Autonomous Robots, [Volumn Number], 1–25 ([Volumn Year])

Multiagent Mission Specification and Execution

DOUGLAS C. MACKENZIE, RONALD C. ARKIN*

doug@cc.gatech.edu, arkin@cc.gatech.edu

Mobile Robot Laboratory, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332

JONATHAN M. CAMERON

jonathan.m.cameron@jpl.nasa.gov

Jet Propulsion Laboratory, MS 198-326, 4800 Oak Grove Drive, Pasadena, CA 91109

Abstract.
Specifying a reactive behavioral configuration for use by a multiagent team requires both a careful choice of the behavior set and the creation of a temporal chain of behaviors which executes the mission. This difficult task is simplified by applying an object-oriented approach to the design of the mission using a construction called an *assemblage* and a methodology called *temporal sequencing*. The assemblage construct allows building high level primitives which provide abstractions for the designer. Assemblages consist of groups of basic behaviors and coordination mechanisms that allow the group to be treated as a new coherent behavior. Upon instantiation, the assemblage is parameterized based on the specific mission requirements. Assemblages can be re-parameterized and used in other states within a mission or archived as high level primitives for use in subsequent projects. Temporal sequencing partitions the mission into discrete operating states with perceptual triggers causing transitions between those states. Several smaller independent configurations (assemblages) can then be created which each implement one state. The **Societal Agent** theory is presented as a basis for constructions of this form. The Configuration Description Language (CDL) is developed to capture the recursive composition of configurations in an architecture- and robot-independent fashion. The *MissionLab* system, an implementation based on CDL, supports the graphical construction of configurations using a visual editor. Various multiagent missions are demonstrated in simulation and on our Denning robots using these tools.

Keywords: Autonomous Robotics, Mission Specification, Visual Programming

1. Introduction

Reactive behavior-based architectures[3], [8] decompose a robot's control program into a collection of behaviors and coordination mechanisms with the overt, visible behavior of the robot arising from the emergent interactions of these behaviors. The decomposition process supports the construction of a library of reusable behaviors by designers skilled in low-level control issues. Subsequent developers using these components need only be concerned with their specified functionality. Further abstraction can be achieved by permitting construction of assemblages from these low-level behaviors which embody the abilities required to exhibit a complex skill.

Creating a multiagent robot configuration 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. The construction and functionality of the Georgia Tech *Mission-*

* This research is funded under ONR/ARPA Grant # N0001494-1-0215. The Mobile Robot Laboratory is supported by additional grants from the U.S. Army and NSF. Tucker Balch and Khaled Ali have contributed to the *MissionLab* system, and provided assistance with the robot experiments. The SAUSAGES simulation system was provided by Jay Gowdy and Carnegie Mellon University.

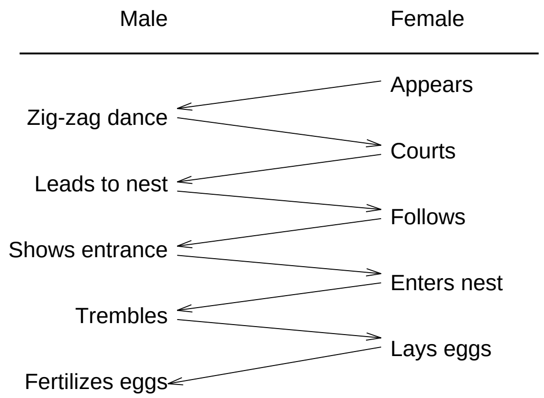


Fig. 1. Sexual behavior of the three-spined stickleback, after [48]

Lab software environment, based upon this procedure, is documented in this article. Support for users in the various stages of mission development (*e.g.*, behavior implementation, assemblage construction, and mission specification) is provided. The primitive behavior implementor must be familiar with the particular robot architecture in use and a suitable programming language such as C++. 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 language knowledge. However, the construction of *useful* assemblages still requires knowledge of behavior-based robot control. Specifying a configuration for the robot team consists of selecting which of the available skills are useful for the targeted environments and missions.

The next section (Section 2) presents the **Societal Agent** theory which forms the theoretical basis for this work. Section 3 presents the Configuration Description Language, the language used to represent configurations by the *Mission-Lab* toolset. Section 4 overviews the *Mission-Lab* toolset. A single robot mission is used to demonstrate binding configurations to different runtime architectures. Section 5 presents a four robot scouting mission developed in simulation to demonstrate the multiagent capabilities of the system, while Section 6 shows a two robot experiment to highlight the retargetability of configurations developed using the system. Section 7 reviews the related work and the summary and conclusions in Section 8 complete the article.

2. The Societal Agent

Thinking of societal agents conjures up 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.

Each buffalo, goose, ant and human can be thought of as possessing a behavior-based controller consisting of a society of agents (cf. [35]). This leads to the view of a flock of geese as a huge society with thousands of interacting agents. 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 real as their component behavioral agents.

