rebound to the AuRA architecture. The code generated in this fashion is tested on the *MissionLab* simulation system and also on a physical robot.

6.6.2 Experimental Setup

An independent third party observer conducts, monitors, and evaluates the experiment to ensure impartiality.

Test Environment

1. A suitable location where the configuration can be evaluated on an actual robot.
2. An X Window-based workstation which runs the *MissionLab* toolset reasonably quickly (SUN SPARC 5).
3. A Mobile robot capable of being controlled by the *MissionLab* toolset.

Desired Test Participants

Since this experiment has a much narrower focus than the previous two, a smaller pool of test subjects should still provide interesting results. The suggested subject pool below will provide information which can be used to shape subsequent experiments to further analyze this new domain.

1. 3 – 6 individuals familiar with the MissionLab toolset
2. 3 – 6 individuals with random skill sets

The time requirement for each participant is 3 hours.

Software

1. GNU C compiler version 2.7 or newer
2. MissionLab Toolset version 1.0 with logging enabled
3. The CMU SAUSAGES simulation system

Programming Model

1. The configurations created by the participants will be executed using the CMU SAUSAGES simulation system, the *MissionLab* simulation system, and a real mobile robot (Denning MRV-2).
2. For this test, localization error accumulated by the robots can be ignored.

6.6.3 Experimental Procedure

This experiment is intended to test the utility of the *MissionLab* toolset for the design process. The participants begin with an empty configuration and the library of standard behavioral primitives. They are instructed to create a configuration consisting of a single robot which performs a simple scouting mission.

1. Each test participant creates a generic single robot configuration where the robot moves through a sequence of waypoints, waits for clearance from the mission commander, and returns. The observer is allowed to answer questions asked by the participants during this experiment.
2. Each subject first binds their configuration to the AuRA architecture, then compiles and runs it using the *MissionLab* simulation system to verify its correctness.
3. If a participant is unable to complete this step in one hour, the experiment stops.
4. The participants deploy their AuRA robot executables on a suitable robot and evaluate its performance. Any required modifications from the simulation to the actual robots are carefully noted.
5. If a participant is unable to complete this step in 30 minutes, the experiment stops.
6. The test participants re-bind the configuration to target the SAUSAGES architecture. The resulting code is evaluated using the CMU SAUSAGES simulation system.

7. If a participant is unable to complete this step in 30 minutes, the experiment stops.
8. When the configuration correctly executes in all three modes, the experiment is complete. If the participants are not finished within a total of 3 hours, the task is interrupted and a failure recorded.

6.6.4 Nature and Type of Data Generated

Metrics measuring the performance of each participant are required to allow determining how the *MissionLab* system is used during design. Since the capabilities demonstrated in this experiment surpass those of existing techniques, little comparison is possible. Therefore, only high level data concerning length of time to complete the tasks is gathered.

6.6.5 Data Analysis Procedures

Using the event logs the following features will be analyzed:

1. Number of compilations to complete each task.
2. Number of runs required to complete each step.
3. Changes required in the configuration between target architectures.
4. Total time required to complete the tasks.
5. Help required during the experiment.

The number of simulation runs required provides a measure of the comprehension supported by the configuration. When a participant runs the configuration, they want to see its performance. If participants do not require many runs to complete a task it would indicate that the editor allows them to understand what the configuration will do before it is actually executed.

6.7 Summary

This chapter introduces the process of usability testing for software systems. Further, it defines specific usability criteria which can be used to evaluate the utility of robot programming toolsets in general, and the *MissionLab* system in particular. A usability criteria specification table was presented which established target values for each of the relevant metrics when using the *MissionLab* toolset.

Three experiments were developed to allow establishing values for each of the usability criteria by conducting usability studies with groups of test participants. Experiment 1 requires each participant to use the graphical editor to create configurations to achieve missions described in each of the five task descriptions. Statistics gathered during this experiment via event logs and survey questionnaires will establish values for the usability criteria. Experiment 2 repeats the same tasks, but requires participants to code solutions using the traditional C programming language. This allows a direct comparison to be made between the two methodologies. Experiment 3 asks participants to build a configuration from scratch to perform a navigation task. This configuration is to be bound to an MRV-2 mobile robot and the executables evaluated both in simulation and on a real Denning robot. After rebinding the configuration to a UGV robot, the SAUSAGES output should be evaluated using the CMU simulator. This task evaluates the utility of the retargetability of *MissionLab*.

Chapter 7

Experimental Evaluation

With an implementation in hand and usability criteria established, we now turn to evaluating the *MissionLab* toolset using the procedures presented in Chapter 6. This chapter will present the results of performing the three experiments using the *MissionLab* toolset. The results will be evaluated using the procedures presented in Chapter 6.

During these experiments the configurations created with the *MissionLab* system will be subjectively evaluated as to their quality during the testing process. It is important to show that the configurations created with *MissionLab* are of acceptable quality, although they may be inefficient with respect to configurations hand-crafted by experts using other tools. The configuration editor encourages users to follow the behavior-based paradigm using data-flow style computation. Clearly, there are good points and bad points to every architecture and this one will be no exception. This style of structuring robot control software has proven fruitful in many other robot architectures[9, 3, 14] and it is expected the configurations created will be of acceptable quality, both in terms of resource requirements and run-time performance. However, there certainly exist some $\langle \text{task}, \text{environment}, \text{robot} \rangle$ tuples where expert roboticists could improve performance by hand-crafting solutions following differing paradigms. However, we intend to show that this toolset in the hands of people who are not expert roboticists can be used to create good configurations. In this case "good" means configurations of sufficient quality that they will perform the required tasks.

The *MissionLab* toolset is an integrated development environment which includes a multi-agent simulator and run-time operator console. However, we expect to show that the majority of the development time benefits of the system arise from use of the visual configuration editor and are directly attributable to the underlying **Societal Agent** architecture. If this is supported by experiments, it provides concrete support for the claims made for the architecture and highlight potential benefits for its application in other domains.

If a significant number of participants perform better on Experiment 1 than 2 it will directly show the benefits of using the *MissionLab* toolset over traditional programming languages. Such speedups are anticipated to arise from a combination of two features of the system: A reduction in the depth of understanding required to successfully create a configuration, and an increase in the rate at which a comparable level of comprehension can be reached.

It is predicted that using the configuration editor will allow the participants to more tightly focus their attention on relevant features of the configuration due to the agent-based recursive presentation. This support for information hiding allows sufficient details to be ignored (which must be dealt with in traditional implementations) to produce a significant speedup by reducing both the scope and depth of comprehension required, and thus reduces the time required for successfully completing the tasks.

It is also suggested that the graphical editor increases the rate at which comprehension of the available primitives may take place. The use of iconic representations for complex computational objects presents a high-level view to the user while still providing whatever depth of presentation the user requires through simple mouse clicks. Based on these presentation features, the same depth of detail can be extracted more rapidly from a configuration written using the *MissionLab* system than from a traditional implementation.

Experiment 3 evaluates the *MissionLab* support for hardware binding and associated run-time architectures. A single generic configuration is used to drive a mobile robot, a simulated AuRA-based robot, and a simulated SAUSAGES-based robot (using the CMU SAUSAGES simulator).

The experiments are presented in Sections 7.1, 7.2 and 7.3. Each section begins by describing how the experiment was conducted. Next, the data generated during the experiment is presented in relation to the usability `values to be measured` defined in Chapter 6 with some additional data developed to support further characterization of the *MissionLab* toolset. Finally, an analysis to determine values for the metrics concludes each section. Section 7.4 summarizes the experimental results and concludes the chapter.

7.1 Experiment 1: CfgEdit Mission Specification

7.1.1 Objective

The objective of experiment 1 is to determine the performance of novice and expert users specifying benchmark robot missions using the Configuration Editor. Since non-programmers are an intended target audience for *MissionLab*, a goal of this experiment is to ascertain if non-programmers are able to use the toolset and, if so, to measure their performance. Participants comfortable using traditional programming languages are included to gather performance data related to this class of users.

This data will also be compared to data gathered during experiment 2 to evaluate if experienced programmers can also benefit from the toolset. This will allow testing both the hypothesis that skilled programmers can utilize the *MissionLab* system with little drop in productivity after minimal training, and that there exists a group of people who can create a *MissionLab* configuration but are unable to construct the corresponding code directly.

7.1.2 Experimental Setup

Test Environment

The tests were conducted in the Georgia Tech Usability Lab. A SPARC 10 workstation was moved into the lab for use in these experiments by the lab coordinator, Randy Carpenter. The lab is outfitted with one-way mirrors and video cameras which allow monitoring and recording the experiments from outside the room. Darrin Bentivegna conducted the experiments to ensure consistency and impartiality. All experiments were videotaped, and logging data was gathered both by Darrin and automatically through the *MissionLab* software.

Test Participants

Twelve people participated in this experiment and are identified with a numeric code ranging from 1 to 12. Participants were solicited via an E-mail announcement to all Georgia Tech College of Computing students and personnel. About 1/2 of the participants were paid subjects. The money was used as an incentive to attract people who might not otherwise volunteer.

To evaluate this research project's military relevance several attempts were made to include ROTC students as test participants. This included a short presentation given to the Georgia Tech Army ROTC students encouraging them to try the experiments. However, we were only able to attract one ROTC student, so no conclusions can be drawn in that area.

The skill set of the participants was as follows:

- 1 ROTC student:
  Participant 12.

- 3 people familiar with the MissionLab toolset:
  Participants 2, 5, and 6.

- 4 people with no programming experience:
  Participants 1, 3, 7, 10, and 12 (Note that 12 is also the ROTC student).

- 4 people with programming skills, but no MissionLab experience: Participants 4, 8, 9, and 11.

Software

The tests were conducted using *MissionLab* Version 1.0 with logging enabled.

Programming Model

The programming model presented in Section 6.4.2 was used in this experiment.

7.1.3 Experimental Procedure

The experimental procedures presented in Section 6.4.3 were followed for this experiment. The scripts and reference material used are reproduced in Appendix A.

7.1.4 Raw Data Generated

Figure 7.1 shows an annotated portion of an event log generated automatically by the *MissionLab* system while a user constructed a configuration. The logs can be used to reconstruct the number and duration of many types of events occurring during the experiments. Events include adding states and transitions, selecting new agents for tasks and triggers, parameterizing those agents, and compilation and execution of the configurations. For example, the time to specialize a step (Modify Agent) occurs in Figure 7.1 from time 58.354 to time 61.832 in the log. This interval started when the user clicked the right mouse button on state 2 to choose a new task. The user selected `MoveTo` from the popup menu of tasks as the new task to be performed while this state is active, ending the event. Each of these events is described in the next subsection.

A parsing tool was developed to extract desired information from the logs. It uses the information of how the various events are denoted in the log files to generate statistics related to the named event type. The raw data measuring usage of the configuration editor was extracted using this tool.

7.1.5 Overview of Experimental Results

This experiment required participants to construct solutions, similar to the one shown in Figure 7.2, for each of five tasks. The *MissionLab* system logged various events while the participants worked to allow establishing values for the various usability criteria.

```
         // Information to identify event file //
  0.000: start Session
  0.001: status StartTime "827354547.598"
  0.002: status Task "4"
  0.002: status Subject "0"
         // A new state was added to workspace //
 16.278: start PlaceState "State1"
 16.931: end PlaceState
         // A state was moved to a new location //
 19.905: start Move
 20.797: end Move
         // A transition was added to connect two states //
 21.796: start AddTransition Trans1
 22.859: status FirstState
 23.616: end AddTransition
         // State2 was changed to the MoveTo behavior //
 58.354: StartModify Agent State2 "Stop"
 61.832: EndModify Agent "MoveTo"
         // Unknown objects targeted for MoveTo //
 64.352: StartModify Parms State2 "MoveTo   None"
 67.198: EndModify Parms "MoveTo   Unknown objects"
         // Transition 1 was changed to Detect trigger //
276.181: StartModify Agent Trans1 "FirstTime"
280.518: EndModify Agent "Detect"
         // Transition 1 was changed to detect Mines //
340.983: StartModify Parms Trans1 "Detect   None"
343.992: EndModify Parms "Detect   Mines"
         // Configuration compiled successfully //
538.415: event StartMake
602.616: event GoodMake
605.778: event EndMake
         // The Configuration was executed //
607.246: start Run
678.730: end Run
         // The task was completed. //
824.233: end Session
```

Figure 7.1: An annotated portion of a *MissionLab* event log. Comments are enclosed in `//` `//` brackets. The numbers are the time the event occurred (in seconds) after the start of the experiment.

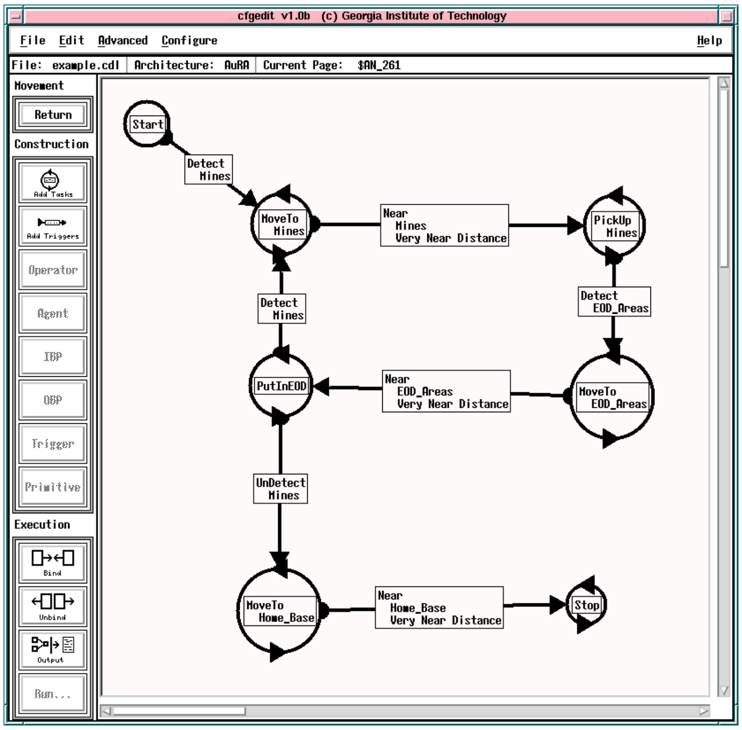


Figure 7.2: A representative task solution for Experiment 1.

Constructing a histogram of the duration of actions allowed visualizing the variability in the data. Using these histograms it was noted that choosing a new behavior for a mission step and a new trigger for a transition both exhibited markedly greater variability than the other actions. This variability suggests that users had difficulty choosing new behaviors and triggers. The current method for selecting behaviors and triggers presents an alphabetized list of choices from which users can pick. These results suggest a different method is required, such as a functional grouping.

In contrast, the `parameterize step` action is very uniform. This suggests that users are able to consistently choose new parameters for behaviors and the current methods using push buttons and slider bars to parameterize behaviors are working well.

The actual values for the usability criteria were estimated using the average durations measured during the experiment. Figure 7.3 presents these results in tabular form. Notice that all times were less than 25% of predicted values. This demonstrates that the system is quite easy for novices to use to construct and evaluate robot missions.

Novice User Performance		
Value to be Measured	*Target Level*	*Measured Value*
Time to add a mission step	10 sec	2.2 sec
Time to specialize a step	30 sec	6.2 sec
Time to parameterize a step	30 sec	4.1 sec
Time to add a mission transition	10 sec	2.6 sec
Time to specialize a transition	30 sec	4.9 sec
Time to parameterize a transition	30 sec	4.0 sec
Number of compiles to create configuration	2	2.0
Time to create a simple configuration	15 min	7.4 min

Figure 7.3: Experiment 1 established values for the usability criteria

Figure 7.4 compares the time taken by non-programmers and experts to construct solutions for each task. Notice that two of the five non-programmers were able to perform close to expert level with only 20 minutes of training. A third was able to complete two of the tasks. The remaining two participants were only familiar with computers in an office environment and struggled with the system. These results suggest that people with an engineering background are able to use the system regardless of their ability to program in traditional languages such as C.

Seconds to Complete each Task						
P	*Task 1*	*Task 2*	*Task 3*	*Task 4*	*Task 5*	*Avg*
1	$\frac{1}{4}$	$\frac{1}{2}$	—	—	—	—
3	$\frac{3}{4}$	$\frac{1}{2}$	—	—	—	—
7	538	570	569	760	399	567
10	234	714	616	557	568	538
12	$\frac{1}{2}$	424	$\frac{3}{4}$	551	—	487*
2	329	$\frac{3}{4}$	408	669	—	469*
4	329	394	522	480	726	490
5	349	494	525	377	441	437
6	319	498	663	551	458	498
8	475	220	283	362	311	330
9	189	520	342	379	215	329
11	234	270	372	332	440	330

Figure 7.4: The top group lists the performance of non-programmers on Experiment 1 and the lower group represents the expert users. The times are seconds required to generate a solution. Fractional values represent partial completion of the task. Notice that 2 of the 5 non-programmers did quite well and a third was able to solve two tasks correctly.

7.1.6 Detailed Experimental Results

The usability criteria established in Figure 6.2 were:

1. `Time to add a mission step`
2. `Time to specialize a step`
3. `Time to parameterize a step`
4. `Time to add a mission transition`
5. `Time to specialize a transition`
6. `Time to parameterize a transition`
7. `Number of compiles required to create a simple configuration`
8. `Time to create a simple configuration`
9. `Ability to create configurations`
10. `General feeling after use`

Each of these criteria will now be evaluated based on the data generated in experiment 1.

Time to add a mission step

Adding steps to a mission is a basic construction task necessary to encode missions. The time required to add a new state gives an indication of how difficult it is for users to build configurations using the graphical editor. The action of adding a step to the mission begins when the user clicks the left mouse button on the `Add State` button. This time is marked in the event logs by a `start PlaceState` event. The action concludes when the user clicks the left mouse button in the workspace to place the new state. This ending time is marked in the event logs with an `end PlaceState` event. Each participant generated a varying number of these actions while completing the five tasks based on how many mis-steps and subsequent deletions they made.

Figure 7.5 shows the durations of all the `Time to add a mission step` actions generated during the experiment. The table is structured with column `P`, the participant's ID number and column `T`, the task number. Due to time constraints, some participants didn't complete all five tasks and the rows for the missing data are not included. The times are presented in chronological order with a particular participant's times read left to right and top to bottom. For example, Participant 12's first time was 1.1 (on task 1) and final time was 0.4 (on task 4).

Notice that the number of events in each row varies. The mission requirements for each task described only the desired behavior of the robot. No hints were given as to how best to structure the solution to achieve a solution. The large variability in the number of events is demonstrative of the differing approaches taken by the test subjects in generating their solutions. Although the number of states and transitions used in solutions varied somewhat from person to person and task to task, even when the final solutions were similar, advanced users tended to use fewer actions than novices to generate the solution.

To better visualize the raw data, Figure 7.6 graphs the times from Figure 7.5 with logs from the different tasks appended temporally into a single graph for each participant. Each of the 12 participants' graphs are stacked on top of the other and separated by horizontal lines. The data for the participants is grouped by skill set. Participants 1, 3, 7, 10, and 12 had little or no programming experience. Participants 4, 8, 9, and 11 had good programming skills, but no experience with the *MissionLab* toolset. Finally, participants 2, 5, and 6 were both good programmers and had previous experience using *MissionLab*.

The vertical axis in Figure 7.6 represents the time, in seconds, the participant took to add a state to the configuration. The horizontal lines mark the zero point and the tick marks are 5 seconds apart. The vertical distance between graphs represents 20 seconds duration. If desired, the exact duration of an event can be determined from Figure 7.5.

One would expect to look at these graphs and see faster (lower) times as the users gain experience with the system. Users should also exhibit less variability as they become more proficient with using the editor. The horizontal span of the graphs represents about an hour (the duration of the sessions) so the users had reasonable exposure to the graphic editor. Looking at Figure 7.6 some users do show this speed-up and reduction in variability, with Participant 7 being the most pronounced. It is also easy to distinguish between novice and experienced computer users, with Participants 7 and 5 being examples of each, respectively.

The distribution graph shown in Figure 7.7 was constructed from the data in Figure 7.5 to aid in visualizing the consistency of the data. The horizontal axis denotes the event duration in seconds with a resolution of 1/2 second. The vertical axis denotes the number of "Add Step" events occurring in Experiment 1 with that duration. Notice the well-defined peak at 1 second duration. The data has a Mode[69] (the most common value) of 1 second and appears to have a nearly normal distribution. Based on this graph, the duration of the add mission steps action for experienced users will likely be near 1 second.

The average value for the `Time to add a mission step` action will be used as the measured value for novice user performance. Based on the 312 data points in Figure 7.5, the average value is 2.20 seconds and the standard deviation is 1.87 seconds. This is a very good time for performing this action. The small standard deviation and limited number of outliers suggests that this action is easily performed with little room for confusion. This is to be expected since it only requires users to click on the new state button and again in the workspace. A target value of 10 seconds had been established and times faster than 30 seconds were considered adequate for novice users. It is estimated that it takes about a minute for a programmer to edit a C file and add a new step to a mission (probably by adding a new case to a switch statement), so there is a large improvement available to users of *MissionLab*.

P	*T*	*Seconds*																
1	1	2.3	4.2	1.9	5.1	2.7	5.2	2.7	1.2	5.2	2.6	1.1						
1	2	2.8	0.9	1.9	1.6													
2	1	1.7	3.3	1.8														
2	2	2.8	4.3	3.9	3.7	3.4	1.7											
2	3	2.5	2.5	3.4	4.1	3.6	4.2											
2	4	0.6	2.8	4.0	7.4	2.0	5.0	2.5										
3	1	2.2	1.6	1.4														
3	2	2.1	2.5	1.9	1.2	0.9	1.0	1.1	1.0	1.5	1.2	1.0						
4	1	2.8	1.5	2.0														
4	2	3.0	3.2	5.3	6.3	3.5	2.0											
4	3	1.0	2.6	2.1	2.2	3.9	5.5											
4	4	1.3	3.5	3.6	4.0	1.6	8.3											
4	5	2.8	2.9	1.6	0.6													
5	1	1.7	1.5	0.8														
5	2	1.2	1.2	0.6	2.1	1.3	0.9	1.4	0.7									
5	3	0.9	1.7	1.4	1.0	0.8	1.1	1.0	0.8									
5	4	1.2	0.7	1.2	0.8	0.4	2.0											
5	5	0.6	0.9	0.9	2.6													
6	1	1.4	1.4	2.3	1.4													
6	2	5.1	2.5	1.9	4.9	1.4	1.3											
6	3	0.9	0.9	0.8	1.8	1.8	1.4	1.4										
6	4	1.0	1.2	2.6	1.3	2.3	0.8											
6	5	1.2	2.0	1.4	2.4	0.9												
7	1	6.3	4.2	1.6	1.6													
7	2	4.9	3.5	3.0	6.6	1.5	1.4											
7	3	14.5	3.1	2.5	3.0	1.7												
7	4	8.2	2.4	4.2	2.6	2.0	1.2											
7	5	1.2	2.1	2.3	2.4	0.9												
8	1	2.2	2.6	2.7	1.9	1.4												
8	2	5.5	1.6	6.4	4.1	1.5	1.4											
8	3	1.6	0.8	3.0	2.1	3.0												
8	4	1.3	1.9	6.2	1.7	6.6	0.9	1.2										
8	5	1.3	1.7	1.8	1.2													
9	1	2.5	2.4	1.9	2.6													
9	2	1.1	8.8	2.3	3.0	2.3	1.1	2.6	1.2									
9	3	1.4	1.3	1.8	1.5	1.8	2.3											
9	4	1.0	0.8	0.8	1.3	0.9	0.8	1.1										
9	5	1.1	1.2	0.9	1.2													
10	1	5.5	7.2	1.9														
10	2	2.2	6.4	2.1	0.9	0.9	1.9	1.8										
10	3	5.1	1.9	2.1	1.3	1.9	1.2	1.2	1.0									
10	4	0.7	17.1	1.7	1.3	1.6	1.5	1.2										
10	5	1.6	1.0	4.0	2.9	3.3	1.4											
11	1	1.3	0.7	0.9														
11	2	1.5	4.5	1.3	1.1	2.2	0.8											
11	3	1.1	0.9	3.1	1.6	1.6	1.3											
11	4	1.3	1.6	1.0	0.9	1.0	1.9	1.4										
11	5	0.9	0.7	3.1	0.8	3.3												
12	1	1.1	1.8	1.8	7.1	1.1	1.0											
12	2	1.3	1.2	1.4	2.0	1.7	2.0	1.6	1.4	0.7								
12	3	0.9	1.3	0.8	1.0	0.9	0.6	0.4	1.1	1.2	1.4	3.3	0.8	0.9	3.1	1.0	1.1	0.7
12	4	0.6	1.1	1.7	1.1	1.0	0.5	0.4										

Figure 7.5: **Time to add a mission step**
Time in seconds taken by participants to add steps to a mission. Column "P" is the participant and "T" is the task. The number of actions varied based on how users proceeded in the development. Tasks not attempted due to lack of time are not included.

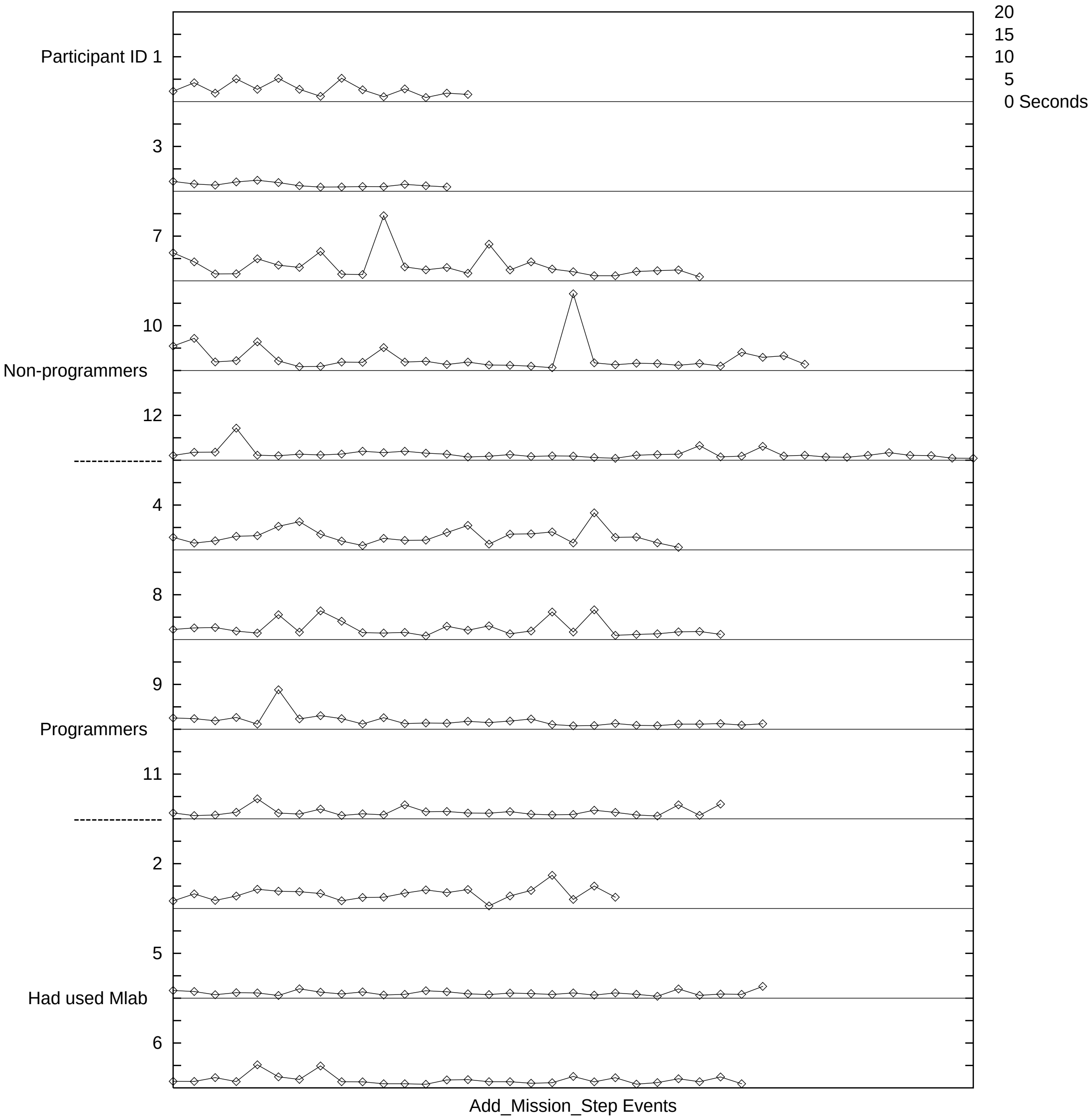


Figure 7.6: **Time to add a mission step**
The data from Figure 7.5 graphed to aid in visualizing trends. The height of each circle marks how long in seconds it took a user to add a step to the mission. Notice that the users tend to get faster and more consistent with experience. The large spikes mark instances where users took longer to decide where to place a state.