This abstraction is equally apparent in military organizations. When commanders refer to their command, they don't speak of individuals, but of the unit abstractions. A company commander might ask for "the strength of platoon Bravo" 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 unified and concrete as an individual soldier.

As a second example of complex agents consider the well-documented sexual behavior of the three-spined stickleback[48] shown in Figure 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"

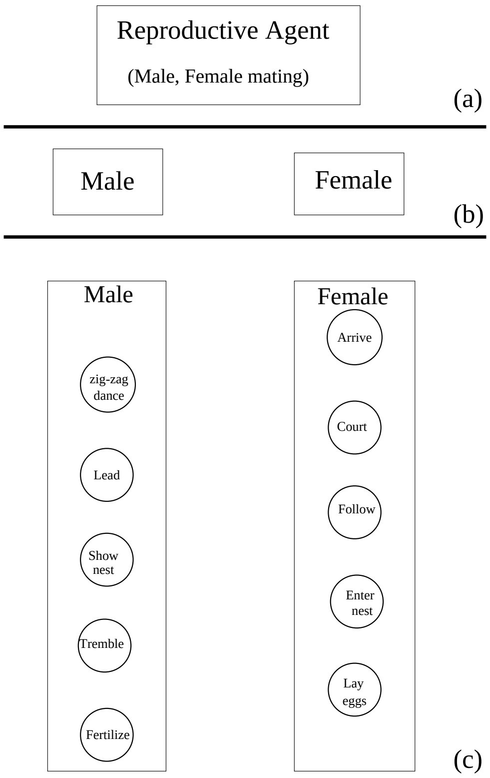


Fig. 2. Schematic of three-spined stickleback mating behavior showing three levels of abstraction. Level *a* represents the mating behavior as a single agent, Level *b* shows the two individual fish, and Level *c* shows the various operating states required to create the mating behavior.

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 represented as individual agents within the **Societal Agent** theory. A coordination operator transcending the individual fish uses these primitive agents to create the sexual behavior apparent in this example.

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 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

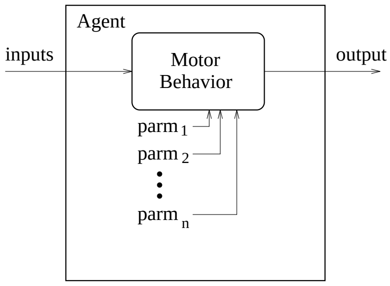


Fig. 3. Schematic diagram of an atomic agent

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 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, separate processes may implement the FSA, perhaps even executing on a computer(s) 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.

2.1. 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 behavioral 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 2.3 below.

The term *agent* has been overused in the literature but seems to most closely convey the essence of what is intended here. 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 to bring the robot closer to the goal. Figure 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 achievable 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 1 formalizes this requirement of computable behaviors.

$$y = f\left(v_1, v_2, \ldots, v_m\right) \mid f \textbf{ is defined } \forall \left(v_1 \times v_2 \times \ldots \times v_m\right) \quad (1)$$

Equation 2 specifies that any entity capable of stimulus-response behavior can be treated as a distinct agent.

$$Agent \equiv Behavior\,(Stimulus) \quad (2)$$

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. Therefore we expand the definition of an agent presented in Minsky's "Society of Mind" [35] to encompass all situated computable functions. For the function to 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.

2.2. Primitive Behavior Classes

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

Sensors are hardware dependent and are not present in a 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 a source for the configuration dataflows.

Similar to sensors, actuators are not present in a free configuration. Instead, output binding points are used to mark where the actuator will be connected during binding. The output binding point is a dataflow sink in the configuration.

Perceptual modules function as virtual sensors[18], [19] 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.

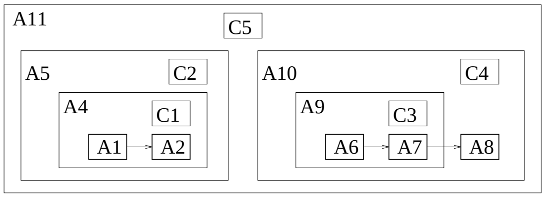


Fig. 4. Schematic diagram of a configuration

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.

2.3. 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 4 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 can be denoted functionally. For example, in Figure 4, 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 provides a complete expansion of the recursive construction of Figure 4.

$$C_5\left(C_2\left(C_1\left(A_1, A_2\right)\right), C_4\left(C_3\left(A_6, A_7\right), A_8\right)\right) \tag{3}$$

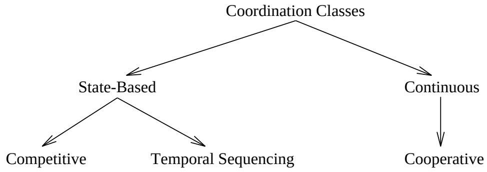


Fig. 5. Classes of Coordination Modules

2.4. Classes of Coordination Modules

A coordination module modifies the activities of the group of agents it is connected to, exploiting the strengths of each to execute a specific task. This intervention may range from none (the null case) to explicit coordination of each and every action of its members. Figure 5 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.