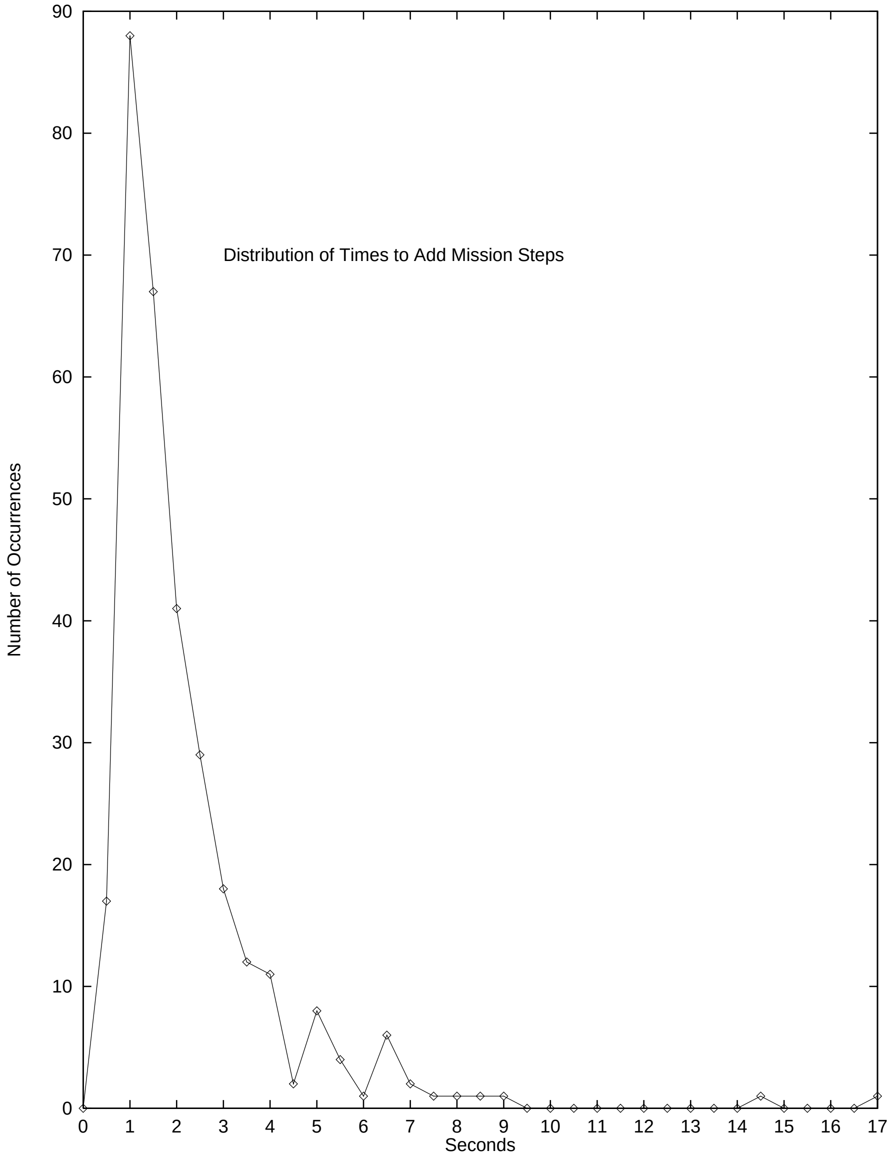


Figure 7.7: Distribution of the time required to add mission steps. The durations from Figure 7.5 were rounded to 1/2 second and the number of occurrences of each duration plotted. This graph helps visualize how consistently users performed this action. The small standard deviation and limited number of outliers suggests that this action is easily performed with little room for confusion. This is to be expected since it only requires users to click on the new state button and again in the workspace.

Time to specialize a step

When a new step is added to a mission it defaults to the `Stop` behavior. Usually this is not the desired behavior and the step must be specialized by selecting the correct behavior from a popup menu. The `Time to specialize a step` attribute measures the time it takes for a user to complete this specialization task. This action is measured from the time the user clicks the middle mouse button on the state until the left mouse button is clicked on the `OK` button in the popup window. These points are denoted in the event logs with the `StartModify Agent` and `EndModify Agent` events.

Figure 7.8 lists the length of time taken by each participant to specialize steps. Notice how much larger the times are and the increased variability compared to the previous action. Figure 7.9 shows this data graphically using the same stacked format as Figure 7.6 to allow better visualization. The vertical scale of the graphs has been expanded to 40 seconds due to the increased variability in this data. Notice how easy it is to discern between experienced and novice users.

Figure 7.10 shows the distribution of this data. Notice that the horizontal resolution of this graph has been reduced to 1 second due to the dispersion of the data. The peak in this graph resembles a normal distribution with a Mode of 3 seconds. This suggests that expert users require about 3 seconds to choose a new behavior for a step. The measured value for the `Time to specialize a step` attribute for novice users is computed as the average of the 260 data points. This works out to 6.15 seconds with a very high standard deviation of 5.86 seconds. The 30 second target value was easily surpassed by these novice users. The estimated time for a programmer to modify a C file to invoke a different behavior was 2 minutes, showing the benefits *MissionLab* users gain.

The long right-hand tail on the distribution graph as well as the variability in Figure 7.9 show a significant difference in performance between novice and expert users on completing this action. Looking at Figure 7.9, Participants 5 and 6 did quite well on this task and generated times consistently in the 5 second range. Compare those records with Participants 7, 10, 11, and 12 who exhibit far greater variability and numerous times in the 20 and 30 second ranges. These long periods are times when the users were confused about which behavior to select. to be useful to people unfamiliar with robotics. of the behaviors were inadequate. The first will be improved by experience using robots and specifying missions and is not directly related to the *MissionLab* toolset. However, unclear behavioral descriptions is a facet of *MissionLab* and the test data suggests they require reworking to be useful to people unfamiliar with robotics.

P	*T*	*Seconds*												
1	1	6.7	6.8	3.5	3.9	4.1	3.2	11.0	2.7	5.4	2.2	2.7	4.6	10.0
1	2	4.3	3.7	4.5	20.0	12.8	39.2							
2	1	34.0	7.4	10.4										
2	2	16.9	2.9	3.4	3.5	14.1								
2	3	3.0	2.8	4.0										
2	4	2.8	3.2	7.2	5.0	2.7								
3	1	24.8	10.1											
3	2	9.5	15.2	7.7	4.4	4.2	3.8	5.5	3.8	3.3	4.7			
		2.8	4.6	3.0	4.2	4.7	3.3	5.4						
4	1	3.3	3.3											
4	2	4.0	3.7	12.8	5.8	5.7								
4	3	3.6	30.5	4.7	2.8	5.0	5.7							
4	4	3.5	6.7	3.8	3.0	3.3								
4	5	5.0	6.9	4.7	4.4	3.0								
5	1	4.6	6.0											
5	2	3.0	2.8	2.2	9.2	3.5	3.8							
5	3	3.7	4.9	2.3	2.6									
5	4	5.9	1.9	3.1	2.4	2.9								
5	5	8.6	4.2	4.7										
6	1	8.2	4.8	3.9										
6	2	4.5	3.9	2.6	4.9	5.7	7.8							
6	3	2.6	2.8	2.2										
6	4	4.9	4.0	4.1	3.2	5.1								
6	5	6.9	4.4	5.9	4.4	2.8								
7	1	18.4	4.6											
7	2	20.8	4.3	3.8	5.7	17.8								
7	3	6.1	4.2	7.6										
7	4	9.7	5.5	9.7	6.6	12.9								
7	5	4.7	5.2	6.1	4.5									
8	1	5.8	4.5	3.5	3.2									
8	2	4.3	2.6	5.2	3.9	3.2								
8	3	4.4	3.1	10.0										
8	4	3.4	4.9	4.5	2.4	2.8								
8	5	3.9	2.8	5.8										
9	1	4.4	2.4											
9	2	3.4	2.5	3.0	4.4	2.7								
9	3	4.9	3.1	2.6	3.1									
9	4	2.4	3.2	4.7	3.0	2.8								
9	5	3.7	15.5	3.1	3.3									
10	1	4.1	11.4											
10	2	17.8	5.0	3.4	5.5	3.9	38.0	3.5	32.3					
10	3	3.6	2.6	4.1	18.8									
10	4	3.5	5.1	3.3	3.2	6.7	2.7	6.9	7.0	6.2				
10	5	2.9	2.5	3.5	3.2	3.4								
11	1	8.7	5.5	8.0										
11	2	8.3	9.9	5.9	12.6	2.9								
11	3	12.6	5.3	2.5										
11	4	2.5	24.8	5.0	6.8	2.8								
11	5	31.1	5.2	2.9	4.1	2.4								
12	1	5.2	6.5	17.3	3.2	2.9	3.7							
12	2	6.1	4.3	2.7	10.2	2.3								
12	3	2.3	2.1	1.8	11.5	2.9	4.4	2.2	2.3	4.6				
12	4	2.6	2.7	7.3	8.5	19.2	4.2	7.7	5.2	3.4	6.7	3.0	4.3	2.9

Figure 7.8: **Time to specialize a step**
Notice that there is more variability in this data than was apparent in the data related to adding new mission steps.

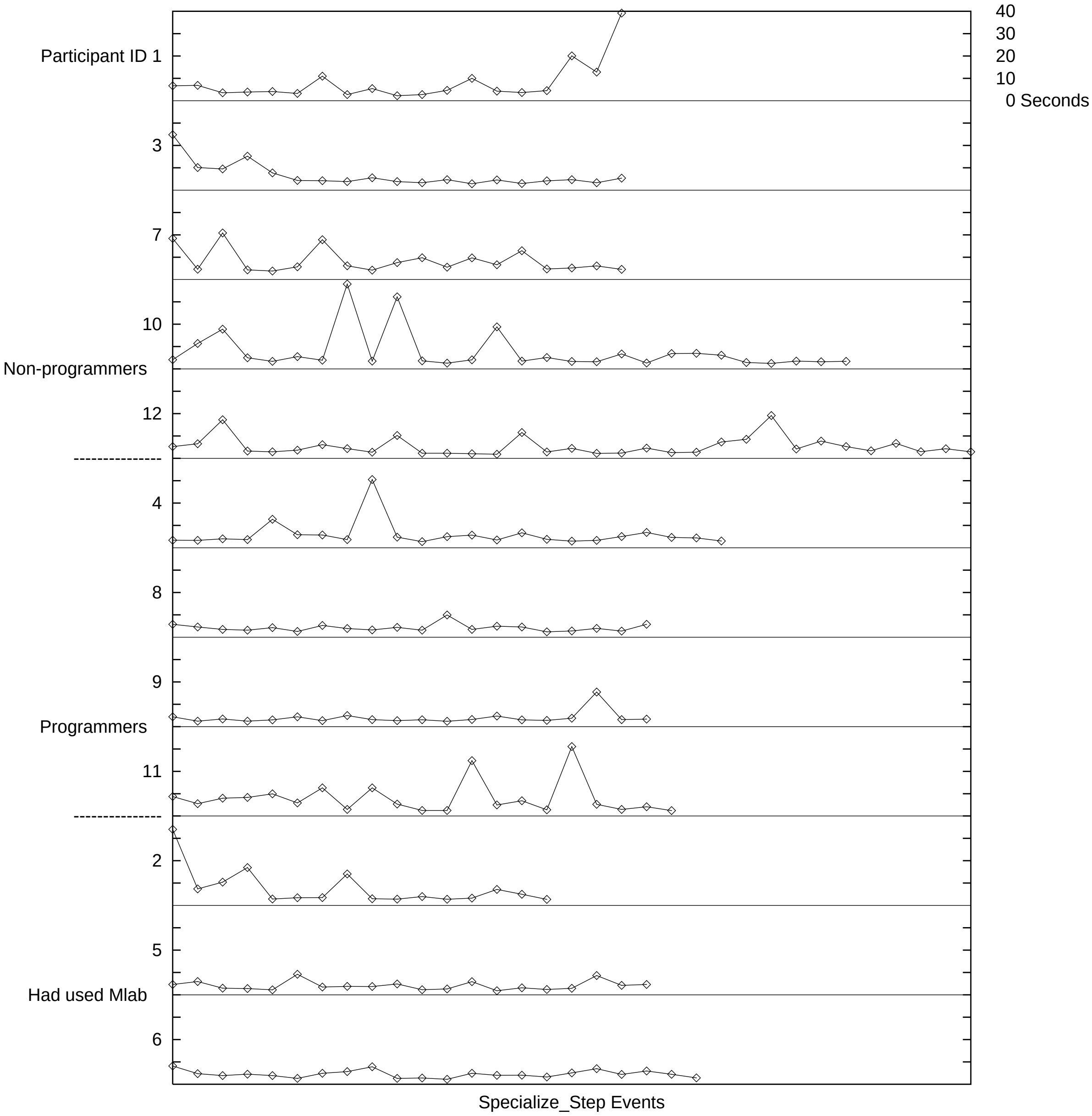


Figure 7.9: **Time to specialize a step**
The vertical scale has been changed to 40 seconds on this graph due to the increased variability compared to Figure 7.6. Notice that there is an apparent difference in performance between experienced and novice users.

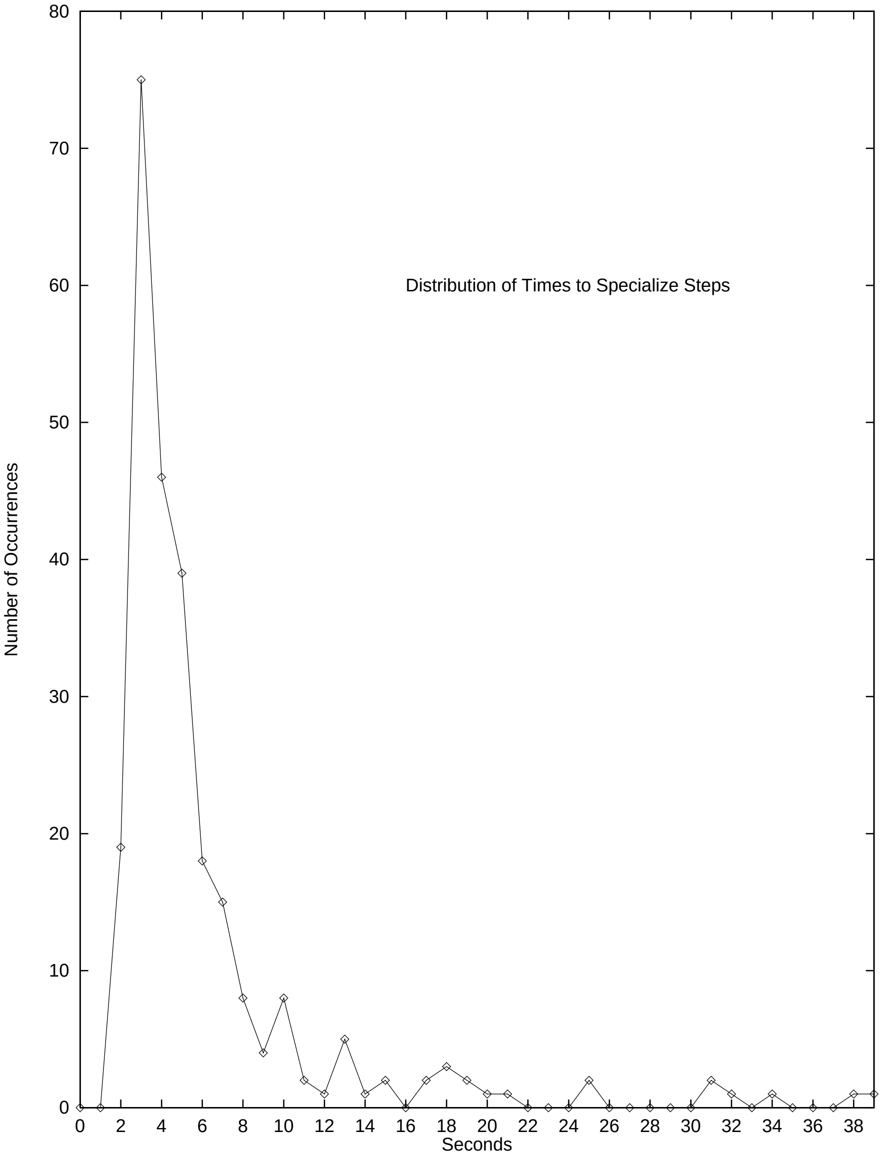


Figure 7.10: Distribution of the time required to specialize mission steps. The Mode occurs at 3 seconds, suggesting that users will be able choose new behaviors for steps in about 3 seconds after they have gained experience with the toolset. The large right-hand tail suggests that some users are having difficulty choosing behaviors. Steps to simplify this process warrant attention in future studies.

Time to parameterize a step

Most behaviors (steps) that can be performed use parameters to further refine their performance. The parameters are modified by opening a popup window and moving slider bars and setting radio buttons to tune the operating characteristics. The `Time to parameterize a step` attribute measures the time it takes users to modify mission step parameters in this fashion. The action is started by clicking the right mouse button on the state and ended by clicking the left mouse button on the `OK` button in the popup window. This duration is represented in the event logs as the time between `StartModify Parms State*` and `EndModify Parms` events (where `State*` denotes the state being modified: `State1`, `State2`, *etc.*)

Figure 7.11 shows how long each participant took to parameterize mission steps. Figure 7.12 graphically presents this data to allow visualizing trends in performance. Notice that these times are relatively consistent except for a few large outliers. Also notice that the spikes appear to be about the same duration.

Figure 7.13 is a distribution graph of the data provided to further characterize the users' performance on this experiment. The Mode (primary peak) at 3 seconds is very well defined and represents the normal performance of this action. However, there appears to be a second, much weaker, peak at 10 seconds and perhaps a third centered at 14 seconds. The task of parameterizing behaviors requires interaction with different types of interface widgets including slider bars and radio buttons. One could posit that the main peak represents interaction with radio buttons and the smaller peaks reflect the extra time required to correctly position slider bars. Unfortunately the event logs are not of sufficient detail to verify such a speculation from the existing data, and the extra peaks may simply reflect noise in the data.

In any case, the average of all the 225 data points gives a value of 4.12 seconds for the `Time to parameterize a step` attribute with a standard deviation of 2.12 seconds. This is much better than the target value of 30 seconds and the estimated 1 minute it takes programmers to modify a C function's parameters. Notice that this data is more consistent than that for specializing a step. This confirms that the methods for setting parameters are easy to understand, while the process of picking new behaviors for states may require further refinement.

P	*T*	*Seconds*													
1	1	5.6	9.8	5.3	5.1	7.8									
1	2	10.3	5.0	4.5	4.1										
2	1	5.6	13.7	4.0											
2	2	4.8	3.7	4.0	4.4										
2	3	5.6	4.8	6.3											
2	4	2.9	10.4	4.4	5.6										
3	1	8.7	4.9	4.5											
3	2	3.5	2.6	3.2	2.4	5.7	3.4	3.2	6.0	3.1	3.0	2.5	5.3	3.2	3.0
4	1	5.9	4.7												
4	2	3.8	2.0	3.5	4.9										
4	3	2.6	2.2	2.7	3.8										
4	4	5.3	4.4	3.5	2.3										
4	5	2.9	3.4	2.5	3.5	4.2	3.3								
5	1	4.2	4.4	1.7	5.3										
5	2	2.5	2.5	2.6	2.3	2.7									
5	3	2.2	1.7	2.8	1.8										
5	4	4.3	4.1	3.2	2.5										
5	5	2.3	2.1												
6	1	3.5	3.1	4.2											
6	2	5.1	3.6	4.2	4.3										
6	3	2.2	3.0	6.9	3.5	5.1	2.3	9.1							
6	4	2.8	4.4	3.0	3.7										
6	5	3.1	3.9	2.5	3.9										
7	1	7.3	4.2												
7	2	4.0	2.7	4.1	3.7										
7	3	3.3	3.1	3.8											
7	4	4.2	6.5	4.1	4.4										
7	5	15.1	4.3	4.4											
8	1	3.7	3.8	3.0	4.4	4.6	2.2	5.4	5.1						
8	2	5.6	2.0	3.2	2.4										
8	3	2.6	3.9	2.5											
8	4	3.1	2.6	3.1	3.3	2.3	11.0								
8	5	2.9	3.5	5.6	2.0										
9	1	3.2	6.3												
9	2	3.9	5.4	3.5	3.2										
9	3	2.3	2.6	10.1											
9	4	2.8	6.7	3.8	3.4	3.5	6.1								
9	5	3.3	2.7	3.2											
10	1	8.1	4.2												
10	2	3.0	9.0	2.8	4.8	4.8	6.5	4.9							
10	3	3.2	3.1	3.1	4.5	4.5	4.4	2.6	3.3						
10	4	2.8	3.3	6.4	3.3										
10	5	2.2	4.2												
11	1	3.6	3.3												
11	2	2.6	2.4	3.8	2.7										
11	3	1.7	4.2	11.2											
11	4	4.3	10.0	3.5	3.4	5.3									
11	5	13.7	2.8	4.5	2.6	2.4									
12	1	3.4	2.5	2.6	2.1	5.2									
12	2	2.6	2.3	3.2	2.7										
12	3	5.3	2.2	3.0	2.4	2.1	3.1	3.3	2.6	2.8					
12	4	3.0	2.9	3.0	5.0	3.0									

Figure 7.11: **Time to parameterize a step**
This data is the basis of the graphs in Figures 7.12 and 7.13. Notice the data appears consistently less than 5 seconds with one or two spikes per participant.

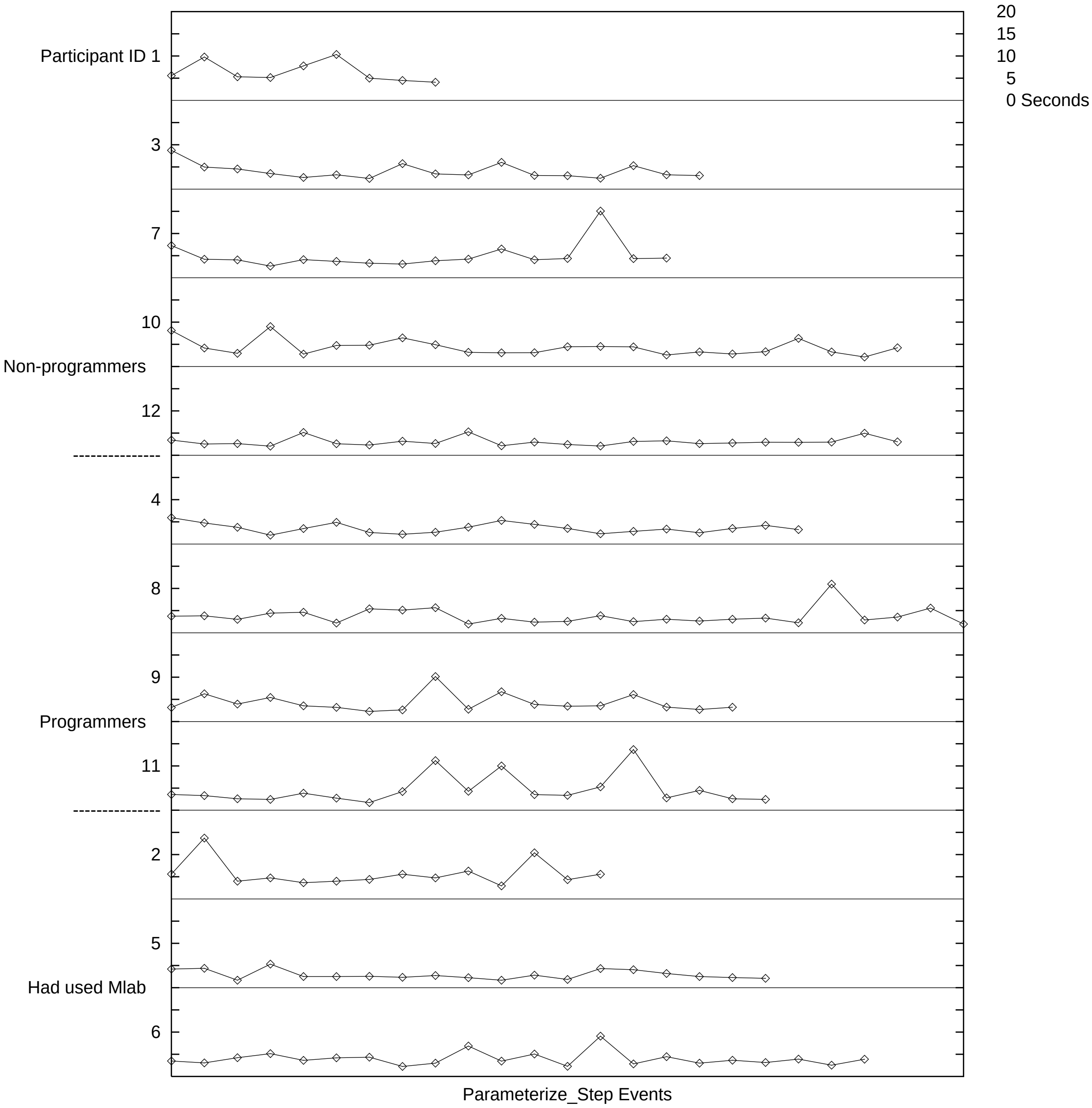


Figure 7.12: **Time to parameterize a step**
The times required by users to parameterize mission steps are graphed to highlight trends in the data. The vertical scale is 20 seconds for this graph. The data is relatively consistent with several large spikes. The spikes don't appear symptomatic of any deficiencies in *MissionLab* and suggest that different aspects of the parameterization task require differing amounts of time.

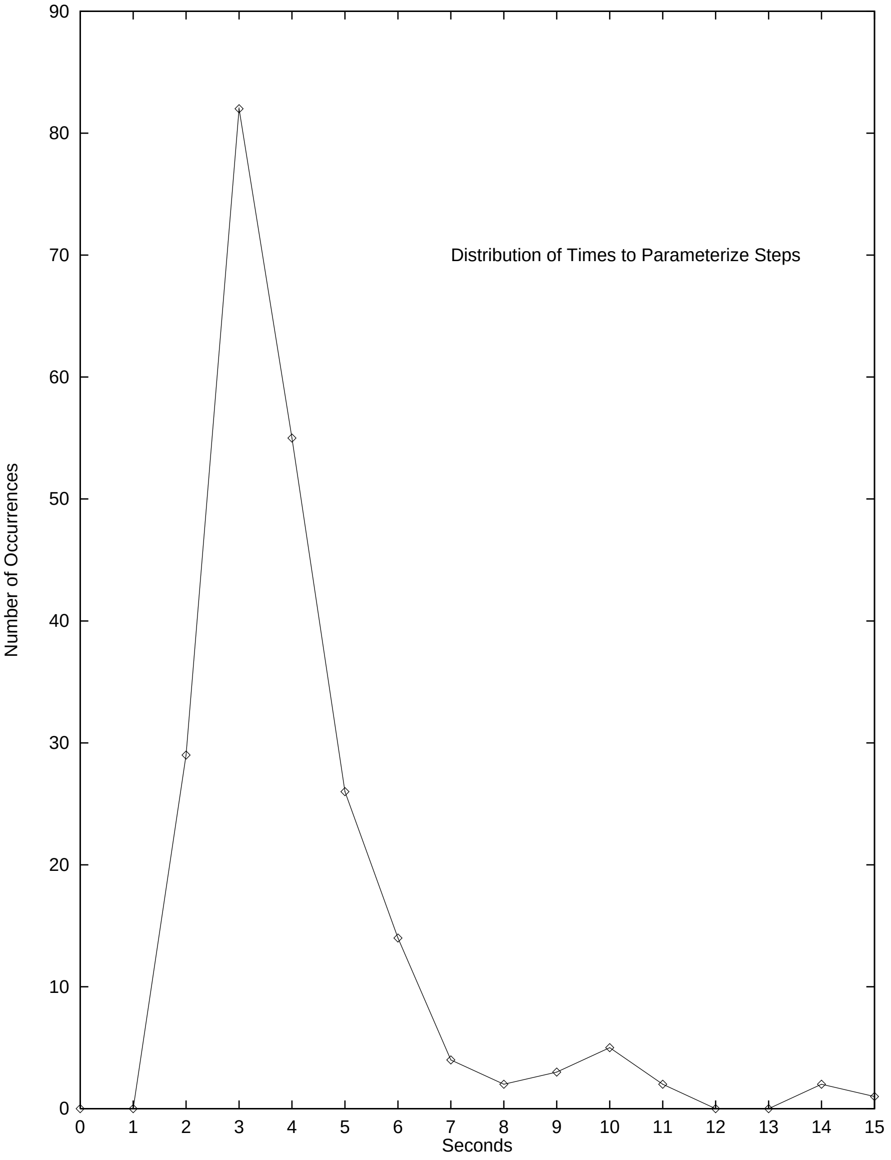


Figure 7.13: Distribution of the time required to parameterize mission steps. The horizontal resolution of the graph is 1 second. Notice the well defined peak at 3 seconds. Users will likely achieve this pace with experience. There appear to be smaller peaks centered at 10 and 14 seconds, possibly reflecting differing interface widgets used to parameterize different behaviors.

Time to add a mission transition

When more than one operating state is established in a mission it is necessary to define when and where transitions will occur between states. The `Time to add a mission transition` attribute measures how long it takes users to add a new transition between states. This action starts when the user clicks the left mouse button on the `Add Transition` button. The left mouse button is then pressed down with the cursor over the source state (`status FirstState` event) and released with the cursor over the destination state to end the action. The duration of this action is extracted from the event logs as the time between `AddTransition Trans*` and `end AddTransition` events (where `Trans*` is the transition identifier: `Trans1`, `Trans2`, *etc.*).

Figure 7.14 lists the duration of these actions for each of the participants and Figure 7.15 shows this data graphically for easier visualization. The data appears consistent with few outliers. Adding new transitions to the workspace is a rather simple task that requires little deliberative effort. The small variability in the data likely reflects the users' inexperience with moving the mouse rather than any aspect of the toolset.

Figure 7.16 is a distribution graph of the data. The well-defined Mode occurs at 2 seconds and suggests the likely performance for experienced users. Averaging the 463 data points used in Figure 7.15 gives a value of 2.60 seconds for the `Time to add a mission transition` attribute. The standard deviation of 1.47 seconds is reasonably small. This very good performance exceeds the target level of 10 seconds established for the experiment and the estimated 1 minute required to add the corresponding transition to a C program.

P	*T*	*Seconds*															
1	1	10.2	1.7	4.7	1.8	4.9	5.2	7.0	4.3	3.9	3.5	2.9	3.0	2.2	1.8	2.0	
1	2	3.9	3.4														
2	1	4.4	2.4	7.5	4.0												
2	2	3.1	3.0	3.4	4.6	6.5	2.6	4.6	3.0	2.4	1.8						
2	3	2.9	2.4	1.1	9.8	2.0	5.0	2.7	2.3	3.6	2.3						
2	4	1.3	1.7	1.7	2.2	2.7	2.3	2.9	2.5	2.7	2.9	2.0	2.0				
3	1	3.0	3.4	2.0	2.2												
3	2	2.6	2.0	2.5	2.4	3.6	1.9	1.8	4.5	2.9	1.3	2.2	2.7	2.4	3.4	4.6	3.4
4	1	1.9	2.2	1.9													
4	2	2.0	2.1	2.6	1.9	4.2	2.5	2.1									
4	3	2.0	2.1	5.9	1.7	2.2	2.3	1.9	3.0	2.6							
4	4	1.5	2.1	1.4	1.8	1.8	2.4	2.5	2.1								
4	5	1.9	1.9	2.3	2.1	4.2	1.8										
5	1	2.0	2.2	3.8	1.8	2.6	2.8										
5	2	1.4	1.8	2.3	1.0	2.0	3.5	2.1	1.5	8.4	2.2						
5	3	5.4	1.8	2.8	3.2	2.0	4.6	2.0	3.1	1.8	1.5	2.7	1.7	2.7			
5	4	1.3	2.3	2.6	1.8	2.1	2.7	1.8	2.0								
5	5	1.3	1.5	1.4	2.7	2.0	1.8										
6	1	3.0	2.4	2.3	2.2												
6	2	1.7	1.4	1.9	2.0	1.8	1.8	2.1	1.4	1.9							
6	3	1.5	1.9	2.2	1.8	10.1	3.0	3.2	1.5	1.3	2.0	2.8	1.7	2.4			
6	4	1.9	2.9	1.6	2.3	1.5	2.0	3.1	2.7	2.6	2.0	2.3					
6	5	1.0	1.3	2.0	2.0	2.0	1.7	2.6	2.4								
7	1	4.2	5.2	5.8	2.5	2.7											
7	2	2.6	2.9	1.9	3.9	2.9	2.2	4.9	2.4								
7	3	6.8	2.0	2.4	2.1	6.2	3.9	4.4	4.9								
7	4	2.1	2.8	3.4	3.4	3.3	3.2	3.2	2.8	3.9	3.5						
7	5	2.1	2.0	2.8	4.8	3.1	1.9										
8	1	2.4	2.5	1.1	2.0	2.1	2.5	2.1	0.9	2.0	2.3	2.0					
8	2	1.9	1.4	1.4	1.4	3.0	1.9	1.8									
8	3	6.0	1.8	1.2	1.4	1.6	1.7	2.8	2.2	2.2							
8	4	0.8	1.5	2.0	2.6	2.1	2.4	1.7	2.1	2.1	1.5	1.7	1.7				
8	5	1.5	1.6	1.8	7.2	1.8	1.5	4.6	8.0	1.5							
9	1	2.6	1.5	3.9	2.6	3.4											
9	2	1.5	4.8	3.2	5.5	4.0	3.5	4.2	2.8	1.4	2.6	4.0					
9	3	2.3	3.4	2.5	2.8	3.9	2.8	2.9	3.0	3.6							
9	4	1.6	1.3	1.7	1.7	1.7	1.7	2.1	1.5	1.0	2.1	2.0	1.2	1.9	2.8		
9	5	2.0	1.9	1.9	1.8	1.8											
10	1	1.7	2.9	2.9													
10	2	2.5	5.5	2.6	3.9	1.9	3.2	2.0	2.3								
10	3	2.2	1.6	8.9	2.5	2.0	4.7	3.3	1.8	2.3	1.9	7.3					
10	4	1.8	2.0	3.3	1.7	1.6	2.2	1.7	2.0	3.1	2.5	4.7	2.0				
10	5	1.4	1.6	1.9	2.7	2.0	2.2	4.8	4.1								
11	1	0.8	1.8	1.8	1.9												
11	2	1.2	1.9	0.8	1.7	2.1	2.4	1.7	2.0								
11	3	1.1	1.6	2.1	2.4	1.6	1.7	1.4	1.8	2.2							
11	4	1.1	1.3	1.4	1.2	1.6	1.7	2.8	1.8	1.9							
11	5	1.2	1.4	1.5	2.1	1.7	2.8	4.4	1.9	2.0							
12	1	4.2	1.7	2.4	1.5	2.0	3.9	2.2	1.9	2.2							
12	2	1.8	1.7	2.1	2.2	2.0	3.2	3.2	1.8	1.6	2.5	3.2					
12	3	1.7	2.3	2.1	1.7	2.3	1.5	2.3	2.0	3.3	11.4	3.1	1.9	1.7	1.5	1.8	
		1.5	4.4	2.5	1.7	2.4	2.2	2.9	2.1	2.0	4.4	2.1	2.8	2.4			
12	4	1.7	1.5	2.7	2.0	12.7	2.2	2.9	2.1	1.8	2.0	1.5					

Figure 7.14: **Time to add a mission transition**
Supporting data for Figures 7.15 and 7.16. Notice the data is very consistent.

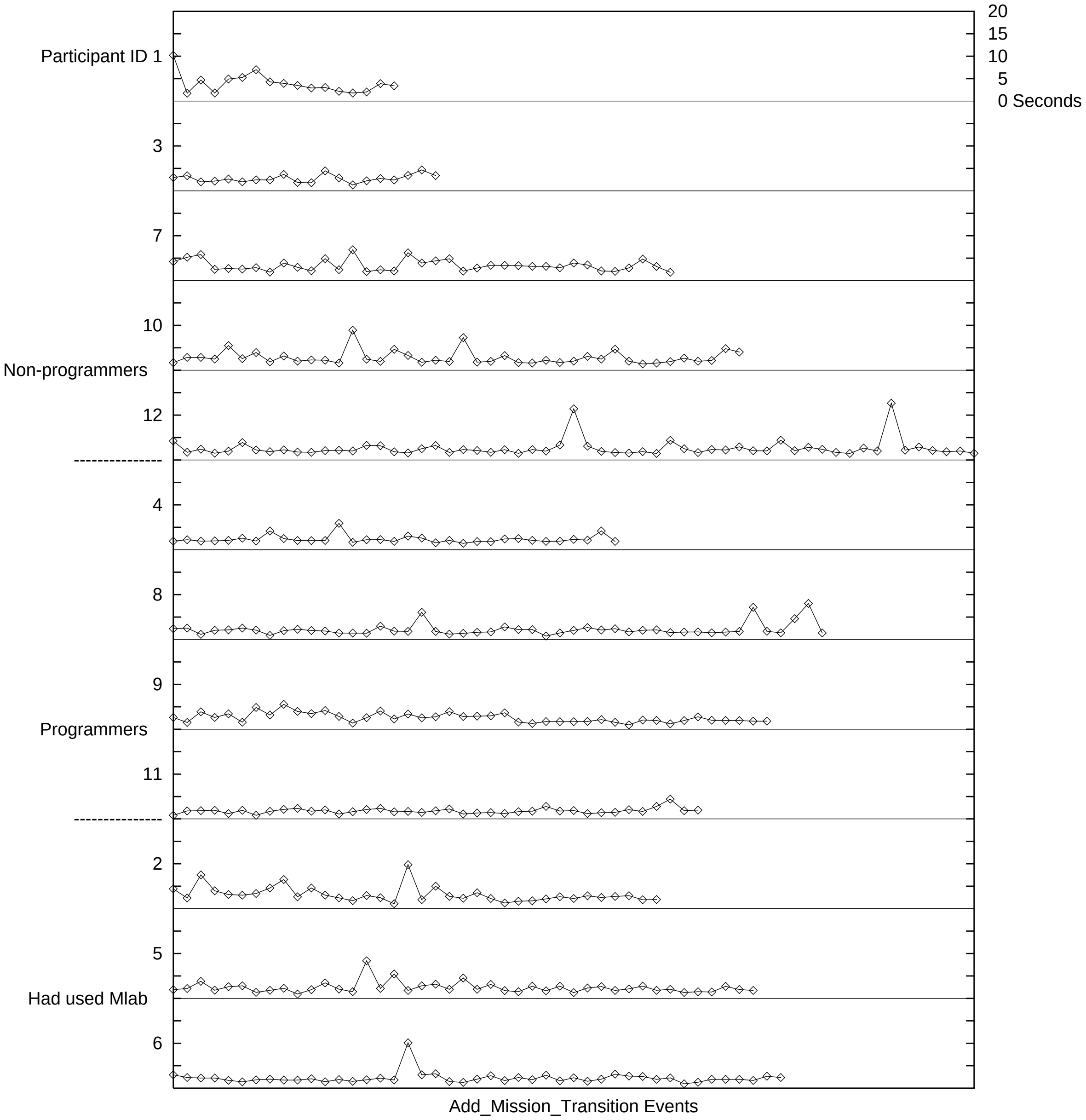


Figure 7.15: **Time to add a mission transition**
The vertical resolution for this graph is 20 seconds. Notice the data is very consistent. Users normally know where they want to connect transitions before they begin the action, so the actual placement is nearly automatic.

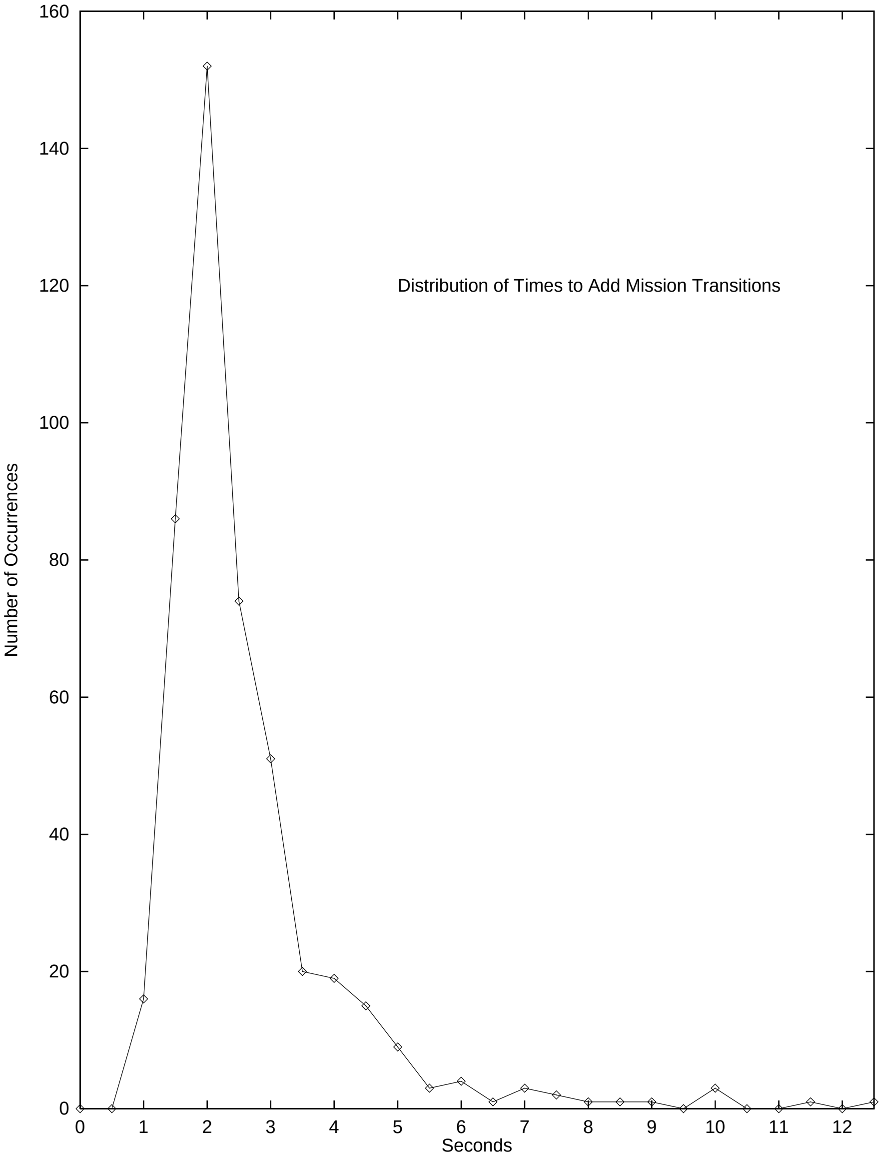


Figure 7.16: Distribution of the time required to add mission transitions. The horizontal resolution is 1/2 second. Notice that the well defined peak appears to match a normal distribution. This is expected since the user doesn't need to make choices during the action. The short right-hand tail further suggests there is little confusion in adding new transitions.

Time to specialize a transition

When transitions are first added to a configuration they default to the `FirstTime` trigger. This causes the transition to be taken the first time after the source state executes. Usually this is not the desired behavior and the operator will specialize the transition by choosing a different trigger. The `Time to specialize a transition` attribute measures how long this procedure takes. This action is measured from the time the user clicks the middle mouse button on the transition and ends when the left mouse button is clicked on the `OK` button in the popup window. This duration is represented in the event logs as the time between `StartModify Agent Trans*` and `EndModify Agent` events.

Figure 7.17 lists the length of time participants took to specialize transitions during the experiment and Figure 7.18 shows the same data graphically. Figure 7.19 is a distribution graph of the data. The graph's Mode occurs at 3 seconds. The very long right-hand tail on the graph raises the average value for the `Time to specialize a transition` attribute to 4.93 seconds with a standard deviation of 4.95 seconds computed over the 398 data points.

This performance is better than the 30 second target value and the 2 minutes a programmer is expected to require to change the trigger causing a transition in a C program. However, the large amount of variability in this data indicates that the process of picking a new trigger for a transition suffers from the same problems as specializing a state. Some of the large values occur at the beginning of a session and the start of a new task, suggesting that users were still designing their solution and that these spikes are unrelated to *MissionLab*. Further study is warranted to determine if the information needed to choose a behavior can be better presented to the operators, based on the remaining variability.

P	*T*	*Seconds*															
1	1	22.6	3.3	4.8													
1	2	11.1															
2	1	8.0	4.4														
2	2	9.4	3.8	3.9	4.4	4.4	3.2										
2	3	6.6	2.9	2.5	2.8	4.6	2.5	4.9									
2	4	2.5	8.8	2.7	3.0	2.6	2.7	2.3	2.9	3.2							
3	1	43.3	5.5	6.2	2.9	2.4	3.0	7.6	7.2	18.8	2.6	5.2	4.1				
3	2	6.0	4.4	16.6	3.1	3.4	3.2	3.7	5.1	3.3	3.3	2.7	4.8	2.8	3.3	5.3	
		2.1	2.8	2.5	3.2	9.9	6.3										
4	1	10.3	5.6														
4	2	3.7	5.2	7.2	5.8	3.7											
4	3	7.7	3.1	3.2	2.5	5.5	8.1	2.4									
4	4	7.4	8.4	3.4	3.2	3.4											
4	5	17.0	2.2	4.0	6.2	3.9	3.2	2.9									
5	1	37.3	3.3	3.4	2.5												
5	2	2.7	2.0	26.4	3.4	2.9	9.8	2.2									
5	3	3.6	2.5	2.4	5.2	2.1	2.2	2.4	2.6	2.6							
5	4	2.2	2.0	2.7	2.8	2.4	3.6	2.7									
5	5	4.3	4.1	4.3	2.8												
6	1	13.8	12.2	4.4	7.3	5.3											
6	2	3.6	3.3	3.1	15.5	2.5	7.0	5.9									
6	3	3.7	3.9	2.6	4.5	5.9	9.0	6.2	12.7	2.7	2.6	4.1	3.1	2.8			
6	4	2.8	2.7	4.6	3.1	3.2	3.1	5.2	4.2	3.7	3.6	3.2					
6	5	6.2	6.1	4.3	3.5	3.6	2.3	7.4	4.0								
7	1	40.8	33.9	5.9													
7	2	9.1	4.8	8.4	7.0	3.2	3.2	4.1									
7	3	7.6	2.8	2.7	6.6	3.7	4.9	10.6	3.1								
7	4	4.4	6.2	3.5	10.2	3.1	3.4	4.1	3.6	5.7	3.2	4.9	4.7				
7	5	4.6	24.8	2.7	4.2	2.6											
8	1	6.0	11.2	4.3	2.8	11.8	2.8	2.4	3.9								
8	2	1.9	2.3	3.1	1.8	2.3											
8	3	5.0	2.8	5.5	3.6	2.7	2.4	2.0	2.3	4.6	2.3						
8	4	3.3	2.2	3.0	3.4	2.2	2.8	4.0	2.2	2.7	3.3						
8	5	2.9	2.7	4.5	3.2	11.0	3.9	3.8	4.4	2.5	2.7						
9	1	7.2	7.2														
9	2	3.4	3.6	5.3	2.4	3.2											
9	3	2.8	2.1	2.7	3.4	3.2	5.0										
9	4	2.4	2.0	2.5	6.7	4.3	4.4	2.0	2.7								
9	5	4.4	4.7	2.4	3.3												
10	1	14.5	11.7														
10	2	2.6	17.4	3.3	7.1	11.7	5.6	3.8	2.9	2.5	3.8						
10	3	5.4	3.1	4.7	7.0	4.7	2.5	4.0	2.8	2.5	2.7	2.9					
10	4	5.1	2.4	7.8	2.4	2.0	3.0	2.8	3.0	3.7	3.0	3.2					
10	5	9.9	3.5	14.4	2.1	4.5	2.6										
11	1	13.0	2.0	17.4													
11	2	2.9	3.3	4.0	10.3	4.5											
11	3	2.9	1.7	3.3	3.4	2.6	2.3										
11	4	2.0	2.7	3.1	3.5	4.0	3.0										
11	5	4.8	11.1	3.8	3.1	2.7											
12	1	25.0	3.9	3.0	3.1	6.7	6.7	4.4	3.3	7.4	2.1	6.7	16.3	3.1	3.0	3.1	
12	2	2.5	2.9	4.6	6.1	16.0	2.4	3.8	2.4	2.0	2.3	3.3					
12	3	2.5	6.3	2.0	2.8	2.3	2.6	3.5	3.2	3.2	2.4	2.4	2.7	2.0	2.3	2.7	3.2
		3.7	2.0	2.9	2.1	2.0	2.7	2.7	2.5	2.4	2.3	2.2	2.2	2.2	2.8	2.9	
12	4	2.2	5.5	2.4	2.5	2.5	2.6	2.9	2.0	1.9	4.5	2.1					

Figure 7.17: **Time to specialize a transition**
The large values occurring at the beginning of tasks can be explained, but the remaining variability indicates users had difficulty choosing the correct triggers.

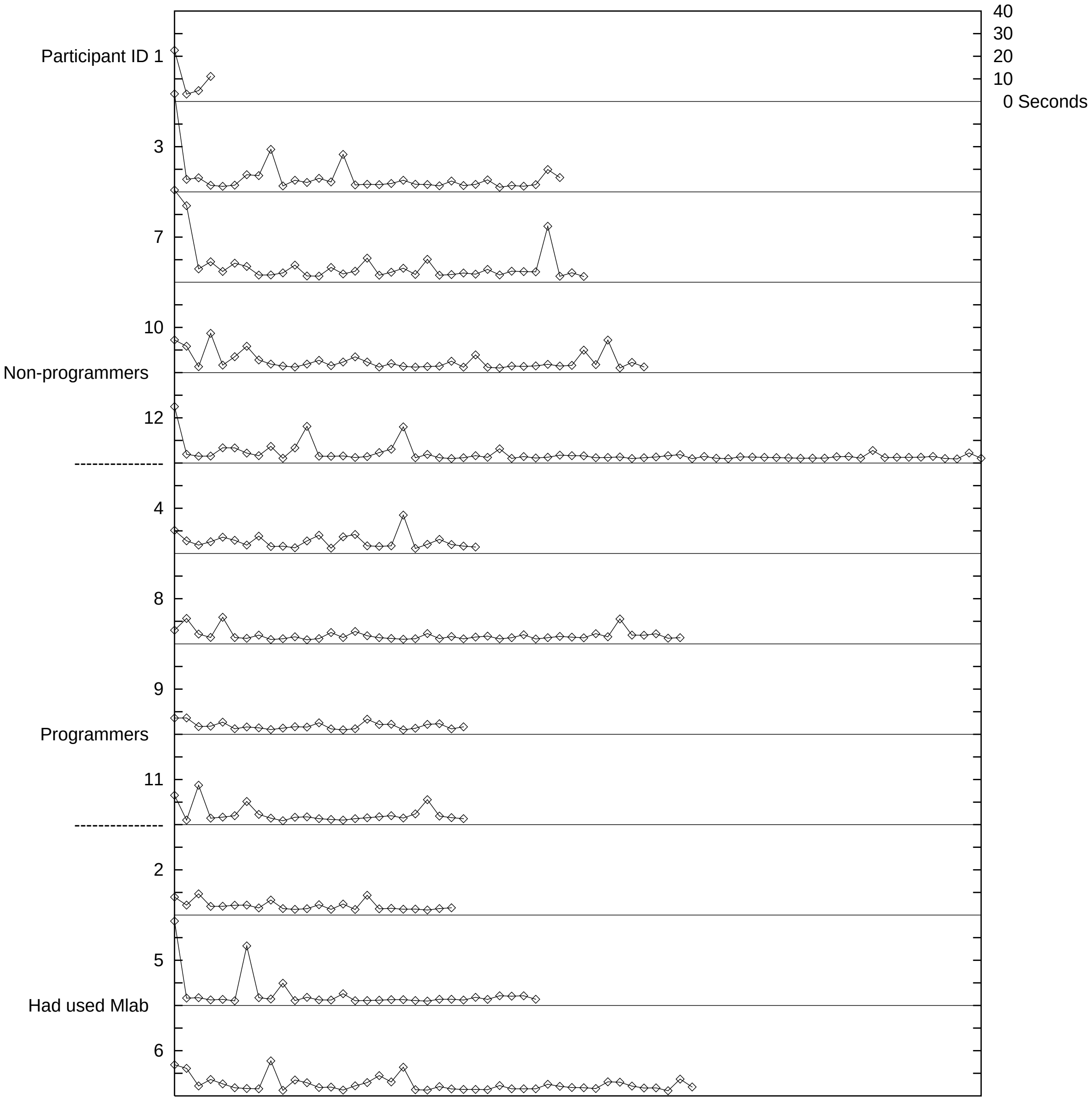


Figure 7.18: **Time to specialize a transition**
Graphical representation of the time required to specialize a transition in a mission. The vertical resolution is 40 seconds.