State-based coordination mechanisms partition the agents they are managing into distinct groups, allowing only 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.

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.

Competitive The competitive style of coordination selects a distinguished subset of the society to activate based upon 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[31], and the subsumption architecture[8].

Figure 6 shows a Colony Architecture network [12] with three behaviors and two suppression nodes (labeled **S**). The design is that if behavior 3 has something to contribute, then it overwrites

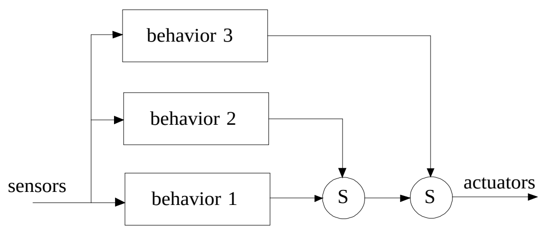


Fig. 6. Example Colony Architecture Network

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.

Consider how Figure 6 would be represented in the **Societal Agent** architecture. First, define the functions computing the three behavior policies as *behav1*, *behav2*, *behav3* 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 4 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.

$$suppress(hi, hi\_valid, low) = \begin{cases} hi & \text{if } hi\_valid \\ low & \text{otherwise} \end{cases} \quad (4)$$

Using Equation 4 twice allows specification of Figure 6 functionally as shown in Equation 5.

$$\begin{aligned} &suppress(behav3, valid3, \\ &\qquad suppress(behav2, valid2, behav1)) \end{aligned} \quad (5)$$

Notice that, based on the definition of *suppress* in Equation 4, *behav3* correctly dominates when it is valid, otherwise *behav2* dominates *behav1* when it has valid data and *behav1* is only allowed to generate an output when both of the other behaviors are not generating useful data.

Temporal Sequencing Temporal sequencing [4] is a state-based coordination mechanism which uses a Finite State Automaton (FSA)[20], [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 agents based on current processing needs and environmental conditions.

Equation 6 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.

$$f_{seq}(a_1, a_2, \ldots, a_m, \alpha) = a_i \mid \text{state } i \text{ is active in } \alpha \quad (6)$$

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 Finite State Automaton (FSA) α is specified by the quadruple[20] (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 the completion of the sensorimotor task.

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 7 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.

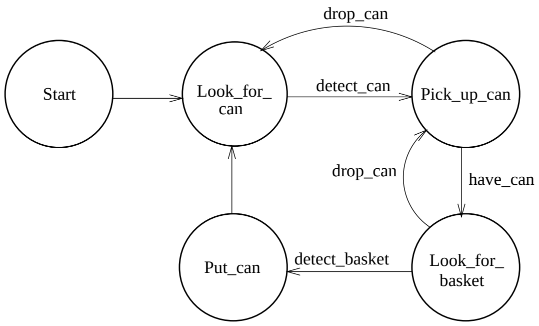


Fig. 7. FSA for a trash collecting robot

The FSA in Figure 7 would be represented by the quadruple

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

Notice that in this case there are no accepting states since during the competition the robots ran until they were manually turned off. The transition function δ for the trash collecting FSA is specified in Table 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.

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. Vector summation in the AuRA[3], [2] architecture is such a mechanism. The AuRA gain-based cooperation operator can be represented functionally as a weighted vector summation, as shown in Equation 7. 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.

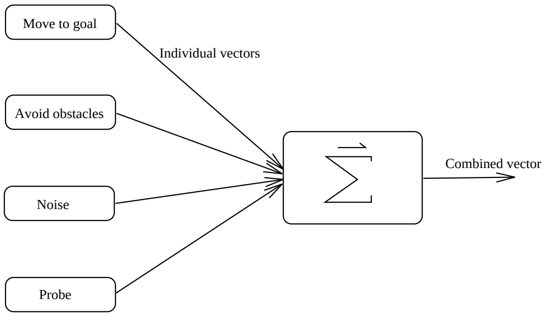


Fig. 8. Vector Summation in AuRA

$$f(\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n, w_1, w_2, \ldots, w_n) = \vec{\sum_{k=1,n}} (\vec{v}_k * w_k) \tag{7}$$

Figure 8 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 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.

3. The Configuration Description 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

Table 1. Tabular representation of δ function for Figure 7 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	$\emptyset$

architecturally independent configurations which are not couched in the terms of some 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. This lifting of 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. 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.

3.1. 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

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

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.

$$\text{Add}\,(\text{A} = \text{Add}\,(\text{A} = \{3\}, \text{B} = \{5\})\,, \text{B} = \{1\})\,; \quad (8)$$

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.

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

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

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

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

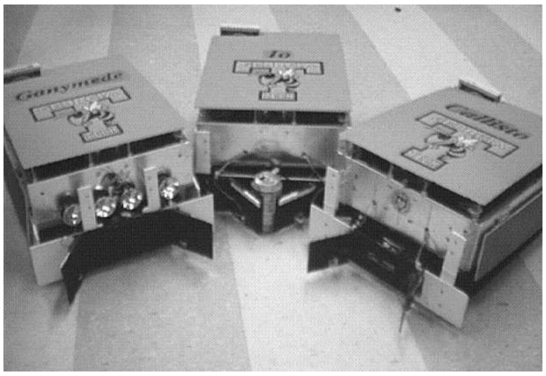


Fig. 9. Three trash collecting robots from AAAI94[5]

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 8 which demonstrated the nesting process. We can turn that agent into an assemblage as follows:

$$\begin{aligned}&\text{defAgent Add8 from Add}(\text{A} = \\ &\qquad \text{Add}(\text{A} = \{3\}, \text{B} = \{5\}), \text{B} = \{\hat{}\ \text{Val}\});\end{aligned}$$

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. The complete definition of CDL can be retrieved from `www.cc.gatech.edu/ai/robot-lab/research/MissionLab` as part of the *MissionLab* documentation.

3.2. Example Janitor Configuration

The use of CDL will now be further demonstrated 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.

Reconsider the trash collecting state-transition diagram Figure 7 from Section 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 9 to retrieve soda cans and place them in wastebaskets[2]. 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 10 represents the three cleanup robots as a single jan-

```
   /* 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;
```

Fig. 10. 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.

itor 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. Conversion 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 priori* that it will control three robots, an early commitment to a three agent society is taken 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 information provides information to the CDL compiler allowing it to generate more efficient code. The `list` keyword defines the input parameter as a list of `movement` 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 9, 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. One of the design requirements for this configuration was to utilize the three available robots as an independent homogeneous group. Figure 10 demonstrates satisfaction of that design goal.

The next step is to define the implementation of the `cleanup` prototype. This behavior is implemented as a state-based coordination operator. The FSA presented in Figure 7 was programmed graphically using the configuration editor which will be presented in Section 4. The code generated by the editor isn't particularly interesting and will be left out for the sake of brevity. A screen snapshot showing the FSA in the editor appears as Figure 17.

The FSA is implemented with agents for each operating state and perceptual trigger. The FSA coordination operator has input connections from each of the state agents and each of the trigger agents. The output of the agent implementing the current state is passed through as the output of the FSA. The FSA δ transition function is used

```
   /*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 = {ˆ }),
2c.  inputs[C] = AvoidObstacles(objects = {ˆ }),
2d.  inputs[D] = AvoidRobots(objects = {ˆ }),

     /* 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 = {ˆ });
```

Fig. 11. Partial CDL description of `LookForCan` agent

along with the perceptual trigger inputs to determine the new (possibly the same) state.

Figure 11 provides a definition of `LookForCan` as a representative example of the motor agents implementing the states in the `cleanup` FSA. Statement 1 defines the `WeightedCombination` coordination operator which computes a weighted combination of its inputs.

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 groups output.

The `AvoidObstacles` agent is shown in Figure 12. Statement 1 defines a new class of input binding points and gives them the name `sense_objects`. The input 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 compared to the other primitives. CDL attempts to keep the syntax similar for all objects.

```
   /* 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 = {ˆ }),

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

Fig. 12. Partial CDL description of `AvoidRobots` agent

Statement 2 creates the `AvoidRobots` agent as an instance of the primitive `AvoidObjects`. This primitive motor module uses a `horizon` and `safety_margin` to determine the strength of its reaction to objects. Statement 2c specifies the list of objects that AvoidRobots will respond to is constructed by `FilterObjectsByColor`. This perceptual filter module removes 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 a particular primitive is to be available in, an agent prototype definition describing the interface to the module is used to make the primitive available for use by 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.

3.3. Binding

One of the strengths of CDL is its support for retargeting of configurations through the use of generic configurations and explicit hardware binding. The binding process maps an abstract config-