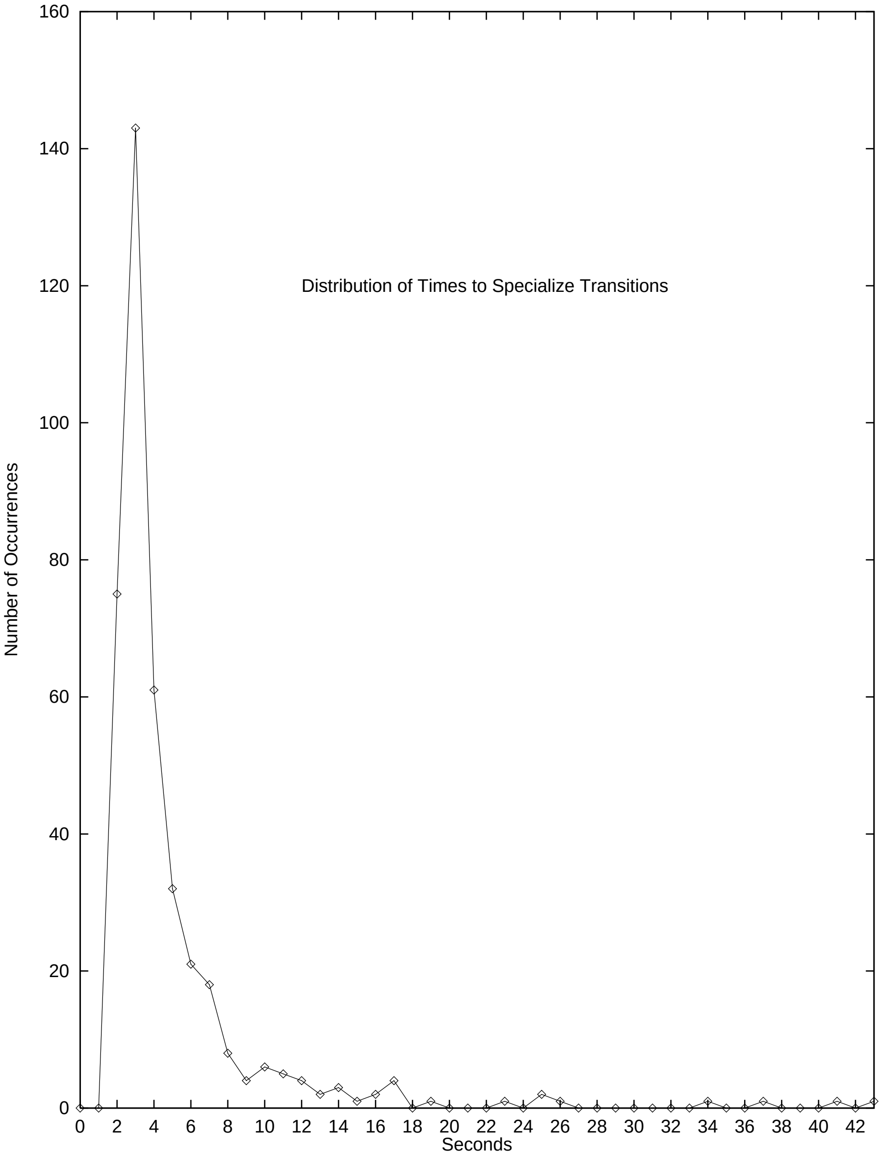


Figure 7.19: Distribution of the time required to specialize transitions for all of the participants. The horizontal resolution is 1 second. The huge right-hand tail indicates that choosing new triggers suffers from the same confusion as specializing states. Both use a similar mechanism which appears insufficient for novice users.

Time to parameterize a transition

The user can set parameters for triggers to constrain when the trigger will cause a transition. They are modified using the same procedure as parameters on states (a popup window). The `Time to parameterize a transition` attribute measures how long it takes users to change these parameters. This action is started by clicking the right mouse button on the transition and ended when the left mouse button is clicked on the `OK` button in the popup window. The duration of the action is represented in the event logs as the time between `StartModify Parms Trans*` and `EndModify Parms` events.

Figure 7.20 lists the duration of these actions during the experiment and Figure 7.21 graphically displays the data. Figure 7.22 is a distribution graph of the data. The Mode occurring at 3 seconds indicates the performance by experienced users. The average value of the 397 data points for the `Time to parameterize a transition` attribute is 4.01 seconds. This exceeds the target value of 30 seconds and the predicted 1 minute required to change parameters in a C program. The standard deviation is high at 2.89 seconds, suggesting users are having difficulty performing this action. There is some evidence of multiple peaks in Figure 7.22 and one could make the same case for disparate controls as was made for parameterizing steps (Figure 7.13). However, it appears that users are also having difficulty choosing parameters for transitions based on the larger magnitude of the spikes.

P	*T*	*Seconds*														
1	1	9.1	14.9	15.5	11.7											
1	2	5.3	8.8	6.3												
2	1	6.4	5.0													
2	2	6.1	2.8	9.1	7.4	3.3	5.6									
2	3	6.0	2.5	2.1	4.8	2.5	2.2	5.4								
2	4	4.5	35.3	2.6	2.3	2.2	2.3	2.6	4.7	5.0	5.1					
3	1	3.7	5.4	4.3	3.3	3.0	3.6	2.3	13.8	4.1						
3	2	4.4	2.8	3.0	1.7	3.0	2.9	4.9	4.1	3.0	3.0	3.4	3.0	2.8	3.4	3.2
		3.3	4.4													
4	1	3.2	4.7													
4	2	3.2	3.1	2.6	3.0	4.2										
4	3	4.9	2.7	1.9	2.6	2.7	2.9	3.5								
4	4	5.6	7.9	6.3	3.0	4.5	3.9	5.4								
4	5	5.8	3.2	3.9	3.0	6.3	3.7	4.0	5.2	5.3	3.1					
5	1	2.7	3.4	2.4	3.4											
5	2	2.3	2.3	2.3	2.3	2.5	2.9									
5	3	4.1	2.1	1.8	2.2	2.2	2.3	3.2	2.3	2.2						
5	4	2.3	1.8	3.6	1.7	2.1	1.6	2.7								
5	5	2.9	2.4	1.9	2.1	2.9										
6	1	9.7	3.8	3.0	3.2											
6	2	3.3	3.3	4.3	2.6	3.3	3.3	6.9								
6	3	2.9	4.1	10.0	2.7	2.7	11.2	2.7	3.3	3.0	2.6	5.4	4.9			
6	4	3.7	2.7	2.8	3.5	3.8	3.8	3.3	2.6	3.5	2.6					
6	5	3.3	2.7	2.5	5.2	2.3	5.8	5.4	8.5							
7	1	3.9	3.9	3.1												
7	2	5.5	3.1	6.3	3.7	5.3	5.8	3.7								
7	3	6.8	3.6	9.1	3.5	3.7	3.5	4.4								
7	4	2.6	20.2	4.1	2.8	5.2	2.7	3.3	4.2	3.1	3.3	17.3	8.3	6.7		
7	5	3.4	3.5	7.7	9.6	3.3	5.0									
8	1	2.6	3.4	3.3	4.9	5.2	3.9	3.0	3.2	3.0	3.0					
8	2	2.6	5.4	3.7	4.7	4.4										
8	3	6.8	2.2	2.0	3.0	2.8	2.2	3.3	2.5							
8	4	2.7	2.8	2.5	2.4	2.5	4.2	2.8	2.5	2.1	4.5					
8	5	2.8	3.4	3.5	2.4	3.5	3.2	2.2	4.7	3.5	3.8	10.5				
9	1	5.3	3.0													
9	2	3.7	2.1	2.3	4.1	5.5										
9	3	3.6	2.3	7.0	3.7	2.6	4.4									
9	4	3.6	2.4	2.3	1.9	3.3	3.1	6.7	3.1							
9	5	3.0	3.3	2.1	3.7	6.9										
10	1	8.0	6.4													
10	2	4.5	2.8	3.1	3.8	4.3	13.4	6.0								
10	3	3.3	3.6	3.1	7.5	5.2	2.9	4.4	11.9	2.9	5.0					
10	4	3.7	3.4	3.6	2.4	3.4	2.6	2.8	2.6	2.5	2.7	2.8				
10	5	4.0	3.7	7.0	3.8	3.3	3.3	2.8	3.1	2.3						
11	1	4.9	4.8													
11	2	2.9	2.4	2.2	2.4	3.0										
11	3	3.1	1.9	3.0	5.1	2.6	2.3									
11	4	2.0	3.0	2.7	2.3	3.4	6.1									
11	5	2.4	5.4	8.4	3.4	2.7										
12	1	7.7	2.8	3.0	3.3	2.5	5.4	3.0	2.1	2.1	2.8	2.5	2.5	2.7	2.8	2.3
12	2	3.7	5.0	3.3	2.1	3.4	4.7	2.3	2.8	2.9	2.2	1.4	2.6			
12	3	17.3	11.3	2.1	1.9	2.0	2.5	2.5	2.5	3.9	2.2	3.4	2.4	1.9	2.0	5.3
		1.9	1.9	3.8	1.8	2.8	3.3	1.8	3.3	2.3	4.8	3.1	2.8	2.3	2.6	4.7
12	4	2.1	2.9	2.3	1.8	2.3	2.3	2.3	3.5	2.1	3.5					

Figure 7.20: **Time to parameterize a transition**
Raw data for Figures 7.21 and 7.22.

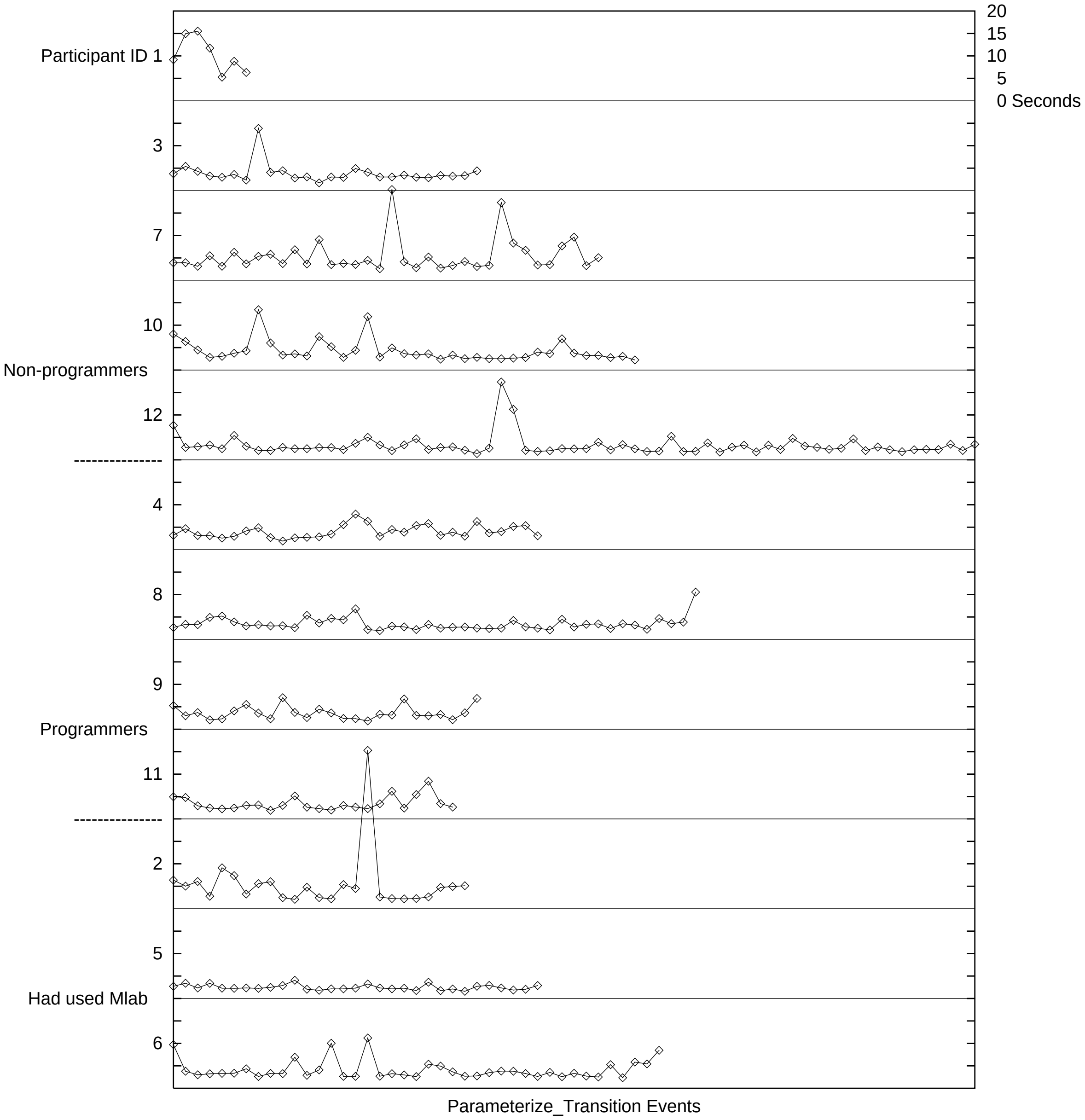


Figure 7.21: **Time to parameterize a transition**
Graphical representation of the time required to parameterize a mission transition (same data as Figure 7.20). The vertical axis represents the duration of the event and the horizontal lines are 20 seconds apart. Notice the large amount of variability present in all logs, including the experienced users.

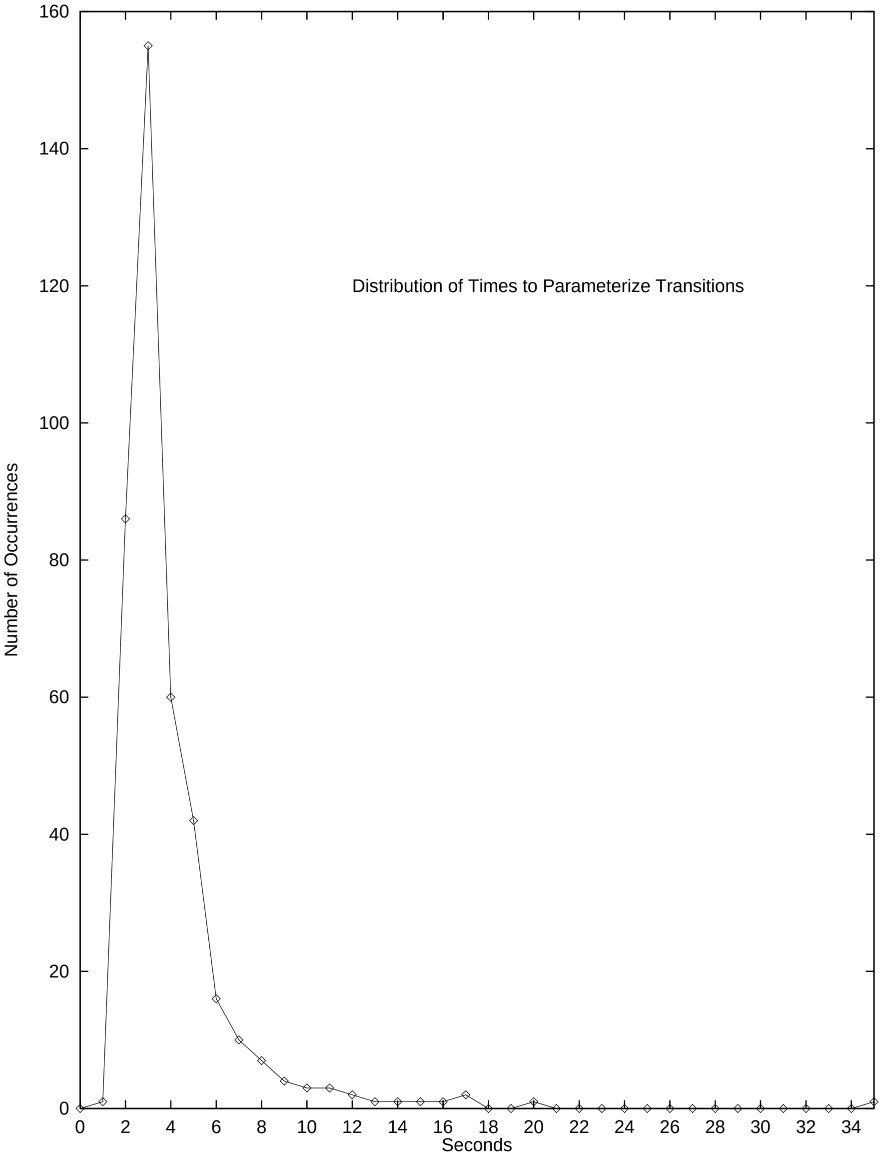


Figure 7.22: Distribution of the time required to change parameters for mission triggers. The number of seconds required is listed on the horizontal axis and the number of occurrences of events with that duration is shown on the vertical axis. The huge right-hand tail suggests participants had difficulty performing this action. The evidence for multiple peaks isn't as strong as in parameterizing steps.

Number of Compilations

The number of compilations each participant made before completing the task is an indication of the difficulty they had in creating a correct configuration. A compilation action is started by clicking the left mouse button on the `Output` button in the editor and is marked in the logs with a `Start Make` event.

Number of Compilations					
Participant	*Task 1*	*Task 2*	*Task 3*	*Task 4*	*Task 5*
1	$*$	$*$	—	—	—
3	$*$	$*$	—	—	—
7	5	2	1	2	1
10	1	2	1	1	1
12	$*$	2	$*$	3	—
4	1	1	1	4	9
8	7	1	3	3	2
9	1	2	1	2	2
11	2	2	1	1	2
2	1	$*$	1	4	—
5	1	1	2	1	2
6	1	1	1	1	4

Figure 7.23: Number of compiles to create a configuration. A dash (—) indicates there wasn't time to work on that task. An asterisk ($*$) indicates that task wasn't completed. The top group of participants are the non-programmers, the middle group were good programmers, and the bottom group were programmers who had previously used *MissionLab*.

Figure 7.23 lists the number of compilations each participant made before completing the tasks. A dash (—) indicates there wasn't time to work on that task and an asterisk ($*$) indicates the task wasn't completed. Figure 7.24 is a distribution graph of the data. Note that data from 7 tasks has been excluded because they were not finished and data from 8 tasks is lacking because they were not attempted due to lack of time. The Mode in the graph occurs at 1 compilation per task which, of course, is the minimum. The average value for the `Number of compiles to create a configuration` attribute computed over the 45 data points is 2.02 with a standard deviation of 1.66 compiles. This matches the target value of 2 compiles per task and surpasses the predicted 4 compiles required for C programmers. Notice only a handful of the sessions required more than 2 compilations. This a satisfying result. The editor appears to allow even novice users to visualize how the configuration will execute before compiling and running it.

Participants 1 and 3 were unable to complete any of the tasks, although they did make substantial progress. These two were administrative people who only used computers in an office environment. They both struggled with the differences between the Motif-based interface used on the SUN workstation compared to their desk-top Macintosh systems. One person didn't understand the concept of building a loop to repeat a task. *MissionLab* users with similar backgrounds will require significantly longer training which also includes an introduction to programming techniques relevant to *MissionLab*.

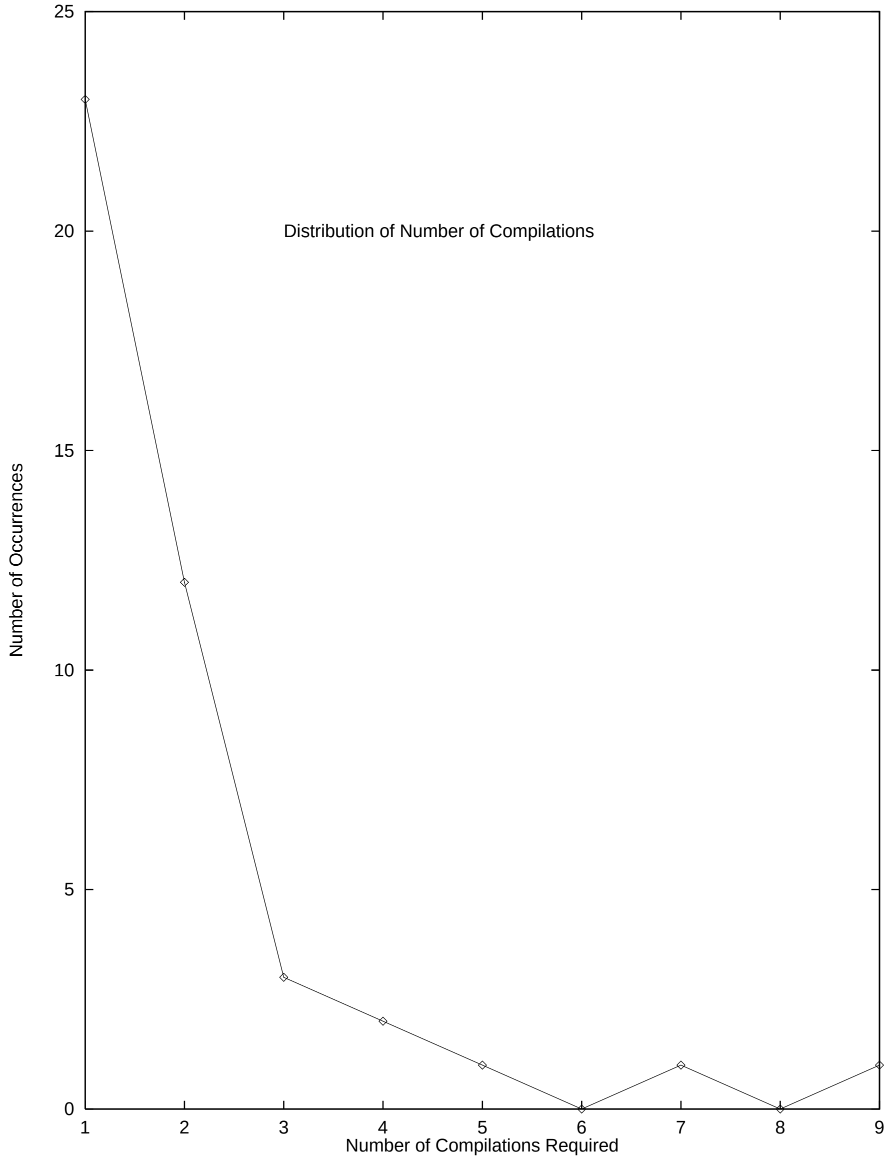


Figure 7.24: Distribution of the number of compilations required to complete each task. Notice that only a handful of sessions required more than 2 compilations. This a satisfying result. The editor appears to allow even novice users to visualize how the configuration will execute before compiling and running it.

Time to create a simple configuration

The most important metric is how long it takes users to construct a configuration which performs the required task. For this experiment, that meant creating a configuration for each task and showing successful execution of the solution in two simulated environments.

The `Time spent editing` will be used as an approximation of this metric. There is sufficient variability in compilation and execution times to dominate variations in the total edit time. There was no consistency in execution times since the simulation system ran until the users manually exited. Compilations took about 45 seconds, but took as much as a minute longer when the network was slow, introducing more variability. Thus, the time spent editing will be computed as the total time for the experiment less the time spent compiling and running the configurations. The `Time spent editing` was computed from the log files as follows: The session lasted from the `start Session` until the `end Session` events. The time between `start Run` and `end Run` events, the time between `event StartMake` and `event EndMake` events, and the time after the final `end Run` event were all subtracted from the total to get the time spent editing the configurations.

Figure 7.25 lists the time participants spent using CfgEdit for each of the tasks in experiment 1. A dash (—) indicates there wasn't time to work on that task. A fraction represents an estimate of how close the participant was to a solution when they were interrupted by the experiment observer due to lack of time, asking for assistance, or being obviously stuck. These times are shown graphically in Figure 7.26. Only those tasks which were successfully completed are graphed. Tasks the participant did not finish or even attempt due to lack of time are not included.

Figure 7.27 is a distribution graph of the data. The Mode in the graph occurs at 400 seconds. The average value for the `Time to create a configuration` is 444 seconds. This was computed from the data in Figure 7.25. The standard deviation is 145.3 seconds computed over the 45 data points. The 444 seconds (7.4 minutes) is half of the target value of 15 minutes to create the tasks and better still than the 20 minutes allotted for C programmers.

Seconds					
Participant	*Task 1*	*Task 2*	*Task 3*	*Task 4*	*Task 5*
1	$\frac{1}{4}$	$\frac{1}{2}$	—	—	—
3	$\frac{3}{4}$	$\frac{1}{2}$	—	—	—
7	538	570	569	760[2]	399
10	234	714	616	557[2]	568
12	$\frac{1}{2}$	424	$\frac{3}{4}$	551[2]	—
4	329	394	522	480	726
8	475	220	283	362	311
9	189	520	342	379	215
11	234	270	372	332	440
2	329	$\frac{3}{4}$	408	669[2]	—
5	349	494	525	377	441
6	319[1]	498	663	551[2]	458

1. Picked up the flag instead of just going near it.
2. Didn't run completely due to race condition (See Section 5.10).

Figure 7.25: Time to create a simple configuration. The fractions estimate progress towards a solution for experiments interrupted by the lab monitor due to lack of time, asking for assistance, or being obviously stuck. The top group of participants are the non-programmers, the middle group the programmers, and the bottom group were programmers who had previously used *MissionLab*.

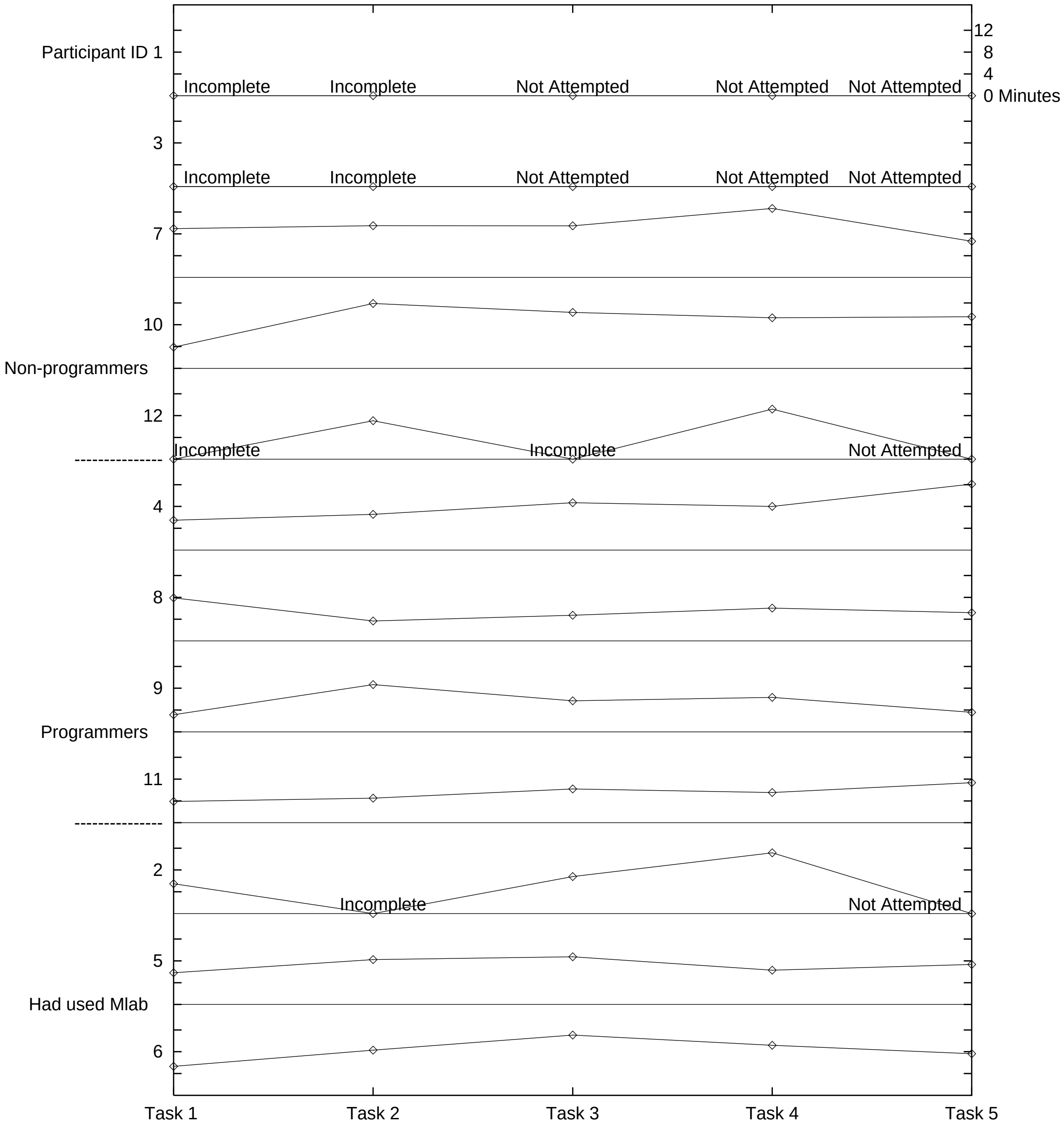


Figure 7.26: **Time to create a simple configuration**
Graphical representation of the time spent editing (same data as Figure 7.25). The tasks are shown along the horizontal axis. The vertical axis represents the duration of the event. The vertical separation is $\sim$ 16 minutes.