```
   /* 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;
```

Fig. 13. CDL description of `janitor` configuration bound to the three robots

uration 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 bindings. 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 sensors available or two robots, each with a single class of 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 to simplify the configurations. 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 styles of coordination operators can be architecture specific.

Figure 13 shows the relevant CDL code for the `janitor` after it has been bound to the three robots shown in Figure 9. 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

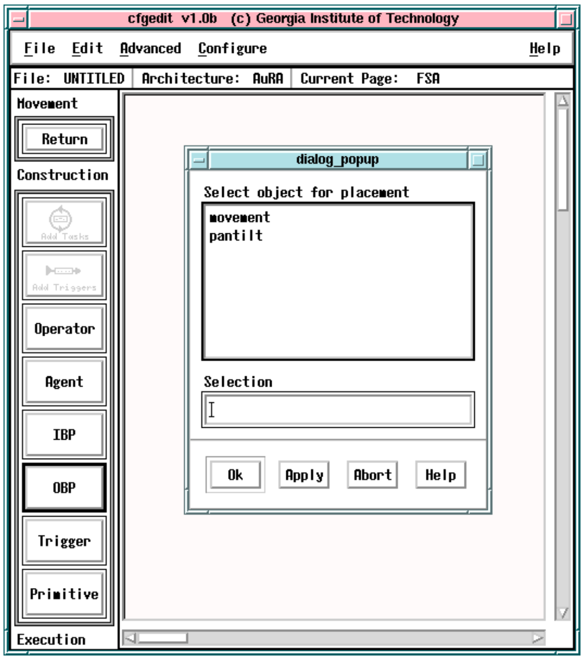


Fig. 14. CfgEdit menu to pick output binding point

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

4. *MissionLab*: An Implementation

The *MissionLab* toolset has been developed based on the Configuration Description Language. It includes a graphical configuration editor, a multiagent simulation system, and two different architectural code generators. The graphical Configuration Editor (*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 nature by allowing creation of coordinated assemblages of components which are then treated as atomic higher-level components available for later reuse. The Configuration Editor allows deferring commitment (binding) to a particular robot architecture or specific vehicles until the configuration has been developed. This explicit binding step simplifies developing a configuration which may be deployed on one of several vehicles which may each require use of a specific architecture. The process of retargeting a configuration to a different vehicle when the available vehicles or the system requirements change is similarly eased. The capability exists to generate either code for the ARPA Unmanned Ground Vehicle (UGV) architecture or instead for the AuRA architecture and executable within the *MissionLab* system. The AuRA executables drive both simulated robots and several types of Denning vehicles (DRV-1, MRV-2, MRV-3). The architecture 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.

4.1. Designing Configurations with MissionLab

To demonstrate use of the toolset we will redevelop the janitor configuration just described. Recall that the design requirements for the configuration called for the creation of a janitorial robot which would 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. We will refer to the configuration fulfilling these design requirements as the *trashbot* configuration. Figure 7 and Table 1 presented the operating states and specified the FSA required to implement the task. Recall that the five operating states are: *Start*, *Look_for_can*, *Pick_up_can*, *Look_for_basket*, and *Put_can*.