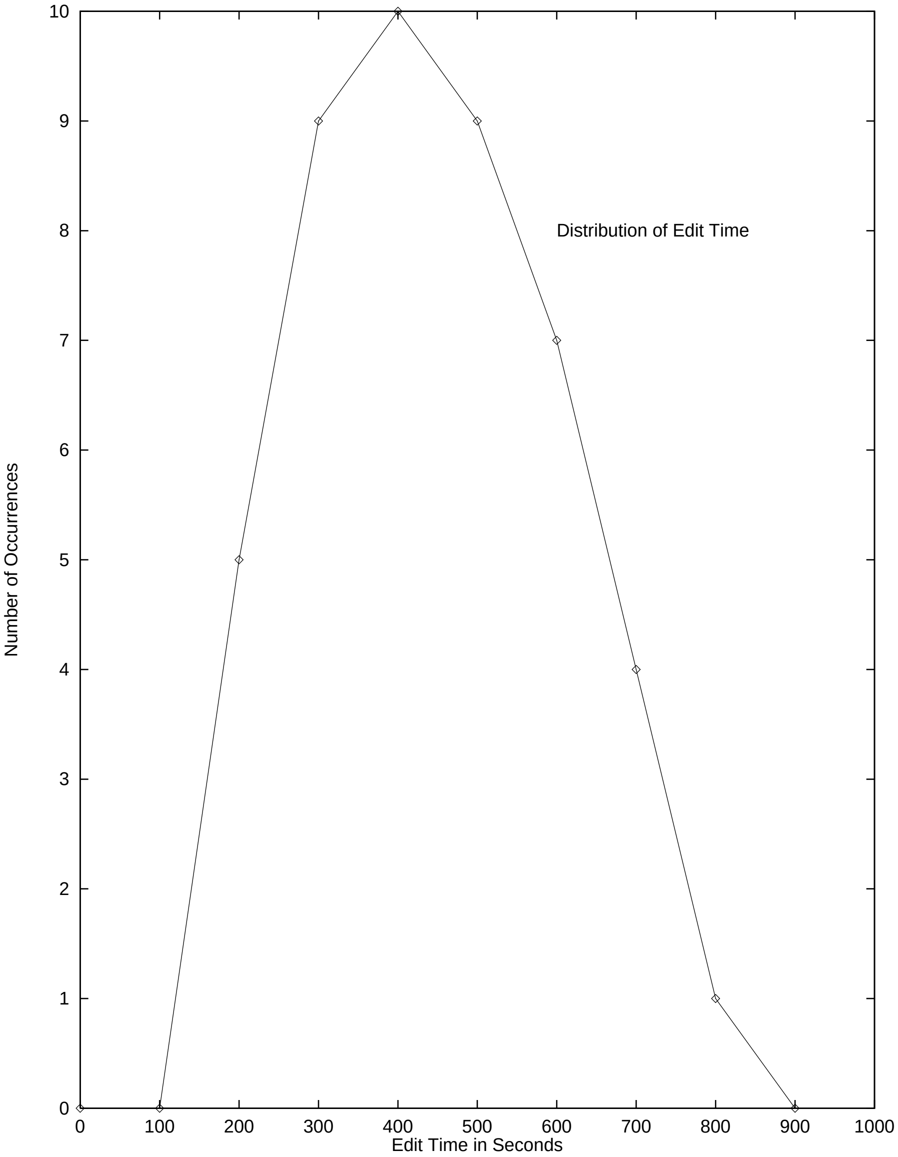


Figure 7.27: Distribution of the time participants spent editing each configuration. The horizontal resolution is 100 seconds and the vertical resolution is one occurrence. The graph is extremely well defined with a single peak and no right-hand tail.

Ability to create configurations

It is useful to partition the participants into three groups. The first natural grouping of the participants will be called the non-programmers. The non-programmers can further be divided into two classes based on their technical backgrounds. People without a technical background are generally familiar with computers based only on their use of commercial software packages such as word processors and E-mail systems. This group (Participants 1, 3, and 12) struggled the most with the user interface. One test participant familiar with one button mice found the three button mouse very cumbersome. A second person didn't think of using a looping construct to repeat an action on multiple objects and instead replicated the specification multiple times, in effect unrolling the loop.

These three struggled with the experiments. Participant 12 finishing two of the five, and Participants 1 and 3 did not complete any solutions. However, most of the incomplete solutions were more than 1/2 complete and made the robot move. One example of the problems faced was trying to decide between using the perceptual triggers "Detect" and "Near" to terminate a "MoveTo" action. Using "Detect" causes a transition as soon as an object of the desired type is detected. "Near" waits until the robot is within a certain distance of the object before causing the transition. The repeated mix-up of these two triggers by the non-programmers points out that a more descriptive help system is necessary.

A second example of the problems faced by this group was in replicating portions of the configuration instead of utilizing looping. This unrolling of loops points to the need for an introduction to basic iterative techniques. Users with no programming background require some basic instruction related to how Finite State Automata function, how one can construct and terminate loops, and other introductory programming information.

The second class of non-programmers were those technically oriented but not able to program in the C language (although some had classroom experience in other languages). These were generally people with engineering backgrounds who never got into programming. Participants 7 and 10 fit this grouping. This is a very important category of users because it represents the target audience for this research. One of the primary goals of the *MissionLab* toolset is to enable people who can not program to specify complex robot missions. As the results for this group show, that goal has been achieved.

The second group consists of the expert programmers. These people were comfortable programming in C and had varying levels of exposure to robotics. They did very well using the GUI and had few problems with the experiments.

The final group included three people who had participated in various portions of the *MissionLab* development. None had used the *MissionLab* GUI before the experiment and their mediocre performances suggest that they didn't gain significant advantage over the programmers.

General feeling after use

The system received great reviews from the users. These statements are from the questionnaire given at the end of the session:

- It was fun and fast.
- The [MissionLab toolset] is straightforward and easy to figure out.
- Easy to specify missions
- It seemed very intuitive and easy to use.
- Very fast start time [with] no learning curve. It deals with the higher levels of a mission, not the specifics of low-level control.

Non-programmers especially commented on how easy it was to tell the robots "what to do" and to check their solution. Some quotes from participants:

- The visual interface makes it much easier to set-up, test, and think about the control loops and decision points; *i.e.*, the logic flow.
- The graphics were very indicative of what an actual mission would look like schematically.
- What I liked about MissionLab were the available range of behaviors and commands.
- The flow-chart type screen used for building missions was very visually appealing rather than only using text.
- The commands were easy to understand and follow.
- It was fun to run [the mission] and see if it worked.

There were some frustrations with the system. Many focused around an apparent race condition in the *MissionLab*. When marking an unknown object as either safe or hazardous, the "Detect Unknown Object" trigger would still fire even if the object just marked was the last unknown object. This issue turns out to be related to the Frame Problem and is discussed in detail in Section 5.10.

Most of the remaining frustrations centered around selecting behaviors for states and triggers for transitions. More effort is needed to identify better names and descriptions for these agents while eliminating any semantic overlap between choices. Some illustrative statements from the participants:

- I think I got caught in a race condition once.
- Not able to see which state the robot is in during simulation.
- Terms are somewhat confusing for beginners.
- Got a little confused at first between add tasks and add triggers.

Experiment 1 successfully demonstrated that the graphical editor is quite usable for constructing robot missions. The participants were excited about the power it gave them to quickly and easily construct robot missions. Their encouraging performance using the toolset shows that *MissionLab* is aiding the development process.

7.2 Experiment 2: Mission Specification Using C

7.2.1 Objective

This experiment was intended to generate data allowing a direct comparison of performance of participants using C and the graphic editor to specify missions. Participants from the same subject pool performed both this experiment and Experiment 1 in a random order. There was a several day break between the two experiments to attempt to minimize the benefits associated with repeating the same tasks. Of course, participants who were not programmers were unable to perform this experiment. The primary goal was to duplicate conditions in Experiment 1 as closely as possible except for the use of the C programming language.

7.2.2 Experimental Setup

Test Environment

Same as Experiment 1.

Test Participants

Each of the people who volunteered for Experiment 1 were asked if they were fluent in the C programming language. Those who were able to program in C were asked if they would be able to take part in two sessions. Three participants (4,5 and 6) could program in C but were unable to participate in more than one session and only completed Experiment 1. Five of the participants (1,3,7,10 and 12) were unable to program in C and therefore didn't complete Experiment 2.

This left four participants (2,8,9, and 11) who completed both Experiment 1 and Experiment 2. The numeric codes assigned to these participants match those from Experiment 1. Two of the participants were randomly selected to complete Experiment 2 before Experiment 1 and the others did Experiment 1 first. Two of the participants were familiar with *MissionLab* and the other two were robotics graduate students, but had not previously used the toolset.

Software

The GNU C++ compiler Version 2.7.2 was used for these tests. A library of functions which mimicked the behaviors available in the graphical editor was created and provided to the participants. A stub program and scripts to build and execute the configurations required the participants only to create a suitable state machine to complete the missions. The software was compiled using the C++ compiler. The library supplied to the participants was structured to allow them to write standard C code if they desired. The standard UNIX text editors `vi` and `emacs` were available for the participants' use. The same *MissionLab* simulation system was used to evaluate their solutions as in Experiment 1.

Programming Model

Same as Experiment 1.

7.2.3 Experimental Procedure

Same as Experiment 1. The additional reference material provided, which explains the function library calling interface, is reproduced in Appendix A, Figures A.15 and A.16.

7.2.4 Raw Data Generated

Due to the use of standard UNIX tools, the ability to automatically log editing events was lost in this experiment. The videotape taken of the experiments was shot over the shoulder of the test participants and not of sufficient quality to recreate their edit session. However, by instrumenting the build and run scripts, some information was still gathered. Figure 7.28 shows an annotated event log from this experiment. The comments are enclosed in `// //` brackets. The start of the experiment is logged, along with the task number. The start and end times for each compile are logged.

```
// Started the experiment //
Wed 10:56:37 AM, Mar 20 1996
         Starting task 3

// 1st build of the solution //
Wed 11:03:28 AM, Mar 20 1996
         Start make

Wed 11:03:36 AM, Mar 20 1996
         End make

// 1st build of the solution //
Wed 11:04:07 AM, Mar 20 1996
         Start make

Wed 11:04:11 AM, Mar 20 1996
         End make

// 2nd build of the solution //
Wed 11:05:08 AM, Mar 20 1996
         Start make

Wed 11:05:23 AM, Mar 20 1996
         End make

// 1st run to check correctness //
Wed 11:05:24 AM, Mar 20 1996
         Start run

Wed 11:05:54 AM, Mar 20 1996
         End run

// 2nd run to check correctness //
Wed 11:06:20 AM, Mar 20 1996
         Start run

Wed 11:06:57 AM, Mar 20 1996
         End run

// Total time: 620 Seconds
// Edit time:  526 Seconds
```

Figure 7.28: An annotated portion of an event log from Experiment 2. Comments are enclosed in `// //` brackets.

This allows counting the number of compilations as well as computing the time the participant spent editing the configuration. The start and end time for each execution of the configuration in the simulation system is also logged.

7.2.5 Overview of Experimental Results

Figure 7.29 shows a representative solution for a task in Experiment 2. Each participant constructed a `robot_command` function which called the library of behaviors and perceptual processes to complete the mission. This support library exactly matched those available in the graphical configuration editor.

```
Vector robot_command()
{
  static int status = 0;
  if(SigSense(SAFE))
  {
    switch (status)
    {
      case 0:   /*  At start  */
        status = 1;
        return MoveTo(flags);
      case 1:  /*  On way to flag  */
        if(Near(flags,0.1))
        {
          status = 2;
          return Stop();
        }
        return MoveTo(flags);
      case 2:  /*  At flag  */
        status = 3;
        return Stop();
      case 3:
        if (Near(home_base,0.1))
          return Stop();
        return MoveTo(home_base);
    }
  }
  else return Stop();
}
```

Figure 7.29: A representative task solution

The experiment was structured to allow a direct comparison between the graphical configuration editor and the `C` programming language. The results, summarized below, clearly demonstrate the advantages of using the graphical editor over handcrafting solutions in `C`. For the 4 people who completed both Experiment 1 and Experiment 2:

- In 12 instances participants completed a task faster using the *MissionLab* configuration editor than they completed the same task using `C`.
- In only one instance did a participant complete a task faster using `C` than using the configuration editor. Note: This occurred on Task 5 and the participant had previously completed the GUI portion.
- In 4 cases times were similar.
- In general, the times required to generate solutions using the configuration editor were more consistent.
- The average time required by the 4 participants for each task was 12.4 minutes using `C` and 5.9 minutes using the configuration editor.
- The average number of compilations was 4 using `C` and only 2 using the configuration editor.

7.2.6 Detailed Experimental Results

Number of Compilations

Number of Compilations using C					
Participant	*Task 1*	*Task 2*	*Task 3*	*Task 4*	*Task 5*
2	4	7	5	1	3
8	3	6	3	2	—
9	3	1	4	4	8
11	4	5	2	10	2

Figure 7.30: Number of compilations required to complete each task using the C language.

The number of compilations each participant made before completing the task is shown in Figure 7.30. Participants 2 and 11 completed the GUI portion first. A dash (—) indicates there wasn't time to work on that task. The average number of compilations required to construct a correct configuration is 4.0 computed over the 19 data points. The standard deviation in this data is 2.37. This matches the predicted 4 compilations necessary when using C.

Figure 7.31 compares the number of compiles required both when using C and the GUI. Recall that the average number of compiles when using the GUI was computed to be 2.02 with a standard deviation of 1.66. This shows that the GUI provides a better development environment by reducing the compile-edit-test cycles by half in this experiment.

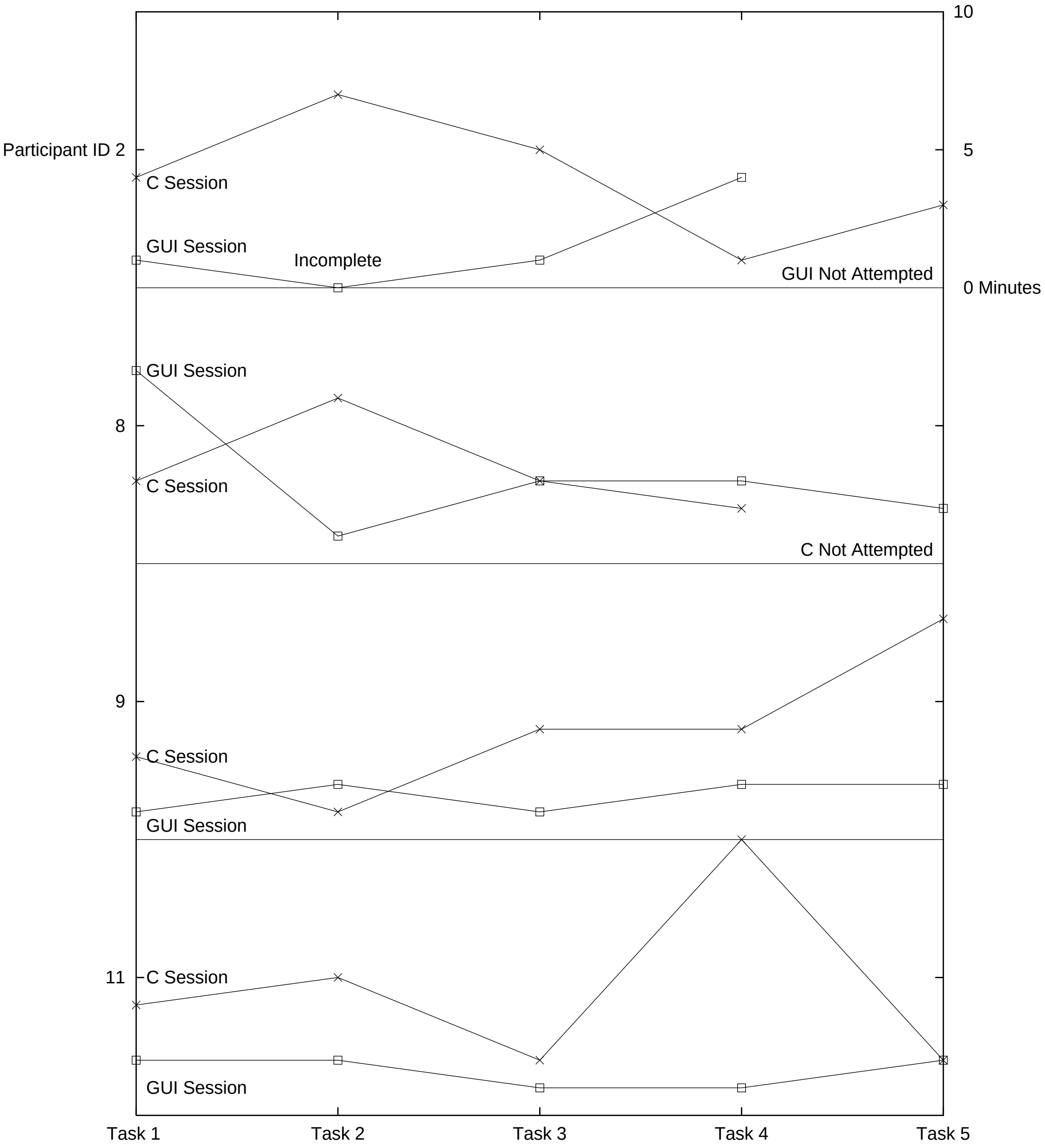


Figure 7.31: Graph of number of compiles required in GUI and C sessions. GUI sessions are marked with □ and C sessions with ×. The vertical axis represents the number of compiles. There are 11 instances where the compiles using C exceeds that for the GUI. This compares to 4 instances where C required fewer compilations and 2 where the numbers match. This shows the advantage in using the GUI over the C programming language.

Time to create a simple configuration

Figure 7.25 presented edit times for the participants using the GUI and Figure 7.33 presents edit times for the participants using the C programming language. Figure 7.32 graphs both sets of data for the four people who participated in both experiments to allow a closer comparison.

There are 12 instances where the time taken using the GUI is less that when using C. There is one clear case of the GUI taking longer and 4 examples where the times are quite similar.

Notice that only Subjects 2 and 11 were able to do better using C than with the GUI. This is interesting since Subjects 2 and 11 performed the GUI portion (Experiment 1) before the C session (Experiment 2). It appears that there is a speed-up from performing the same experiments again using the other modality. However, even these participants performed the tasks faster using the GUI in most instances.

7.3 Experiment 3: Configuration Synthesis

7.3.1 Objective

This experiment exercised the *MissionLab* toolset's multi-architecture support. Existing technologies do not support retargeting of configurations and no direct comparison is possible. Therefore, the experiment concentrated on demonstrating the *MissionLab* capabilities. Data gathered during the experiment will provide insights into the strengths and weaknesses of *MissionLab* for these tasks.

A small group of researchers experienced with creating robot configurations participated in this experiment. They first created a generic configuration (from scratch) which moved the robot through two waypoints to an area where the operator controlled it using teleoperation. When the operator exited teleoperation mode, the robot returned to the starting location.

The configuration was first bound to a Denning MRV-2 robot and evaluated using the *MissionLab* simulator. Once it worked as expected, the mission was repeated with the configuration deployed on a real MRV-2 robot. Finally, the configuration was unbound and subsequently rebound to a HUMMER robot, which uses the SAUSAGES run-time architecture. The SAUSAGES code generated by *MissionLab* was then evaluated using the CMU SAUSAGES simulation package to verify its correctness.

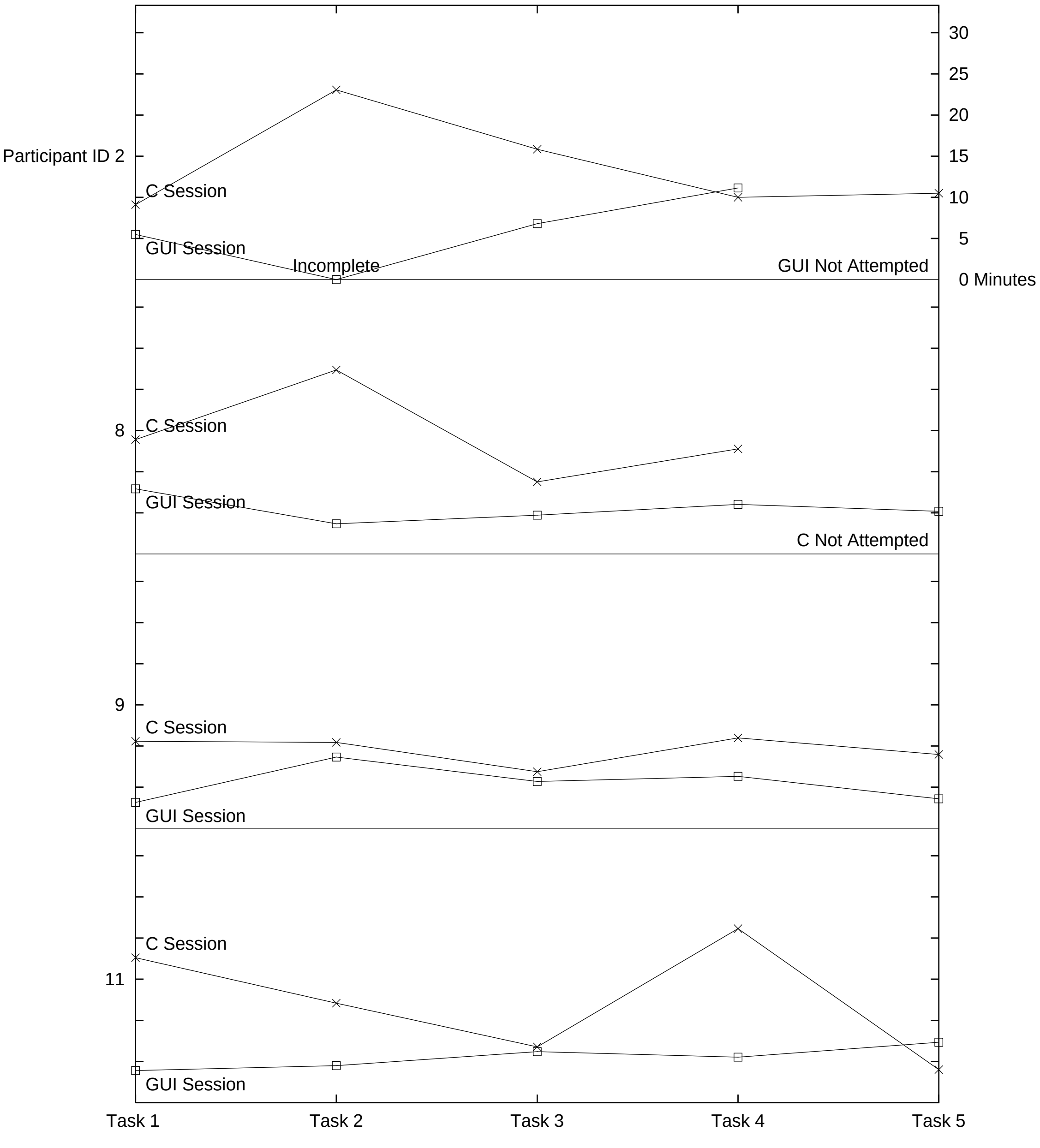


Figure 7.32: Graph of the time spent editing in both the GUI and C sessions. GUI sessions are marked with □ and C sessions with ×. The vertical axis represents time required to complete the tasks. There are 12 instances where the time taken using the GUI is less that when using C. There is one clear case of the GUI taking longer and 4 examples where the times are quite similar. These results show using the GUI speeds the development process.

Edit time using C					
Participant	*Task 1*	*Task 2*	*Task 3*	*Task 4*	*Task 5*
2[1]	547	1383	951	600	630
8[2]	834	1342	526	766	—
9[2]	635	626	413	659	538
11[1]	1057	725	407	1269	241

1. Performed the GUI tasks first.
2. Performed the C tasks first.

Figure 7.33: Total edit time (in seconds) when using the C programming language. A dash (—) indicates there wasn't time to work on that task.

7.3.2 Experimental Setup

Test Environment

The tests were conducted in the Georgia Tech Mobile Robot Laboratory. A SPARC 5 workstation was used for the experiments. An MRV-2 Mobile robot was connected to the SPARC via radio modems to allow using it from *MissionLab*. The sessions were videotaped using a fixed camera positioned behind the participant. The author supervised the experiment due to lack of other personnel. The participants were given rather explicit instructions on how to perform the experiment.

Test Participants

Three expert roboticists participated in this experiment. Two of the participants had previously taken part in Experiment 1 and one of these had also completed Experiment 2. The third had not used *MissionLab* previously. None of the participants had performed any binding operations using the graphical editor prior to this experiment. Codes assigned to participants are the letters `A`, `B`, and `C`.

Software

The *MissionLab* toolset version 1.0*b* was used in this experiment. The SAUSAGES simulation system provided by Jay Gowdy and Carnegie-Mellon University was used to evaluate the SAUSAGES script files generated during the experiment.

Programming Model

The programming model developed in Section 6.6.2 was used for this experiment.

7.3.3 Experimental Procedure

The procedures outlined in Section 6.6.3 were followed during this experiment. The script used is reproduced as Figure A.17 in Appendix A.

7.3.4 Raw Data Generated

Raw data was gathered using the *MissionLab* event logging facilities. The event log parsing tool was used to extract the information presented below from the logs. The format of the event logs was documented in Section 7.1.4.

7.3.5 Experimental Results

Time to complete Experiment 3			
Participant	*Part 1*	*Part 2*	*Part 3*
A	694	5	94
B	690	1	82.7
C	471	24	93.5

Figure 7.34: Time to complete Experiment 3. Part 1 is the number of seconds it took the participant to construct the initial configuration and deploy it on a simulated MRV-2 robot. Part 2 is the number of seconds to restart the configuration driving a real MRV-2 mobile robot. Part 3 is the number of seconds to unbind the configuration, rebind it to target a UGV robot and output the corresponding SAUSAGES code. Notice it only took about 1.5 minutes to retarget configurations. This is a huge savings over recoding.

Figure 7.34 lists the times in seconds that it took the three participants to complete the portions of the experiment. Part 1 required the participants to create a configuration which moved the robot through the specified mission. The time listed in Figure 7.34 includes the time required to compile the configuration.

The time after evaluating the configuration in simulation to restart the system to drive one of the real MRV-2 mobile robots is shown as Part 2 in the figure. This time is minimal except for the final one, where the robot was not prepared before the experiment began. None of the configurations required any changes after simulating them before they were ready to drive the mobile robot.

Part 3 times shown reflect the time required to retarget the configuration to a UGV robot. This requires unbinding the configuration from an MRV-2 robot, rebinding to a UGV robot, and recompiling to invoke the SAUSAGES code generator. The time required to save the file is also included since the procedures requested participants to save their configuration before recompiling. Notice this only took about 1.5 minutes to retarget a configuration to a different architecture and generate executables for a different robot. This is a great speedup compared to the 10 minutes required to construct the initial configurations. The retargeting support of *MissionLab* will greatly improve developers' ability to support heterogeneous robots.

None of the participants experienced any errors in compiling or binding their configurations. This experiment did not include enough subjects to perform any in-depth analysis of the logs or to produce statistically significant information.

A comparative analysis is not possible since there is no other system which supports retargeting robot configurations to other run-time architectures. However, the experiment demonstrated a factor of 5 speedup from recoding the configuration from scratch to retargeting the configuration using *MissionLab*. The participants were able to construct a single configuration and run it both on MRV-2 robots using the AuRA architecture and on a simulated HUMMER robot using the SAUSAGES run-time system. No modifications to the configuration were required, just rebinding to the correct architecture. This success is achieved by raising configuration specification above the constraints imposed by individual robot run-time architectures.

7.4 Summary

This chapter has documented the experimental evaluation of the *MissionLab* toolset. These studies focused on use of the graphical configuration editor for encoding and deploying missions on mobile robots. In Experiment 1, twelve participants used the graphical editor to construct five different configurations based on mission descriptions. Experiment 2 repeated the same tasks with four participants who coded solutions for the missions using the C programming language. Experiment 3 used three participants to demonstrate the ability to retarget configurations developed using the graphical editor.

Values for various usability criteria for the graphical editor were established using event logging data gathered in Experiment 1. Table 7.1 is a reproduction of Table 6.2 with the measured values column added. This presents the usability criteria in tabular form for ease of comparison. Notice the measured values are all far superior to the expected values. However, the large amount of variance in the time users spent picking new behaviors and new triggers points out some difficulty in that area. A popup window currently presents an alphabetical list of choices, each with a short description. More effort is needed in both naming and describing the behaviors and triggers to make their usage apparent.

Looking back at the analysis of Experiment 1 it is apparent that Subjects 7 and 10 were very successful with the GUI even though they were unable to program in C (*i.e.,* Figure 7.26). Neither had any significant programming experience. Subject 12 is a non-programmer ROTC student (undergraduate ME) who struggled a bit learning the system, but was very successful on Task 2 with only one hour of exposure to the system. Subjects 1 and 3 were administrative people who use computers as part of their job duties, but only use application software such as e-mail or word processors. Both were able to construct configurations where the robot partially completed the tasks. Examples of partial completion include sending the robot straight home instead of first moving to the flag (because of an incorrect selection of trigger to terminate the move to flag behavior), and transporting one object successfully but not knowing how to construct a loop to move all the objects (this subject began replicating the successful portion of the configuration and moved a total of three objects). The fact that there were two users doing very well and the remainder making significant progress with such minimal training supports the claim that this system empowers non-programmers.

Experiment 2 was conducted with only 4 participants. The group of expert programmers had varying levels of exposure to robotics. After a mere twenty minutes of training with the graphical editor the participants were able to achieve parity with their C programming times and many were faster using the graphical toolset. The reduction in the number of compilations necessary when using the GUI also makes it apparent that it is easier to create correct robot configurations using the graphical editor. This shows that even expert robot programmers can benefit by using the MissionLab toolset and the learning curve is sufficiently short to encourage switching.

Experiment 3 was used to demonstrate the ability to retarget configurations created using the graphical editor. The group of participants were able to easily create a generic configuration. They were then able to rapidly deploy their solution on a simulated Denning robot, a real robot, and on a simulated UGV robot using the CMU SAUSAGES simulator. The speedup gained from retargeting versus recoding demonstrates the success of the explicit binding process and multiple code generators used by *MissionLab.*

The experiments have established the power and utility of the *MissionLab* toolset. The graphical configuration editor is an especially powerful tool which allows non-programmers to construct, evaluate, and deploy robot missions. This success shows

Table 7.1: The *MissionLab* usability criteria with the experimentally measured values included.

***MissionLab* Usability Criteria**							
	Usability Attribute	Value to be Measured	Current Level	Worst Acceptable Level	Target Level	Best Possible Level	Measured Value
1.	Novice user performance	Time to add a mission step	1 Min	30 sec	10 sec	1 sec	2.2 sec
2.	Novice user performance	Time to specialize a step	2 min	1 min	30 sec	3 sec	6.2 sec
3.	Novice user performance	Time to parameterize a step	1 min	1 min	30 sec	2 sec	4.1 sec
4.	Novice user performance	Time to add a mission transition	1 min	30 sec	10 sec	2 sec	2.6 sec
5.	Novice user performance	Time to specialize a transition	2 min	1 min	30 sec	3 sec	4.9 sec
6.	Novice user performance	Time to parameterize a transition	1 min	1 min	30 sec	2 sec	4.0 sec
7.	Novice user performance	Number of compiles to create a configuration	4	5	2	1	2.0
8.	Novice user performance	Time to create a simple configuration	20 min	20 min	15 min	5 min	7.4 min
9.	Non-programmer performance	Ability to create configurations	No	Yes	Yes	Yes	Yes
10.	User acceptance	General feeling after use	N/A	medium	good	great	good

that the uniform representation provided by the Configuration Description Language is an asset when developing user programming tools.

Chapter 8

Summary and Contributions

8.1 Summary

Behavior-based robotic systems are becoming both more prevalent and more competent. However, operators lacking programming skills are still forced to use canned configurations hand-crafted by experienced roboticists written in traditional programming languages such as C and LISP. This inability of ordinary people to specify tasks for robots is inhibiting the spread of robots into everyday life. Even expert roboticists are limited by this problem. Since there is no commonality of configuration descriptions, researchers are unable to share solutions in executable forms. Further, since each model of robot typically has a disparate run-time architecture, a configuration commonly requires significant rework before it can be deployed on a different robot, even one with similar capabilities. This dissertation has attacked this problem from three fronts.

First, the foundational **Societal Agent** theory describes how agents form abstract structures at various levels in a recursive fashion (Chapter 3). It provides a uniform view of agents, no matter what their physical embodiment. Agents are treated consistently across the spectrum, from a primitive motor behavior to a configuration coordinating large groups of robots. The recursive nature of the agent construction facilitates information hiding and the creation of high-level primitives. The **Societal Agent** theory provides a framework for analyzing societies of agents transcending robotics.

Secondly, the *MissionLab* toolset was developed to measurably improve the process of specifying robot configurations (Chapter 5). *MissionLab* supports the graphical construction of architecture- and robot-independent configurations by using the Configuration Description Language (CDL) as the representation of the configuration (Chapter 4). This independence allows users to directly transfer designs to be bound to the specific robots at the recipient's site. Support for multiple code generators ensures that a wide variety of robots can be supported.

MissionLab utilizes the *assemblage* construction to support building new coherent behaviors from coordinated groups of other behaviors. The new behaviors can then be archived and reused in subsequent designs with parameters differing based on mission requirements. This recursive construction process allows users to build increasingly high-level primitives which are domain specific and directly tailored to the needs of the organization.

The methodology of *temporal sequencing* brings an object-oriented approach to robotics by partitioning a complex mission into discrete operating states with perceptual triggers causing state transitions. This allows the construction of several smaller configurations (assemblages) to implement each of the states. The *temporal sequencing* methodology is directly supported by *MissionLab*, which allows graphical construction of such state-transition diagrams.

Thirdly, specific usability criteria for toolsets such as *MissionLab* were established (Chapter 6). Three usability studies were defined to allow experimental establishment of values for these criteria. As part of this research, these three studies were carried out using the *MissionLab* toolset and confirmed its benefits over conventional techniques (Chapter 7). An impressive result was that users unable to program in conventional languages were still able to specify complex robot missions with only 30 minutes training using the toolset. This confirms that a major goal of this research, to empower non-programmers to create and evaluate configurations, has been met.

8.2 Specific Contributions

This research project has produced several specific contributions to the field of robotics:

The Societal Agent theory

The fundamental contribution of this dissertation is development of the **Societal Agent** theory. This theory provides a uniform representation of computational objects at all levels in hierarchical organizations. The power of this representation supports recursive composition of operators, information hiding, and code reuse. It also provides a framework for representing and studying complex societal structures such as ant colonies, animal herds, and military organizations. The utility of the **Societal Agent** theory has been borne out by its successful implementation in the *MissionLab* toolset.

Usability criteria and experiments

A contribution to the robotics domain is the creation of usability criteria for toolsets such as *MissionLab* and the design of experiments suitable for experimentally measuring values of the criteria. These experiments evaluate the performance of people completing certain benchmark tasks using the tools being tested. These experiments were used to evaluate the *MissionLab* toolset, both to measure the toolset's usability and to validate and benchmark the experiments themselves.

The Configuration Description Language

The Configuration Description Language (CDL) was developed to capture the recursive composition of configurations in an architecture- and robot-independent fashion. CDL provides a compact, exact description of individual robot configurations as well as the interactions of societies of cooperating mobile robots. The language is used to specify the *configuration* of behaviors and not the implementation of behaviors. This allows construction of generic configurations which can be bound to specific robots through an explicit binding step. No other language currently supports this explicit binding and, thus, the independence afforded by CDL.

A feature of behavior-based controllers is that behaviors themselves become almost trivial in complexity. The complexity of the system is transferred into the coordination of the multitude of simple behaviors active within the configuration. Therefore, an important facet of CDL is the explicit representation of coordination mechanisms. By forcing designers to partition coordination from behaviors, both are made less complex and easier to modify.

The *MissionLab* toolset

The *MissionLab* integrated development environment was created as part of this research to support the creation of behavior-based robot configurations. *MissionLab* includes an interactive designer to allow graphical specification and visualization of robot missions. *MissionLab* is based on CDL and supports the uniform representation of components inherent in that language. Support for explicit binding and multiple code generators allow *MissionLab* to support the many varieties of robots and robot run-time systems in common use. The usability experiments confirmed that *MissionLab* empowers non-programmers to specify missions for mobile robots.

MissionLab provides a least-commitment design architecture where users are free to build configurations unconstrained by hardware limitations and later map them onto vehicles providing the required functionality. The graphical editor facilitates configuration design and modification by non-programmers and speeds the process even for expert programmers. The graphical presentation improves configuration maintainability. Component reuse is encouraged by the recursive construction. As components are created they can be easily archived to libraries and reused as components in subsequent constructions. The result is to improve the design process both in speed and accuracy and to improve the quality of the designs themselves through the reuse of tested components.

8.3 Future Work

Research projects generally open more avenues for exploration than they close and this effort is no different. This section will attempt to describe those areas of future research which have occurred to the author over the course of this project.

The **Societal Agent** theory makes strong claims as to the utility and pervasiveness of representing coordinated collections of agents in a recursive fashion. Although used in this research within the robotics domain, the theory also provides a framework for representing and studying complex societal structures such as ant colonies, animal herds, and military organizations. It would be interesting to conduct psychological studies to attempt to verify that humans indeed follow these abstractions when interacting with such complex structures.

The *MissionLab* toolset is well structured for use in an educational setting. The support for visually constructing robot missions without the use of conventional programming languages allows students to concentrate on building solutions without getting bogged down in programming details. The integrated compilation and simulation subsystems allow quick and easy evaluation of solutions. The rapid retargeting of configurations allows these solutions to be easily deployed on actual robots for further testing. An interdisciplinary course in robotics would benefit greatly from use of *MissionLab*, since many of the students would likely not be proficient programmers.

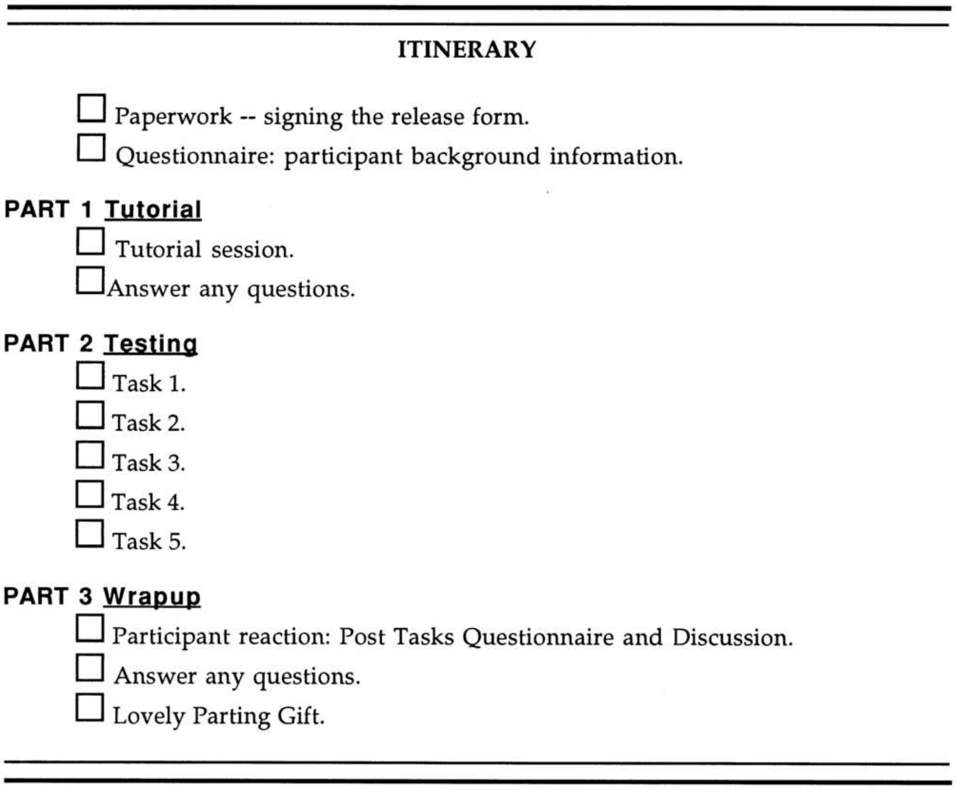
The *MissionLab* toolset will continue to evolve as new researchers tailor it to their needs. The recent *MissionLab* public release of version 1.0 is only a milestone in its continuing evolution. There are several features that could be added to *MissionLab* to improve its usability. Adding the capability to place arbitrary comments within the configuration, support run-time monitoring from within the editor (*i.e.*, source level debugging), and better run-time support for interacting and monitoring large numbers of vehicles are just a few of the possibilities. The usability studies pointed out a likely place for improvement is the behavior and trigger selection process.

8.4 Conclusion

The title of this dissertation is *A Design Methodology for the Configuration of Behavior-Based Mobile Robots*. The focus on improving the design process grew out of a desire to make robots commonly accessible. Before robots can move beyond the control of their designers and into the hands of **users** the *process* of specifying missions must be made accessible to ordinary people. The research documented in this dissertation has moved the state of the art in robot programming a substantial distance in that direction, and thereby achieved its purpose.

Appendix A

Documents from the Usability Experiments



ITINERARY

☐ Paperwork -- signing the release form.
☐ Questionnaire: participant background information.

PART 1 Tutorial

☐ Tutorial session.
☐ Answer any questions.

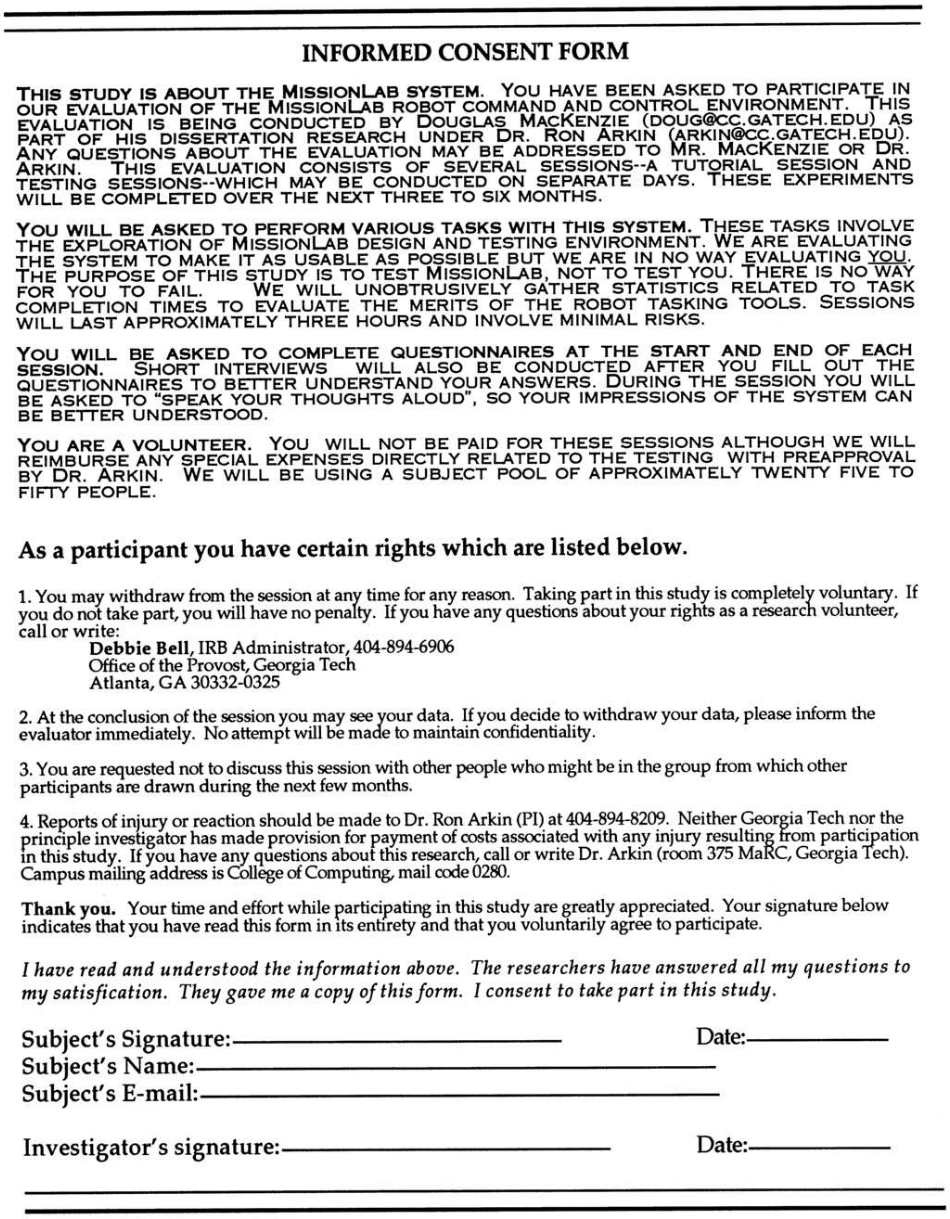
PART 2 Testing

☐ Task 1.
☐ Task 2.
☐ Task 3.
☐ Task 4.
☐ Task 5.

PART 3 Wrapup

☐ Participant reaction: Post Tasks Questionnaire and Discussion.
☐ Answer any questions.
☐ Lovely Parting Gift.

Figure A.1: This checklist was used for Experiments 1 and 2 by the monitor to maintain consistency between sessions.



INFORMED CONSENT FORM

THIS STUDY IS ABOUT THE MISSIONLAB SYSTEM. YOU HAVE BEEN ASKED TO PARTICIPATE IN OUR EVALUATION OF THE MISSIONLAB ROBOT COMMAND AND CONTROL ENVIRONMENT. THIS EVALUATION IS BEING CONDUCTED BY DOUGLAS MACKENZIE (DOUG@CC.GATECH.EDU) AS PART OF HIS DISSERTATION RESEARCH UNDER DR. RON ARKIN (ARKIN@CC.GATECH.EDU). ANY QUESTIONS ABOUT THE EVALUATION MAY BE ADDRESSED TO MR. MACKENZIE OR DR. ARKIN. THIS EVALUATION CONSISTS OF SEVERAL SESSIONS--A TUTORIAL SESSION AND TESTING SESSIONS--WHICH MAY BE CONDUCTED ON SEPARATE DAYS. THESE EXPERIMENTS WILL BE COMPLETED OVER THE NEXT THREE TO SIX MONTHS.

YOU WILL BE ASKED TO PERFORM VARIOUS TASKS WITH THIS SYSTEM. THESE TASKS INVOLVE THE EXPLORATION OF MISSIONLAB DESIGN AND TESTING ENVIRONMENT. WE ARE EVALUATING THE SYSTEM TO MAKE IT AS USABLE AS POSSIBLE BUT WE ARE IN NO WAY EVALUATING YOU. THE PURPOSE OF THIS STUDY IS TO TEST MISSIONLAB, NOT TO TEST YOU. THERE IS NO WAY FOR YOU TO FAIL. WE WILL UNOBTRUSIVELY GATHER STATISTICS RELATED TO TASK COMPLETION TIMES TO EVALUATE THE MERITS OF THE ROBOT TASKING TOOLS. SESSIONS WILL LAST APPROXIMATELY THREE HOURS AND INVOLVE MINIMAL RISKS.

YOU WILL BE ASKED TO COMPLETE QUESTIONNAIRES AT THE START AND END OF EACH SESSION. SHORT INTERVIEWS WILL ALSO BE CONDUCTED AFTER YOU FILL OUT THE QUESTIONNAIRES TO BETTER UNDERSTAND YOUR ANSWERS. DURING THE SESSION YOU WILL BE ASKED TO "SPEAK YOUR THOUGHTS ALOUD", SO YOUR IMPRESSIONS OF THE SYSTEM CAN BE BETTER UNDERSTOOD.

YOU ARE A VOLUNTEER. YOU WILL NOT BE PAID FOR THESE SESSIONS ALTHOUGH WE WILL REIMBURSE ANY SPECIAL EXPENSES DIRECTLY RELATED TO THE TESTING WITH PREAPPROVAL BY DR. ARKIN. WE WILL BE USING A SUBJECT POOL OF APPROXIMATELY TWENTY FIVE TO FIFTY PEOPLE.

As a participant you have certain rights which are listed below.

1. You may withdraw from the session at any time for any reason. Taking part in this study is completely voluntary. If you do not take part, you will have no penalty. If you have any questions about your rights as a research volunteer, call or write:

Debbie Bell, IRB Administrator, 404-894-6906
Office of the Provost, Georgia Tech
Atlanta, GA 30332-0325

2. At the conclusion of the session you may see your data. If you decide to withdraw your data, please inform the evaluator immediately. No attempt will be made to maintain confidentiality.

3. You are requested not to discuss this session with other people who might be in the group from which other participants are drawn during the next few months.

4. Reports of injury or reaction should be made to Dr. Ron Arkin (PI) at 404-894-8209. Neither Georgia Tech nor the principle investigator has made provision for payment of costs associated with any injury resulting from participation in this study. If you have any questions about this research, call or write Dr. Arkin (room 375 MaRC, Georgia Tech). Campus mailing address is College of Computing, mail code 0280.

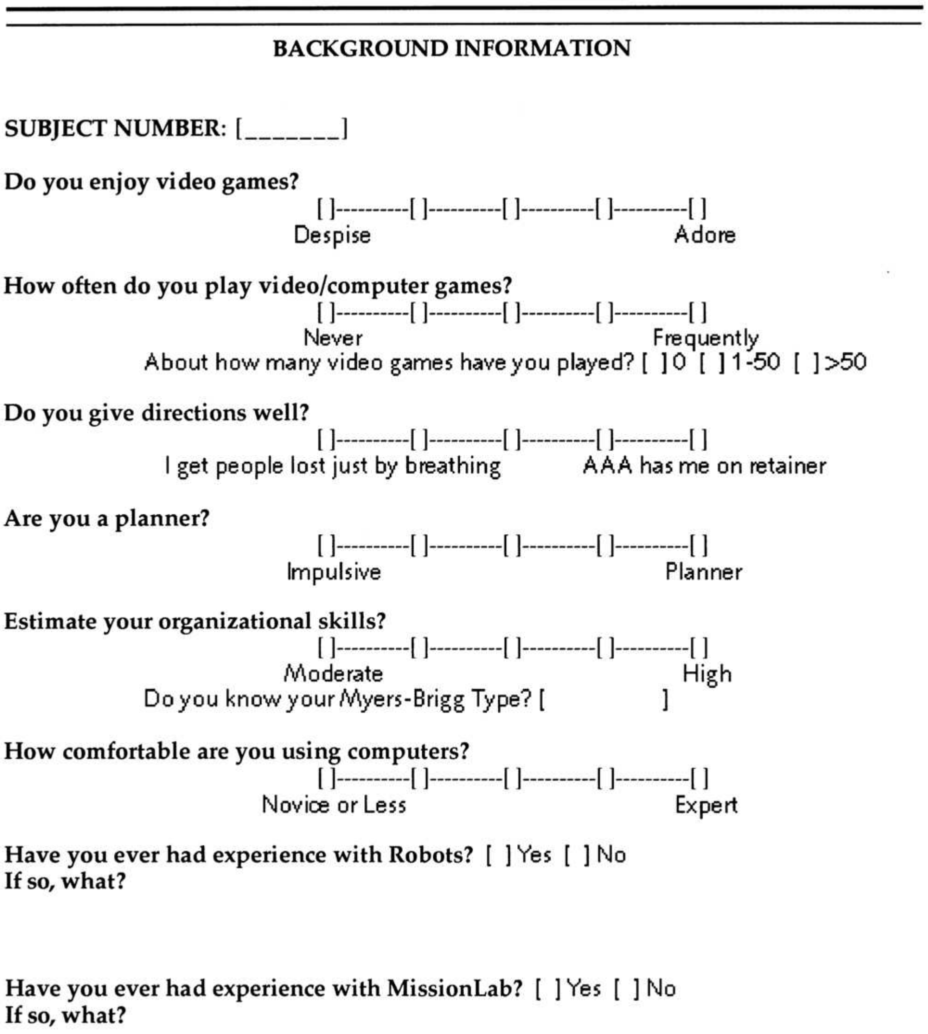
Thank you. Your time and effort while participating in this study are greatly appreciated. Your signature below indicates that you have read this form in its entirety and that you voluntarily agree to participate.

I have read and understood the information above. The researchers have answered all my questions to my satisfication. They gave me a copy of this form. I consent to take part in this study.