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

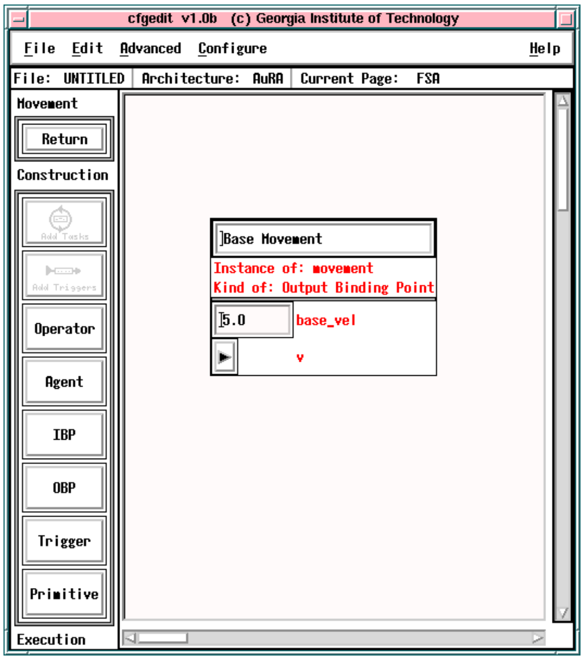


Fig. 15. The output binding point for the wheels placed in the workspace.

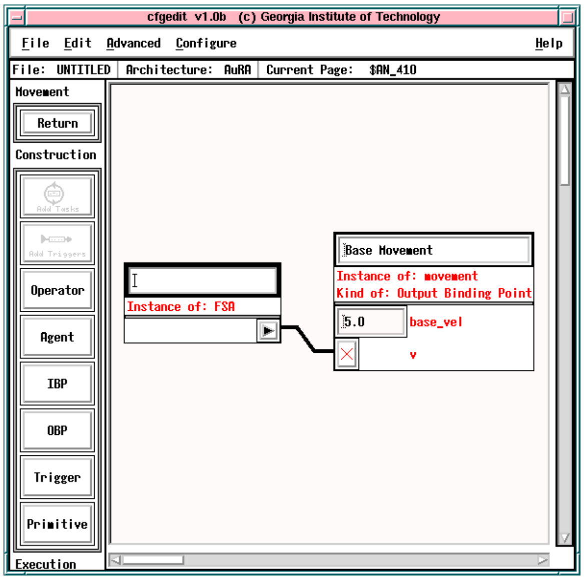


Fig. 16. An FSA coordination operator has been attached as the behavior defining the *trashbot* robot.

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.

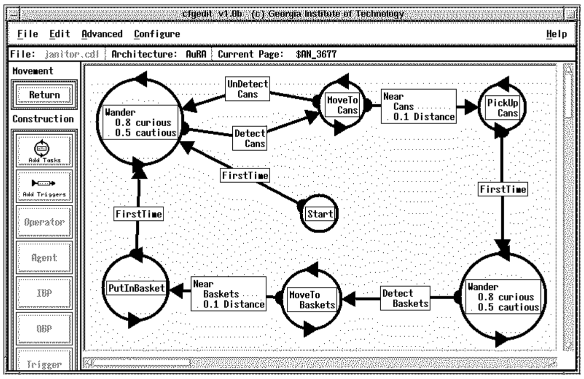


Fig. 17. The state diagram defining the FSA for the trash collecting robot. Notice the similarity with Figure 7.

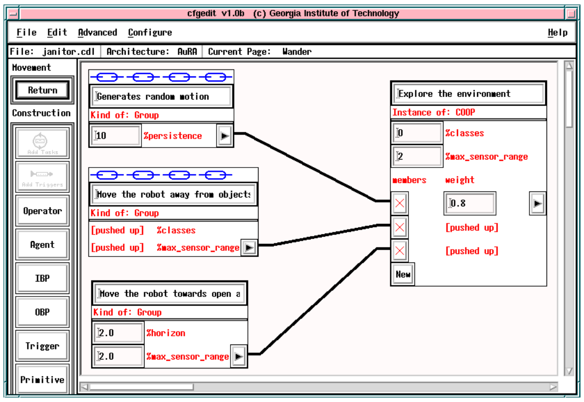


Fig. 18. The *Wander* skill assemblage used to find the cans and home base.

To start construction an output binding point is placed in the workspace where the actuator to drive the wheels of the robot around will later be attached. Figure 14 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 15 shows the workspace with the binding point in place.

During the design process it was determined that the *recycle_cans* skill required at 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 16 shows the workspace after this connection is made.

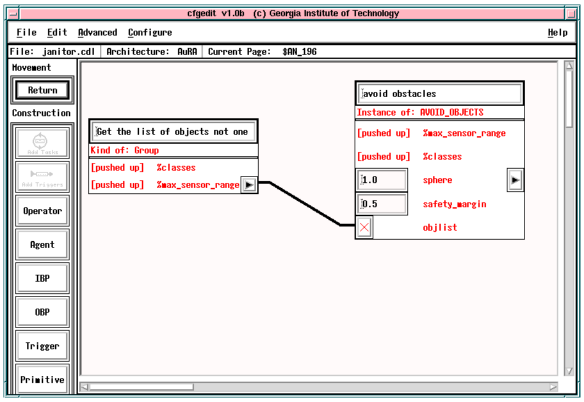


Fig. 19. The *avoid_static_obstacles* behavior used in the *Wander* skill

The Configuration Editor supports the graphical construction of FSAs. In Figure 17, the operator has moved into the FSA workspace and defined its state diagram. The figure shows the state machine implementing the *recycle cans* skill for the trash collecting robot. Compare Figure 17 with the state diagram shown in Figure 7. The circles represent the various operating states within the FSA with the rectangle in the center of each circle listing 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. Clicking on the states or transitions with the right button brings up the list of assemblages or perceptual triggers to choose from. Clicking the middle button on an icon brings up the list of parameters for that particular assemblage or trigger, allowing parameterization.

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 generally are cooperative constructions. In this case, we will examine the wander assemblage. Notice that 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.

Figure 18 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 19 the operator has moved into the *avoid_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 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.

Returning to the top level of the configuration we now have defined the behavior of the *recycle_cans* assemblage. At this point it is a good time to bind the configuration to a specific robot, in this case one of our Blizzards. Clicking on the "Bind" button starts this process. First, a popup menu allows selecting to which architecture the configuration will be bound. This determines the code generator and runtime system that will be used. In this case we will choose the AuRA ar-