Subject's Signature:____________________ **Date:**__________
Subject's Name:____________________________
Subject's E-mail:____________________________

Investigator's signature:____________________ **Date:**__________

Figure A.2: This consent form was approved by the Georgia Tech oversight board for use in the usability experiments. All test subjects signed a copy of this form before taking part in any experiments.



BACKGROUND INFORMATION

SUBJECT NUMBER: [_______]

Do you enjoy video games?

[ ]----------[ ]----------[ ]----------[ ]----------[ ]
Despise Adore

How often do you play video/computer games?

[ ]----------[ ]----------[ ]----------[ ]----------[ ]
Never Frequently
About how many video games have you played? [ ] 0 [ ] 1-50 [ ] >50

Do you give directions well?

[ ]----------[ ]----------[ ]----------[ ]----------[ ]
I get people lost just by breathing AAA has me on retainer

Are you a planner?

[ ]----------[ ]----------[ ]----------[ ]----------[ ]
Impulsive Planner

Estimate your organizational skills?

[ ]----------[ ]----------[ ]----------[ ]----------[ ]
Moderate High
Do you know your Myers-Brigg Type? [ ]

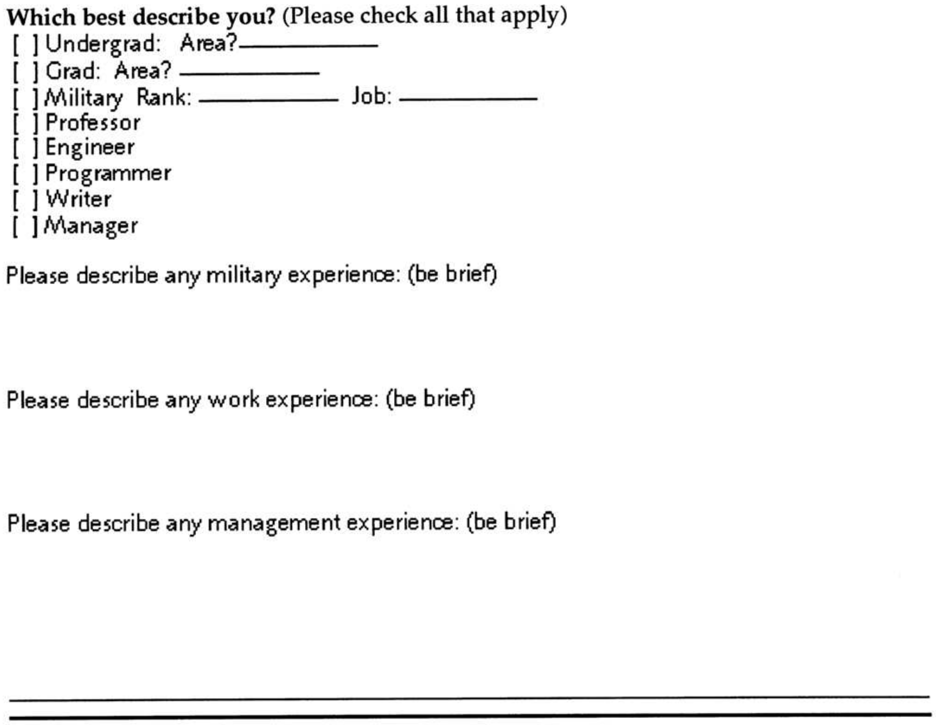
How comfortable are you using computers?

[ ]----------[ ]----------[ ]----------[ ]----------[ ]
Novice or Less Expert

Have you ever had experience with Robots? [ ] Yes [ ] No
If so, what?

Have you ever had experience with MissionLab? [ ] Yes [ ] No
If so, what?

Figure A.3: This is part 1 of the background questionnaire used to gather information about participants in the experiments (part 2 is Figure A.4). All test subjects filled out a copy of this form at the start of the experiments.



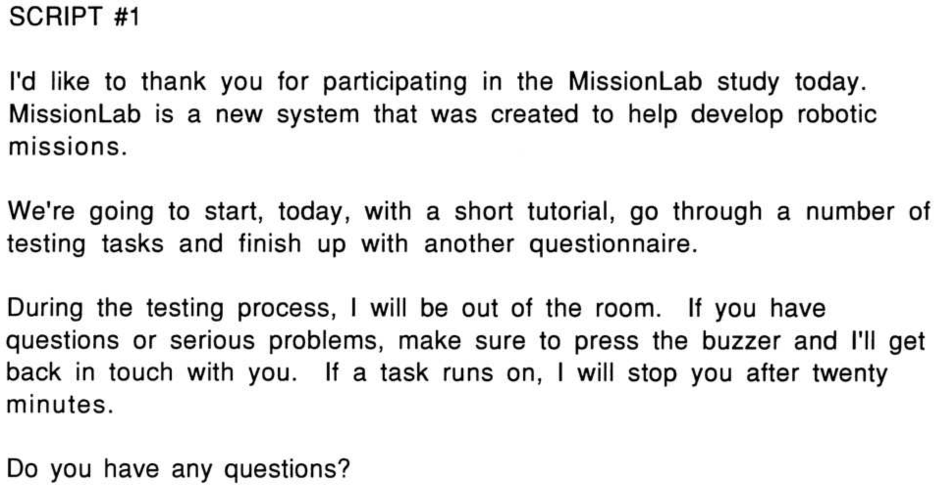
Which best describe you? (Please check all that apply)
[ ] Undergrad: Area? ____________
[ ] Grad: Area? ____________
[ ] Military Rank: ____________ Job: ____________
[ ] Professor
[ ] Engineer
[ ] Programmer
[ ] Writer
[ ] Manager

Please describe any military experience: (be brief)

Please describe any work experience: (be brief)

Please describe any management experience: (be brief)

Figure A.4: This is part 2 of the background questionnaire (part 1 is Figure A.3).



SCRIPT #1

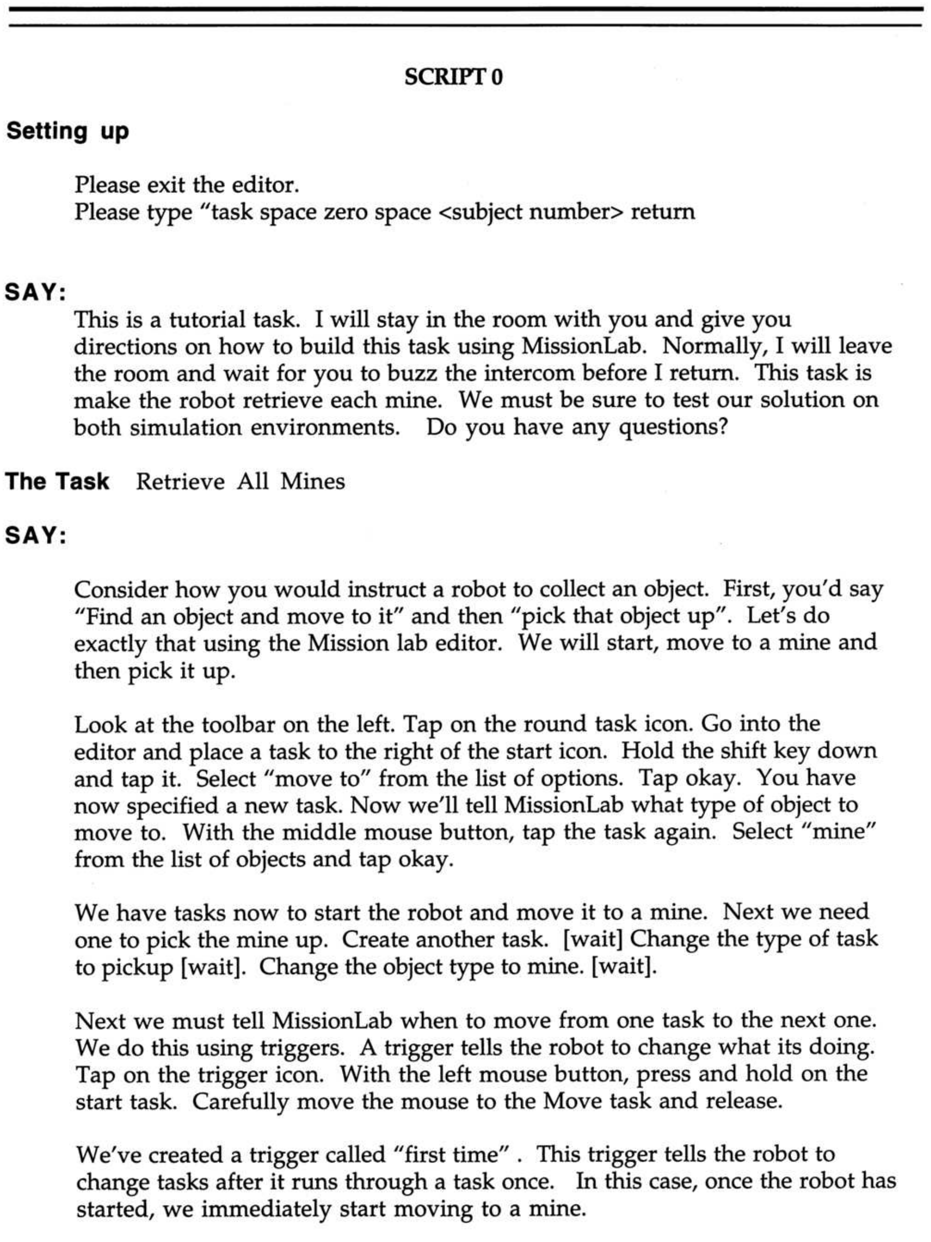
I'd like to thank you for participating in the MissionLab study today. MissionLab is a new system that was created to help develop robotic missions.

We're going to start, today, with a short tutorial, go through a number of testing tasks and finish up with another questionnaire.

During the testing process, I will be out of the room. If you have questions or serious problems, make sure to press the buzzer and I'll get back in touch with you. If a task runs on, I will stop you after twenty minutes.

Do you have any questions?

Figure A.5: The interaction with the experiment monitor was scripted to ensure consistency between sessions. This is the "hello" script used to start the session.



SCRIPT 0

Setting up

Please exit the editor.
Please type "task space zero space <subject number> return

SAY:

This is a tutorial task. I will stay in the room with you and give you directions on how to build this task using MissionLab. Normally, I will leave the room and wait for you to buzz the intercom before I return. This task is make the robot retrieve each mine. We must be sure to test our solution on both simulation environments. Do you have any questions?

The Task Retrieve All Mines

SAY:

Consider how you would instruct a robot to collect an object. First, you'd say "Find an object and move to it" and then "pick that object up". Let's do exactly that using the Mission lab editor. We will start, move to a mine and then pick it up.

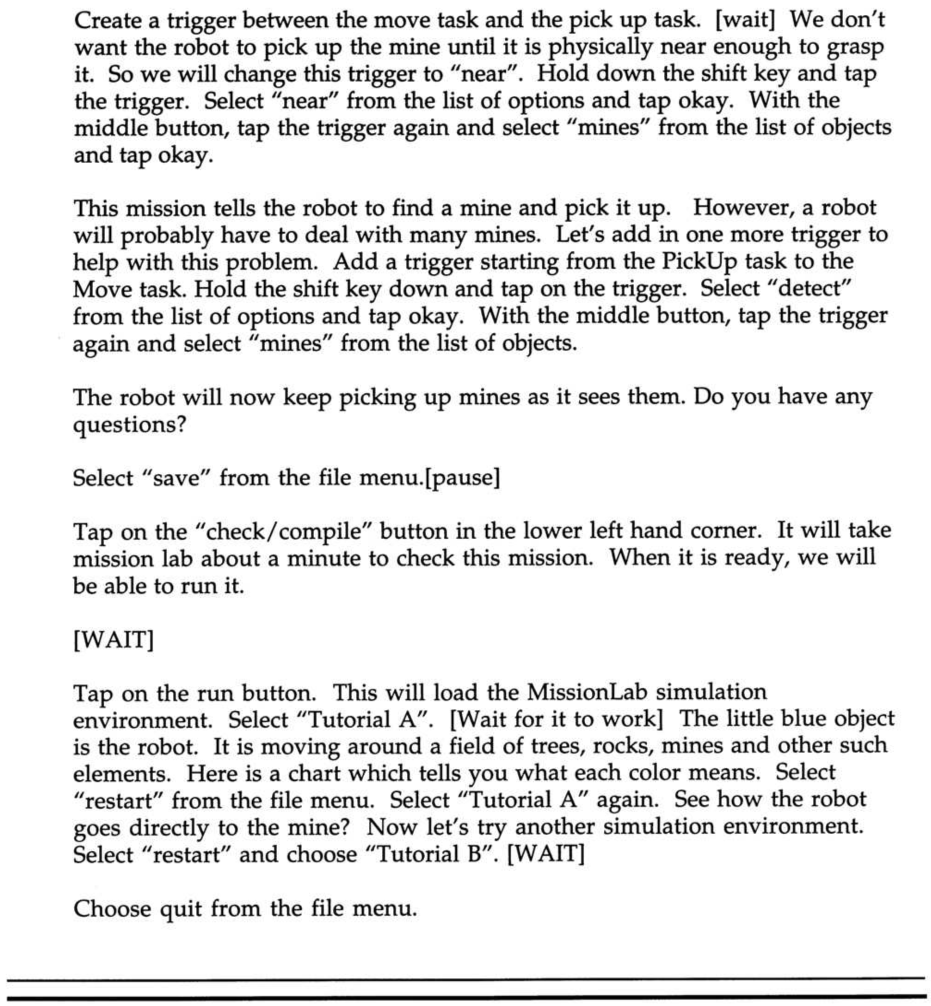
Look at the toolbar on the left. Tap on the round task icon. Go into the editor and place a task to the right of the start icon. Hold the shift key down and tap it. Select "move to" from the list of options. Tap okay. You have now specified a new task. Now we'll tell MissionLab what type of object to move to. With the middle mouse button, tap the task again. Select "mine" from the list of objects and tap okay.

We have tasks now to start the robot and move it to a mine. Next we need one to pick the mine up. Create another task. [wait] Change the type of task to pickup [wait]. Change the object type to mine. [wait].

Next we must tell MissionLab when to move from one task to the next one. We do this using triggers. A trigger tells the robot to change what its doing. Tap on the trigger icon. With the left mouse button, press and hold on the start task. Carefully move the mouse to the Move task and release.

We've created a trigger called "first time" . This trigger tells the robot to change tasks after it runs through a task once. In this case, once the robot has started, we immediately start moving to a mine.

Figure A.6: The participants first completed a warmup exercise while the monitor remained in the room and assisted them. This is part 1 of the script used during this warmup task (part 2 is Figure A.7).



Create a trigger between the move task and the pick up task. [wait] We don't want the robot to pick up the mine until it is physically near enough to grasp it. So we will change this trigger to "near". Hold down the shift key and tap the trigger. Select "near" from the list of options and tap okay. With the middle button, tap the trigger again and select "mines" from the list of objects and tap okay.

This mission tells the robot to find a mine and pick it up. However, a robot will probably have to deal with many mines. Let's add in one more trigger to help with this problem. Add a trigger starting from the PickUp task to the Move task. Hold the shift key down and tap on the trigger. Select "detect" from the list of options and tap okay. With the middle button, tap the trigger again and select "mines" from the list of objects.

The robot will now keep picking up mines as it sees them. Do you have any questions?

Select "save" from the file menu.[pause]

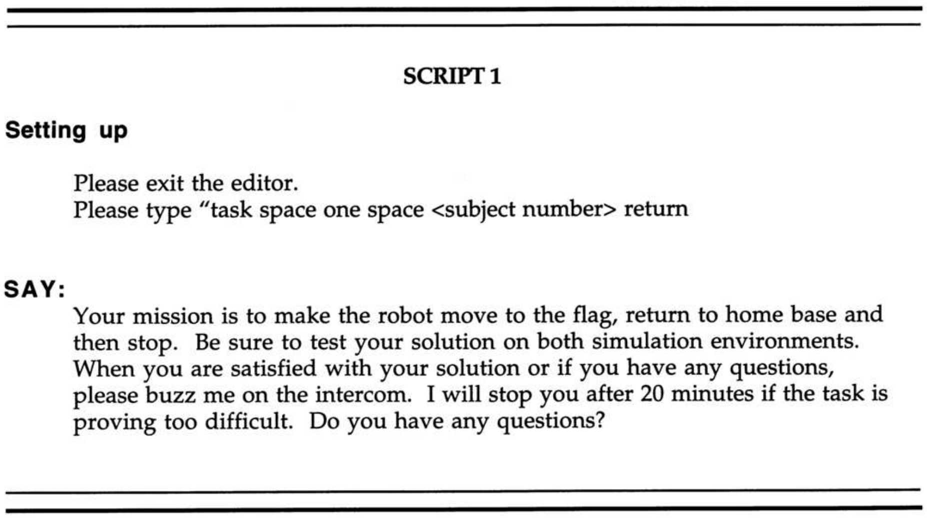
Tap on the "check/compile" button in the lower left hand corner. It will take mission lab about a minute to check this mission. When it is ready, we will be able to run it.

[WAIT]

Tap on the run button. This will load the MissionLab simulation environment. Select "Tutorial A". [Wait for it to work] The little blue object is the robot. It is moving around a field of trees, rocks, mines and other such elements. Here is a chart which tells you what each color means. Select "restart" from the file menu. Select "Tutorial A" again. See how the robot goes directly to the mine? Now let's try another simulation environment. Select "restart" and choose "Tutorial B". [WAIT]

Choose quit from the file menu.

Figure A.7: This is part 2 of the script for the warmup task (part 1 is Figure A.6).



SCRIPT 1

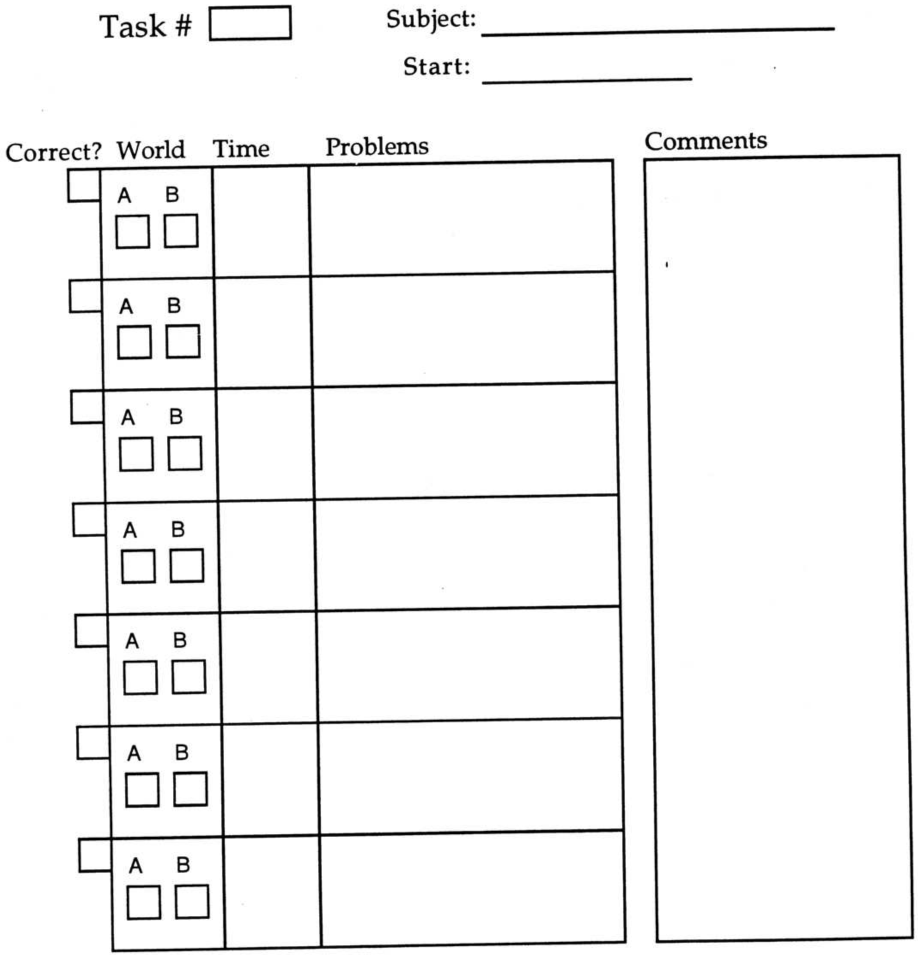
Setting up

Please exit the editor.
Please type "task space one space <subject number> return

SAY:

Your mission is to make the robot move to the flag, return to home base and then stop. Be sure to test your solution on both simulation environments. When you are satisfied with your solution or if you have any questions, please buzz me on the intercom. I will stop you after 20 minutes if the task is proving too difficult. Do you have any questions?

Figure A.8: This is the script used for task 1. The experiment monitor read this script and then left the room while the participants completed the task.



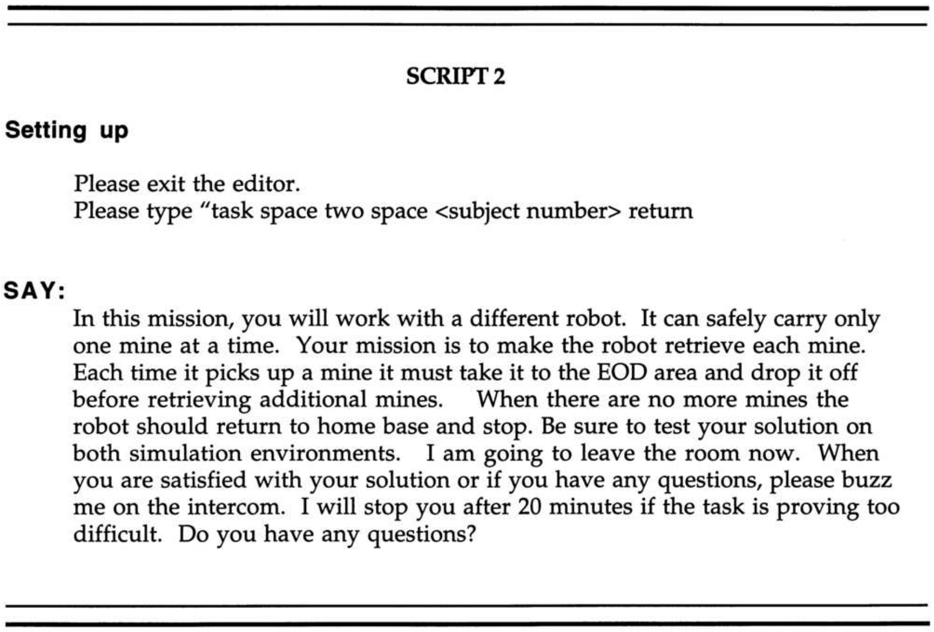
Task # ☐

Subject: ______________________

Start: ______________

Correct?	World	Time	Problems	Comments
☐	A B ☐ ☐			
☐	A B ☐ ☐			
☐	A B ☐ ☐			
☐	A B ☐ ☐			
☐	A B ☐ ☐			
☐	A B ☐ ☐			
☐	A B ☐ ☐			

Figure A.9: This form was used by the experiment monitor to record progress of the participants as they completed the tasks. Any unusual occurrences were also recorded for later examination. The participants were instructed to test their solutions in two different simulated worlds. The `World` column was used to record the results of their testing in each of the two environments.



SCRIPT 2

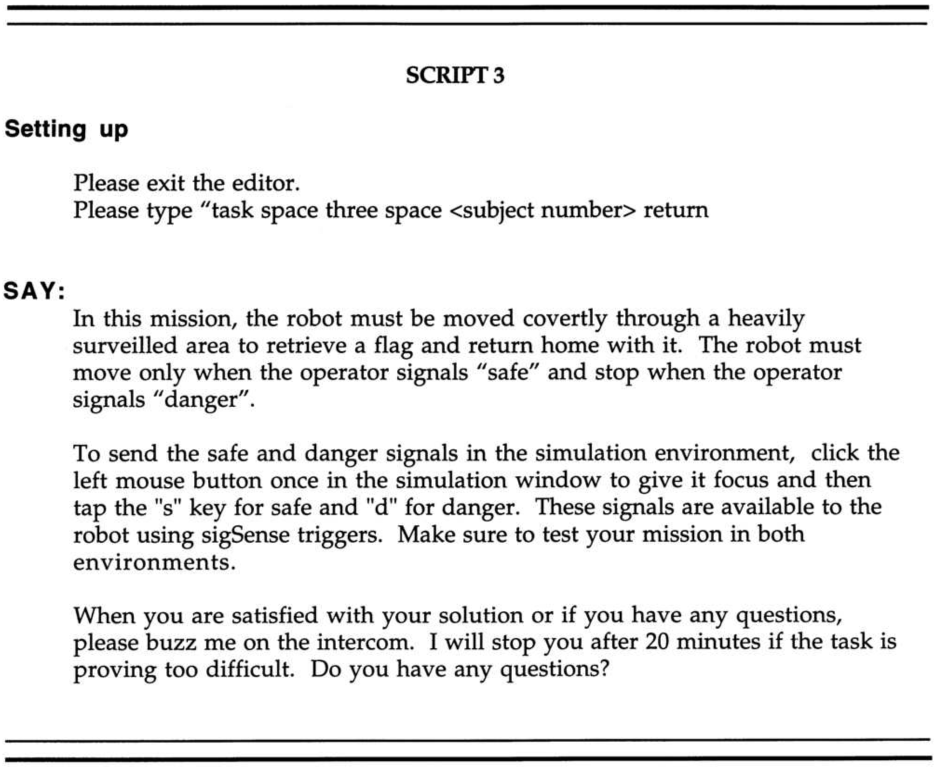
Setting up

Please exit the editor.
Please type “task space two space <subject number> return

SAY:

In this mission, you will work with a different robot. It can safely carry only one mine at a time. Your mission is to make the robot retrieve each mine. Each time it picks up a mine it must take it to the EOD area and drop it off before retrieving additional mines. When there are no more mines the robot should return to home base and stop. Be sure to test your solution on both simulation environments. I am going to leave the room now. When you are satisfied with your solution or if you have any questions, please buzz me on the intercom. I will stop you after 20 minutes if the task is proving too difficult. Do you have any questions?

Figure A.10: This is the script for Task 2.



SCRIPT 3

Setting up

Please exit the editor.
Please type “task space three space <subject number> return

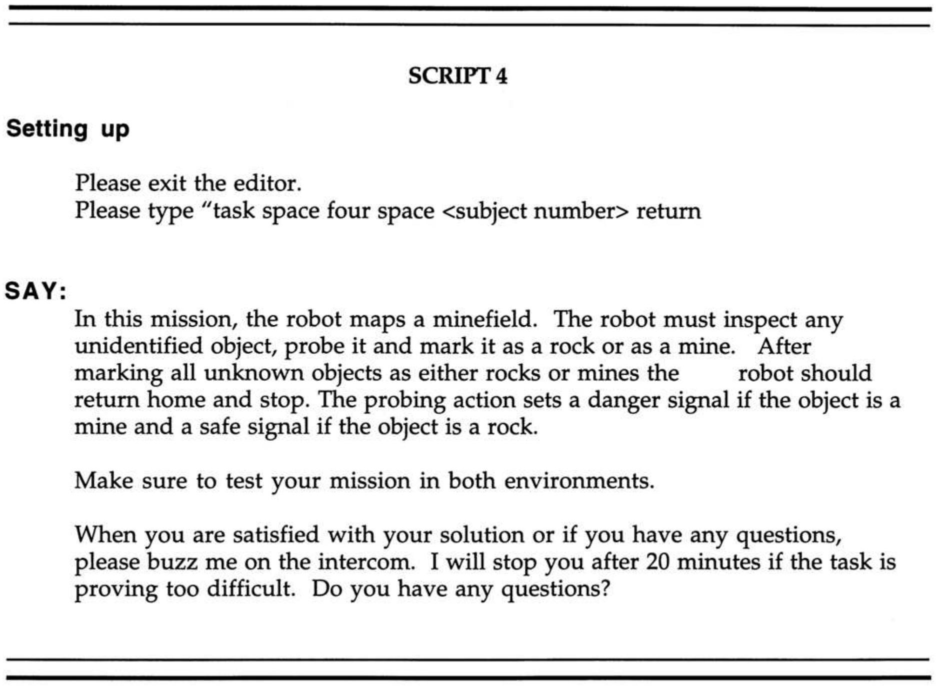
SAY:

In this mission, the robot must be moved covertly through a heavily surveilled area to retrieve a flag and return home with it. The robot must move only when the operator signals “safe” and stop when the operator signals “danger”.