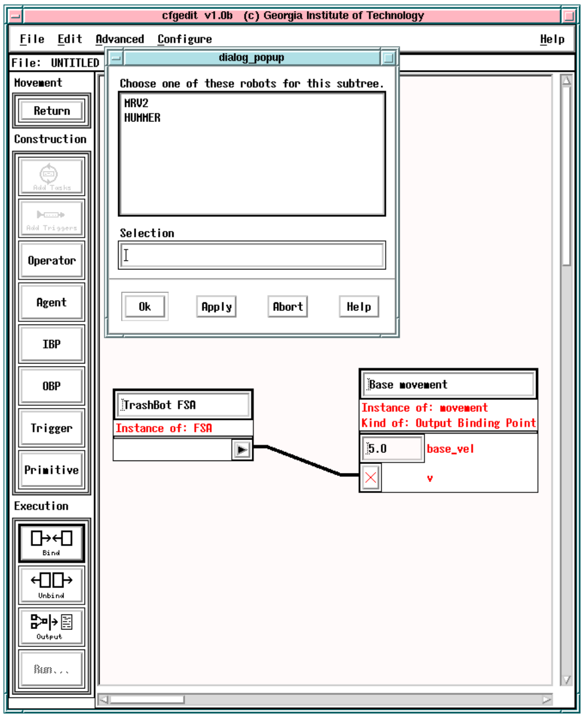


Fig. 20. The robot selection menu, presented during binding

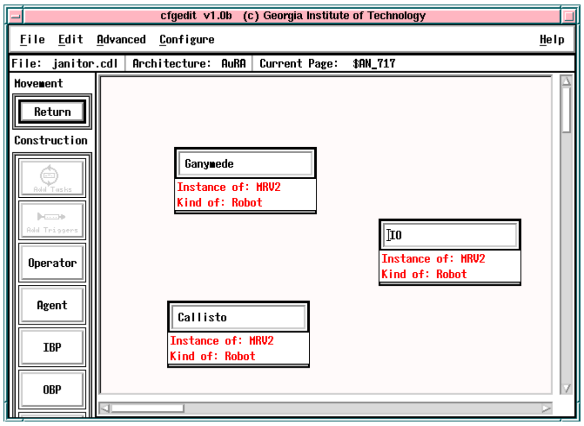


Fig. 21. trashbot configuration with three robots

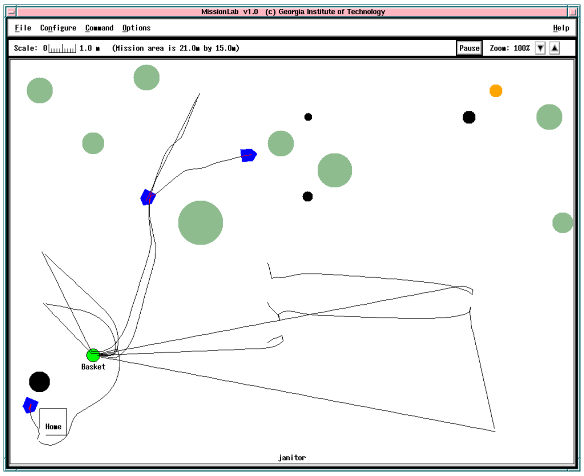


Fig. 22. The trashbot configuration executing in simulation. The cans have all been gathered and returned to the circle labeled **basket**. The trails show the paths the robots took completing the mission. The remaining shaded and solid circles of various sizes represent simulated obstacles within the arena. The robots treated both classes of obstacles the same during this mission.

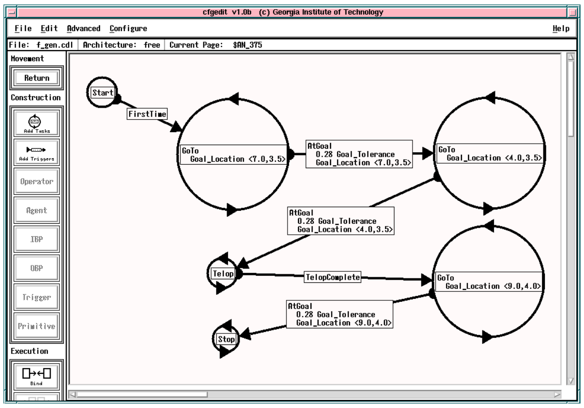


Fig. 23. Generic configuration suitable for binding to either architecture

chitecture (the other current choice is the UGV architecture).

Next, the system prompts for selection of a robot to be bound to the assemblage (Figure 20). In this case we will 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 21 shows the resulting configuration with three robots specified.

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 highlight usage of the Configuration Editor. The next step in the development process is to generate a robot executable and begin evaluation of the configuration.

When the configuration is bound to the AuRA architecture the CDL compiler generates a Configuration Network Language (CNL) specification

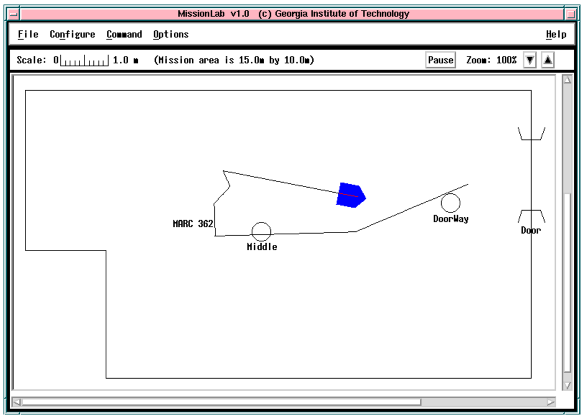


Fig. 24. The configuration from Figure 23 executing in the *MissionLab* simulator. The two circles are landmarks in the map overlay which were not used during this mission.

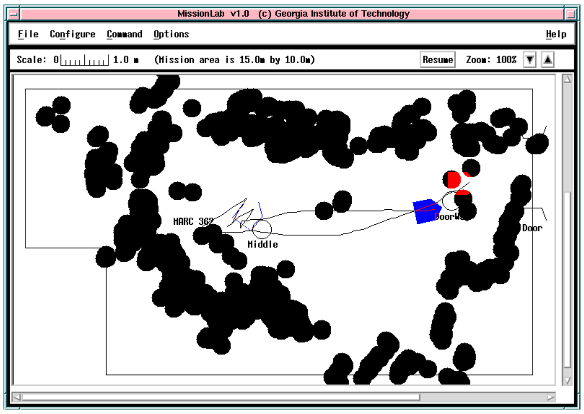


Fig. 25. Snapshot of operator console after executing the mission shown in Figure 23 on a real MRV-2 Denning robot. The dark circles mark obstacle readings during the run in the Mobile Robot Lab. The same map overlay was used as in the simulated mission. in Figure 24.

of the configuration as its output. CNL is a hybrid dataflow language[25] using large grain parallelism where the atomic units are arbitrary C++ functions. CNL adds dataflow extensions to C++ which eliminate the need for users to include communication code. The output of the CNL compiler is a C++ file which is compiled into a robot executable.