To send the safe and danger signals in the simulation environment, click the left mouse button once in the simulation window to give it focus and then tap the "s" key for safe and "d" for danger. These signals are available to the robot using sigSense triggers. Make sure to test your mission in both environments.

When you are satisfied with your solution or if you have any questions, please buzz me on the intercom. I will stop you after 20 minutes if the task is proving too difficult. Do you have any questions?

Figure A.11: This is the script for Task 3.



SCRIPT 4

Setting up

Please exit the editor.
Please type “task space four space <subject number> return

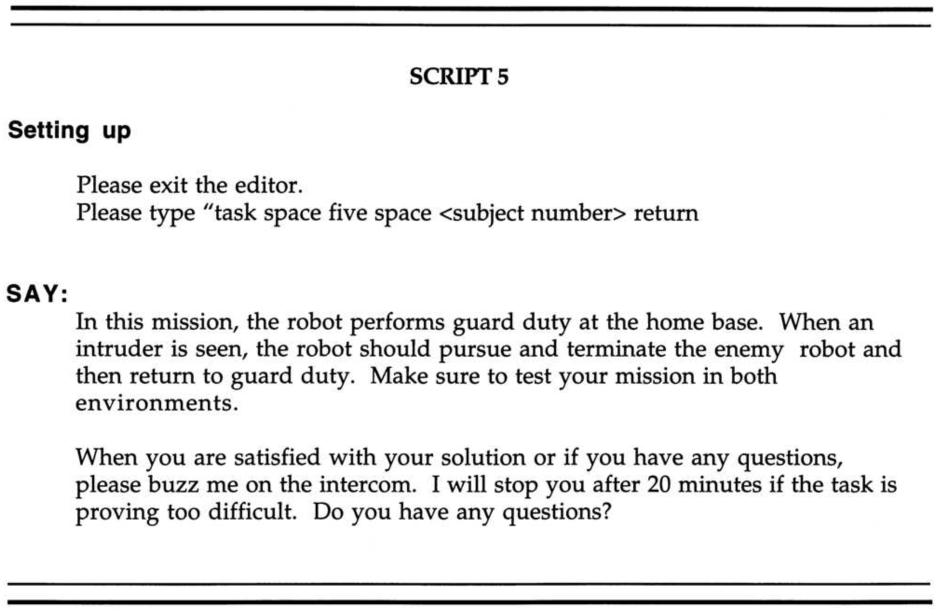
SAY:

In this mission, the robot maps a minefield. The robot must inspect any unidentified object, probe it and mark it as a rock or as a mine. After marking all unknown objects as either rocks or mines the robot should return home and stop. The probing action sets a danger signal if the object is a mine and a safe signal if the object is a rock.

Make sure to test your mission in both environments.

When you are satisfied with your solution or if you have any questions, please buzz me on the intercom. I will stop you after 20 minutes if the task is proving too difficult. Do you have any questions?

Figure A.12: This is the script for Task 4.



SCRIPT 5

Setting up

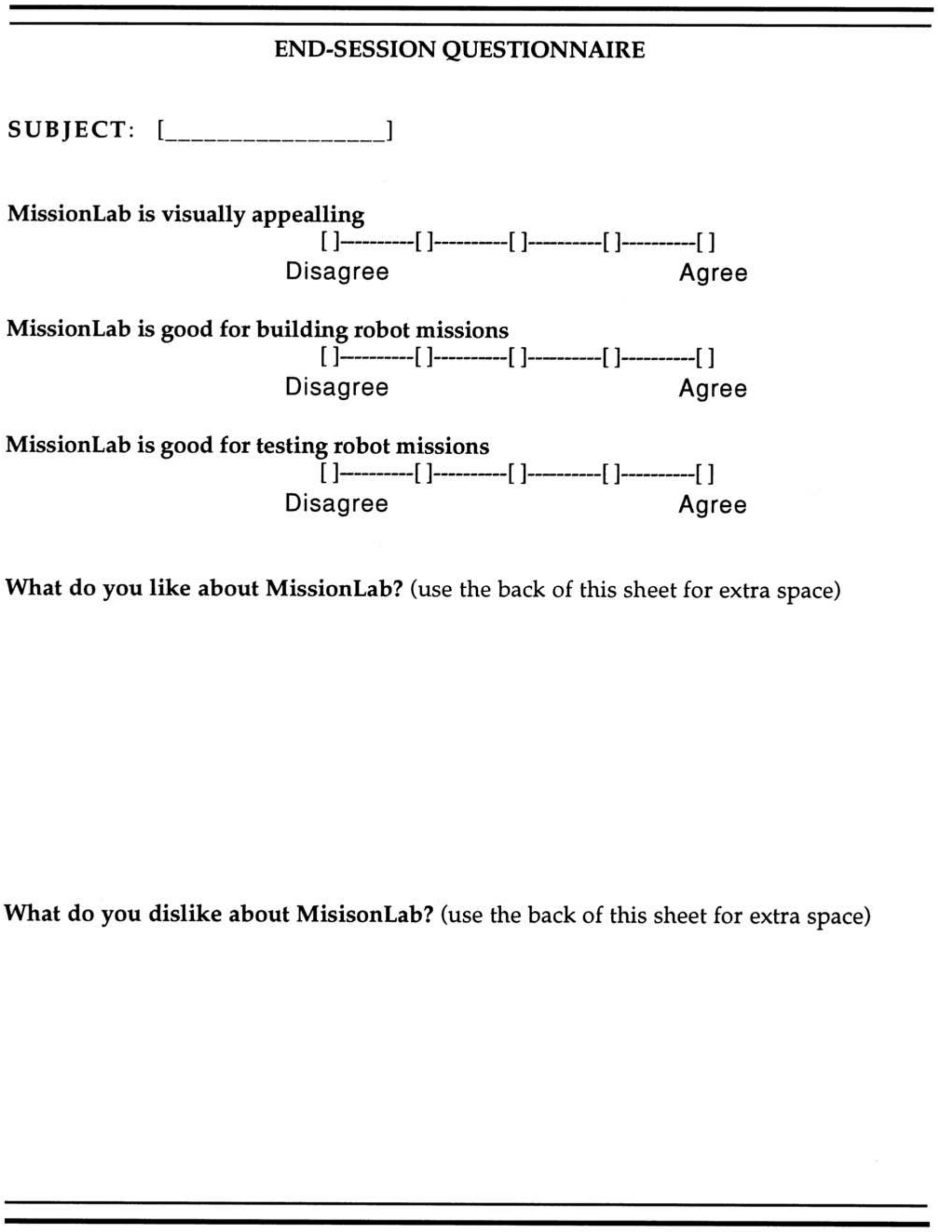
Please exit the editor.
Please type “task space five space <subject number> return

SAY:

In this mission, the robot performs guard duty at the home base. When an intruder is seen, the robot should pursue and terminate the enemy robot and then return to guard duty. Make sure to test your mission in both environments.

When you are satisfied with your solution or if you have any questions, please buzz me on the intercom. I will stop you after 20 minutes if the task is proving too difficult. Do you have any questions?

Figure A.13: This is the script for Task 5.



END-SESSION QUESTIONNAIRE

SUBJECT: [________________]

MissionLab is visually appealling

[ ]---------[ ]---------[ ]---------[ ]---------[ ]
Disagree Agree

MissionLab is good for building robot missions

[ ]---------[ ]---------[ ]---------[ ]---------[ ]
Disagree Agree

MissionLab is good for testing robot missions

[ ]---------[ ]---------[ ]---------[ ]---------[ ]
Disagree Agree

What do you like about MissionLab? (use the back of this sheet for extra space)

What do you dislike about MisisonLab? (use the back of this sheet for extra space)

Figure A.14: After the participants had completed as many of the tasks as possible in the alloted time they were asked to fill out this exit survey.

1. **Constants**

```
// The ObjectClasses parameter is a bit mask with the following fields:
#define mines                (1 << 0)
#define enemy_robots         (1 << 1)
#define flags                (1 << 2)
#define EOD_areas            (1 << 3)
#define rocks                (1 << 4)
#define trees                (1 << 5)
#define home_base            (1 << 6)
#define unknown_objects      (1 << 7)

// Signals
#define DANGER               'd'
#define SAFE                 's'
```

2. **Triggers**

```
// Causes an immediate transition
bool FirstTime();
// Never take this transition
bool Never();
// start a wait cycle
void StartWait();
// Causes a transition after the delay expires
bool CheckWait(int delay);
// Are we near one of the marked objects?
bool Near(ObjectClasses classes, double distance);
// Are we away from all objects of these types?
bool AwayFrom(ObjectClasses classes, double distance);
// Is there one of the objects we are looking for?
bool Detect(ObjectClasses classes);
// Causes a transition when an object is not detected
bool UnDetect(ObjectClasses classes);
// Check the last signal sent (remains true until a new one arrives)
bool SigSense(int signal);
```

Figure A.15: Reproduction of the handout describing the usage of the behavior library for Experiment 2, page 1 of 2.

3. Actions

```
// Kills the robot
void Terminate();

// The robot doesn't move
Vector Stop();

// Check the type of the object
Vector ProbeObject();

// move the robot to the closest object of a certain type
Vector MoveTo(ObjectClasses classes);

// move the robot away from selected types objects
Vector MoveFrom(ObjectClasses classes);

// the robot wanders about the environment
Vector Wander(double curious, double cautious);

// Pickup the closest object of the desired type
Vector PickUp(ObjectClasses classes);

// put the object into the EOD container
Vector PutInEOD();

// Change the object to the new type
Vector MarkObjectAs(ObjectClasses new_class);

// Causes the robot to drop a flag at the current location
Vector DropFlag();

// Terminate the nearest enemy robot
Vector TerminateEnemy();
```

Figure A.16: Reproduction of the handout describing the usage of the behavior library for Experiment 2, page 2 of 2.

Start cfgedit by typing: “task 1 id” and select a ”new robot” workspace in the editor. Start the process by clicking on the “OBP” button to bring up the list of output binding points. In this case, there is only one choice which is common to both the UGV and AuRA architectures. Click on “movement” and then “OK” and then place the output binding point icon in the workspace.

In this exercise, the movements of the robot will be controlled by a state-based coordination operator. Click on the “Operator” button to get the list of coordination operators. Again, there is only one that is common to both architectures, so click on “FSA” and then “OK”. Now place the FSA icon in the workspace and then connect its output to the input of the binding point by clicking on the arrows.

Move down to define the FSA by clicking the middle mouse button on the FSA icon. Now add a state which causes the robot to move to location ¡7.0, 3.5¿, then ¡4.0, 3.5¿. At this point the robot should go into Teleop mode, allowing the commander to guide it, and then finally it should return to location ¡9.0 4.0¿ and stop.

After you have constructed the state transition diagram, save your file as ”f_gen.cdl”. Then, click on the “Bind” button and select the AuRA architecture. We will be targeting the MRV2 robots for this mission, so select that class of devices when prompted. Now click on the “Output” button to generate an AuRA robot executable.

When the build finishes, click on “Run” and select a simulated robot to evaluate your work. When you are satisfied that it performs the correct tasks listed above, rerun it and select “Ren” as the robot to deploy the configuration on. (Make sure the radios and Ren are turned on). Ren should be positioned near the door to the lab, facing out the door. Run the mission on Ren to further evaluate your work.

Save this file as ”f_aura.cdl”. Click on the “Unbind” button to revert back to a generic configuration and then click on the “Bind” button and this time select the “UGV” architecture as the target. Generate the necessary LISP code by pressing the “Output” button and then invoke the SAUSAGES simulation system by pressing the ”Run” button.

Once the SAUSAGES system starts up (it takes a while) you will see a graphic display of the room. Click the left mouse button on the “Mission” pulldown menu and select “Load Mission”. Then repeat the process to select “Run Mission”. When you select “Run Mission” the robot should appear as a blue dot on the right end of the red line. Now pull down the “Flow” menu and select “continue” to start the robot moving. The active segment turns green and then blue when it has been completed. When the robot reaches the Teleop state, the “Modes” indicator on the upper left toolbar will switch to “Pick Point”. Now you can direct the robot by merely clicking the left mouse button in the workspace. Note: The robot can not cross the solid black room boundaries. When you have finished the Teleop leg, pull down the “Flow” menu and select “continue” to terminate this state.

When the mission is complete, pull down the “File” menu and select “Quit”. Save this file as “f_ugv” and you are finished.

Figure A.17: Reproduction of the script used for Experiment 3.

Bibliography

[1] M.A. Arbib, A.J. Kfoury, and R.N. Moll. *A Basis for Theoretical Computer Science*. Springer-Verlag, NY, 1981.

[2] R.C. Arkin. *Towards Cosmopolitan Robots: Intelligent Navigation of a Mobile Robot in Extended Man-made Environments*. Ph.D. dissertation, University of Massachusetts, Department of Computer and Information Science, 1987. COINS TR 87-80.

[3] R.C. Arkin. Motor schema-based mobile robot navigation. *The International Journal of Robotics Research*, 8(4):92–112, August 1989.

[4] R.C. Arkin and D.C. MacKenzie. Temporal coordination of perceptual algorithms for mobile robot navigation. *IEEE Transactions on Robotics and Automation*, 10(3):276–286, June 1994.

[5] T. Balch, G. Boone, T. Collins, H. Forbes, D. MacKenzie, and J. Santamaría. Io, Ganymede and Callisto - a multiagent robot trash-collecting team. *AI Magazine*, 16(2):39–51, Summer 1995.

[6] Tucker Balch and Ronald C. Arkin. Motor-schema based formation control for multiagent robotic teams. In *Proc. 1995 International Conference on Multiagent Systems*, pages 10–16, San Francisco, CA, 1995.

[7] Daniel G. Bobrow et al. Common LISP object system. In Guy L. Steele Jr., editor, *Common LISP: The Language*, chapter 28, pages 770–864. Digital Press, 1990.

[8] Mark Bradakis, Thomas C. Henderson, and Joe Zachary. Reactive behavior design tools. In *Proc. IEEE International Symposium on Intelligent Control*, pages 178–183. IEEE, 1992.

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

[10] R.A. Brooks. A robot that walks: Emergent behaviors from a carefully evolved network. *Neural Computation*, 1(2):253–262, 1989. Also MIT AI Memo 1091.

[11] R.A. Brooks. The behavior language: User's guide. AI Memo 1227, MIT, 1990.

[12] Jonathan M. Cameron and Douglas C. MacKenzie. *MissionLab User Manual.* College of Computing, Georgia Institute of Technology, Available via *http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/mlab_manual.ps.gz*, Version 1.0 edition, May 1996.

[13] John P. Chin, Virginia A. Diehl, and Kent L. Norman. Development of an instrument measuring user satisfaction of the human-computer interface. In E. Soloway et al., editors, *Proc. CHI'88, Human Factors in Computing Systems*, pages 213–218. ACM, 1988.

[14] J. Connell. A colony architecture for an artificial creature. AI Tech Report 1151, MIT, 1989.

[15] E. Coste-Maniere, B. Espiau, and E. Rutten. A task-level robot programming language and its reactive execution. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2751–2756, Nice, France, 1992.

[16] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* North-Holland Publishing Company, Amsterdam, The Netherlands, 1989.

[17] J. Firby. Adaptive execution in complex dynamic worlds. Computer Science Tech Report YALEU/CSD/RR 672, Yale, January 1989.

[18] E. Gat. Alfa: A language for programming reactive robotic control systems. In *Proceedings 1991 IEEE International Conference on Robotics and Automation*, volume 2, pages 1116–1121, Sacramento, CA, 1991.

[19] E. Gat. Robust low-computation sensor-driven control for task-directed navigation. In *Proceedings 1991 IEEE International Conference on Robotics and Automation*, volume 2, pages 2484–2489, Sacramento, CA, 1991.

[20] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings AAAI Conference*, San Jose, CA, 1992.

[21] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings AAAI Conference*, pages 677–682, 1987.

[22] Matthew W. Gertz, Roy A. Maxion, and Pradeep K. Khosla. Visual programming and hypermedia implementation within a distributed laboratory environment. *Intelligent Automation and Soft Computing*, 1(1):43–62, 1995.

[23] J. J. Gibson. *The Senses Considered as Perceptual Systems.* George Allen and Unwin Ltd., London, 1968.

[24] J. J. Gibson. Notes on affordances. In E. Reed and R. Jones, editors, *Reasons for Realism: Selected Essays of James J. Gibson*, pages 401–436. Lawrence Erlbaum Associates, 1982.

[25] B. M. Gothard, R. D. Etersky, and R. E. Ewing. Lessons learned on a low-cost global navigation system for the surrogate semi-autonomous vehicle. In *Proceedings SPIE Conference on Mobile Robots VIII*, pages 258–269, Boston, MA., 1993.

[26] J. Gowdy. *SAUSAGES Users Manual.* Robotics Institute, Carnegie Mellon, version 1.0 edition, February 8 1991. SAUSAGES: A Framework for Plan Specification, Execution, and Monitoring.

[27] J. Gowdy. SAUSAGES: Between planning and action. Technical Report Draft, Robotics Institute, Carnegie Mellon, 1994.

[28] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1), January/February 1990.

[29] Thomas C. Henderson. Logical behaviors. *Journal of Robotic Systems*, 7(3):309–336, 1990.

[30] Thomas C. Henderson and Esther Shilcrat. Logical sensor systems. *Journal of Robotic Systems*, 1(2):169–193, 1984.

[31] Deborah Hix and H. Rex Hartson. *Developing User Interfaces.* John Wiley and Sons, New York, 1993.

[32] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–581, 1969.

[33] Charles Antony Richard Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2:335–355, 1973.

[34] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, page 79. Addison-Wesley, 1979.

[35] Marcus J. Huber, Jaeho Lee, Patrick Kenny, and Edmund H. Durfee. *UM-PRS V1.0 Programmer and User Guide.* Artificial Intelligence Laboratory, The University of Michigan, 28 October 1993.

[36] L. P. Kaelbling. An architecture for intelligent reactive systems. Technical Note 400, SRI International, October 1986.

[37] L. P. Kaelbling. Rex programmer's manual. Technical Note 381, SRI International, 1986.

[38] L. P. Kaelbling. Goals as parallel program specifications. In *Proceedings AAAI Conference*, volume 1, pages 60–65, St. Paul, MN, 1988.

[39] L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. *Robotics and Autonomous Systems*, 6:35–48, 1990. Also in *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, P. Maes Editor, MIT Press, 1990.

[40] P. Kahn. Specification & control of behavioral robot programs. In *Proceedings SPIE Conference on Sensor Fusion IV*, Boston, MA., November 1991.

[41] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[42] J. Kosecka and R. Bajcsy. Cooperative behaviors - discrete event systems based approach. Unknown source, 1993.

[43] B. Lee and A.R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, pages 27–39, August 1994.

[44] Jaeho Lee, Marcus J. Huber, Edmund H. Durfee, and Patrick G. Kenny. UM-PRS: An implementation of the procedure reasoning system for multirobot applications. In *Proceedings AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS '94)*, 1994.

[45] D. R. Lefebvre and G. N. Saridis. A computer architecture for intelligent machines. In *Proceedings 1992 IEEE International Conference on Robotics and Automation*, pages 2745–2750, Nice, France, May 1992.

[46] Douglas B. Lenat and R.V. Guha. *Building Large Knowlege-Based Systems.* Addison-Wesley, New York, 1990.

[47] Willie Lim. Sal - a language for developing an agent-based architecture for mobile robots. In *Proceedings SPIE Conference on Mobile Robots VII*, pages 285–296, Boston, MA., 1992.

[48] Michelle A. Lund. Evaluating the user interface: The candid camera approach. In L. Borman et al., editors, *Proc. CHI'85, Human Factors in Computing Systems*, pages 107–113. ACM, 1985.

[49] Damian M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, 9(3):241–256, June 1993.

[50] Damian M. Lyons and M. A. Arbib. A formal model of computation for sensory-based robotics. *IEEE Journal of Robotics and Automation*, 5(3):280–293, June 1989.

[51] Douglas C. MacKenzie. *Configuration Network Language (CNL) User Manual.* College of Computing, Georgia Institute of Technology, Available via *http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/cnl_manual.ps.gz*, Version 1.5 edition, June 1996.

[52] Douglas C. MacKenzie and Ronald C. Arkin. Formal specification for behavior-based mobile robots. In *Proceedings SPIE Conference on Mobile Robots VIII*, pages 94–104, Boston, MA., 1993.

[53] P. Maes. The dynamics of action selection. In *Proceedings Eleventh International Joint Conference on Artificial Intelligence, IJCAII-89*, volume 2, pages 991–997, 1989.

[54] P. Maes. Situated agents can have goals. *Robotics and Autonomous Systems*, 6:49–70, 1990. Also in *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, P. Maes Editor, MIT Press, 1990.

[55] M. J. Mataric. Designing emergent behaviors: From local interactions to collective intelligence. In *Proceedings From Animals to Animats, Second International Conference on Simulation of Adaptive Behavior (SAB92).* MIT Press, 1992.

[56] M.J. Mataric. Minimizing complexity in controlling a mobile robot population. In *Proceedings 1992 IEEE International Conference on Robotics and Automation*, Nice, France, May 1992.

[57] David J. Miller and R. Charleene Lennox. An object-oriented environment for robot system architectures. In *Proc. IEEE International Conference on Robotics and Automation*, volume 1, pages 352–361, Cincinnati, OH, 1990.

[58] M. Minsky. *The Society of Mind.* Simon and Schuster, New York, 1986.

[59] Frank G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice-Hall, New Jersey, 1981.

[60] Lynne E. Parker. Adaptive action selection for cooperative agent teams. In *Proceedings of 2nd International conference on Simulation of Adaptive Behavior*, number 92 in SAB, Honolulu, HA, 1992.

[61] Lynne E. Parker. Local versus global control laws for cooperative agent teams. Technical Report AI Memo No. 1357, MIT, 1992.

[62] Lynne E. Parker. A performance-based architecture for heterogeneous, situated agent cooperation. In *AAAI-1992 Workshop on Cooperation Among Heterogeneous Intelligent Systems*, San Jose, CA, 1992.

[63] Lynne E. Parker. *Heterogeneous Multi-Robot Cooperation.* Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, 1994.

[64] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optimization*, 25(1):206–230, 1987.

[65] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77-1(1):81–97, January 1989.

[66] J.K. Rosenblatt and D.W. Payton. A fine-grained alternative to the subsumption architecture for mobile robot control. In *IEEE INNS International Joint Conference on Neural Networks*, volume 2, pages 317–323, 1989.

[67] S.J. Rosenschein and L.P. Kaelbling. The synthesis of digital machines with provable epistemic properties. Technical Note 412, SRI International, Menlo Park, California, April 1987.

[68] A. Saffiotti, Kurt Konolige, and E Ruspini. A multivalued logic approach to integrating planning and control. Technical Report 533, SRI Artificial Intelligence Center, Menlo Park, California, 1993.

[69] Robert Sandy. *Statistics for Business and Economics.* McGraw-Hill, New York, 1990.

[70] Stanley A. Schneider, Vincent W. Chen, and Gerardo Pardo-Castellote. The ControlShell component-based real-time programming system. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2381–2388, 1995.

[71] K. Schwan et al. A C thread library for multiprocessors. ICS Tech Report GIT-ICS-91/02, Georgia Institute of Technology, January 1991.

[72] M.P. Singh, M.N. Huhns, and L.M. Stephens. Declarative representations of multiagent systems. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):721–739, October 1993.

[73] L. Spector. *Supervenience in Dynamic-World Planning.* Ph.D. dissertation, University of Maryland, Department of Computer Science, 1992. Also Tech Report CS-TR-2899 or UMIACS-TR-92-55.

[74] David B. Stewart and P.K. Khosla. Rapid development of robotic applications using component-based real-time software. In *Proc. Intelligent Robotics and Systems (IROS 95)*, volume 1, pages 465–470. IEEE/RSJ, IEEE Press, 1995.

[75] N. Tinbergen. *The Study of Instinct.* Oxford University Press, London, second edition, 1969.

[76] University of New Mexico. *Khoros: Visual Programming System and Software Development Environment for Data Processing and Visualization.*

[77] F.Y. Wang, K.J. Kyriakopoulos, A. Tsolkas, and G.N. Saridis. A petri-net coordination model for an intelligent mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):777–789, July/August 1991.

[78] Allen C. Ward. *A Theory of Quantitative Inference for Artifact Sets Applied to a Mechanical Design Compiler.* Ph.D. dissertation, MIT, Department of Mechanical Engineering, 1989. Also Tech Report AI Tech Report 1089.

[79] Allen C. Ward and Warren P. Seering. The performance of a mechanical design compiler. AI Tech Report 1084, MIT, 1989.

[80] David A. Watt. An extended attribute grammar for PASCAL. *SIGPLAN Notices*, 14(2):60–74, February 1979.

[81] P Wegner. The Vienna definition language. *ACM Computing Surveys*, 4(1):5–63, 1972.

Vita

Douglas Christopher MacKenzie was born in Hastings, Michigan on January 12, 1963, the son of Douglas Lee and Judith Yvonne MacKenzie. After graduation from Lakewood High School in 1981, he attended Michigan Technological University. He graduated with honors in 1985 with a Bachelor of Electrical Engineering degree. Doug continued at Michigan Tech in the Computer Science department working towards a Master's degree.

While attending Michigan Tech, he worked summers and part time for several years at Hough Brothers, automating feed mills in the poultry industry. Doug left Houghton and moved to Cleveland, Ohio in the summer of 1988 to work as a Software Engineer in Allen-Bradley's Programmable Controller Division. Overlapping writing of his thesis with work, he received the Master's in Computer Science degree in the spring of 1989.

Doug married Karen Beth Brehob in Dearborn, Michigan in the fall of 1988. In the fall of 1990 Doug and Karen moved to Atlanta so Doug could attend Georgia Tech. While at Tech, Doug initiated development of the MissionLab toolset and created the graphical Configuration Editor which brings visual programming to the behavior-based robotics domain. Doug is particularly interested in empowering people who are not experts in robotics to be able to use mobile robots. This goal of making robotics easy to use reflects his belief that it is time to move mobile robotics beyond the research labs. He also believes what is holding robots back is the lack of software that is both powerful and easy to use. MissionLab begins to fill that void.

Doug currently lives near Grand Rapids, Michigan and has formed the Mobile Intelligence Corporation to further his goal of dispersing robotics into everyday life. He can be contacted via E-mail at `doug@mobile-intelligence.com`.