MissionLab includes an operator console used to execute missions in the AuRA architecture by simulation or with real robots. The operator display shows the simulation environment, the locations of all simulated robots, and the reported positions of any real robots. Figure 22 shows the `Janitor` configuration executing in simulation using the AuRA runtime architecture. Within the main display area robots, obstacles, and other features are visible. The solid round black circles are obstacles. The three robots are moving actively gathering trash and the paths they have taken are shown as trails. For more details on *MissionLab*, see [11].

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.

Figure 23 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 and then it returns to the starting area before halting. First the configuration is bound to the AuRA architecture and deployed on the MRV-2 robots. Figure 24 shows the configuration executing in the *MissionLab* simulation system. Figure 25 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 25 appears very similar to Figure 24 even though the first reflects a real robot run and the second shows a simulated execution. Figures 26 and 27 show the robot during the mission.

As a final demonstration, the configuration is unbound and then 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 29 is a screen snapshot of the SAUSAGES simulator after execution of the mission. The robot does not leave trails in this simulator, although the waypoints are connected by straight lines to show the projected route.

Fig. 26. Photo of robot executing the Figure 23 mission at start/finish location near the `doorway` landmark.

Fig. 27. Photo of robot executing the Figure 23 mission at Teleop location near the `middle` landmark.

5. Simulated Robot Scouting Mission

A four robot scouting mission has been constructed and evaluated in simulation. A `MoveInFormation` behavior 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 30 shows the state transition diagram used in the mission. In this case, explicit coordinates are used as destinations. Notice that 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.

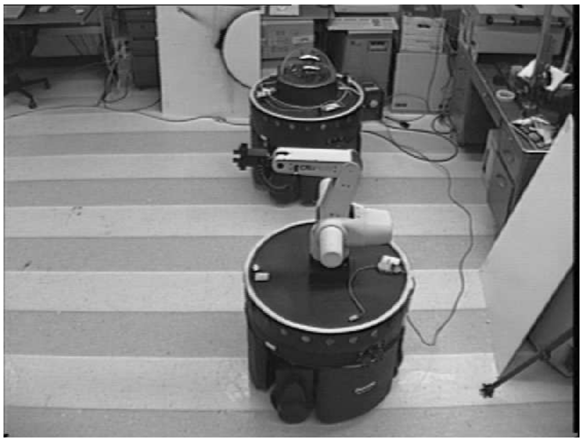
Figure 31 shows the robots during execution of the scout mission in the *MissionLab* simulator. The robots started in the bottom left corner moving up in line formation, then moved right in column formation, and are now moving to the right in a wedge formation. Figure 32 shows the completed mission with the robots occupying the objective in a diamond formation.

6. Indoor Navigation with Two Robots

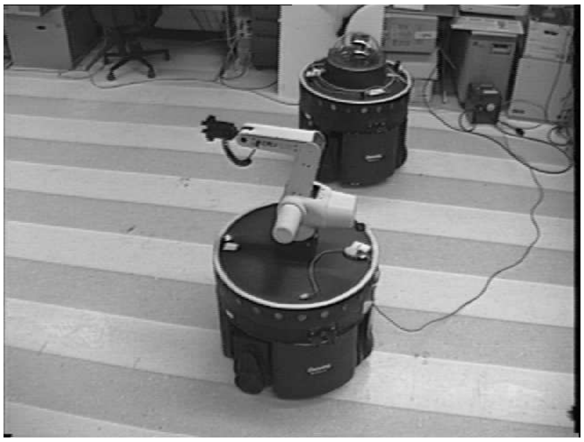
Figure 33 shows *MissionLab* with the overlay representing the Georgia Tech Mobile Robot Lab loaded. The gap in the upper right represents the door to the laboratory. The goal circles were positioned arbitrarily to use as targets for the move-to-goal behavior in the mission. The pair of robots are shown 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 28 shows a sequence of photographs of the robots executing this mission.

7. Related Work

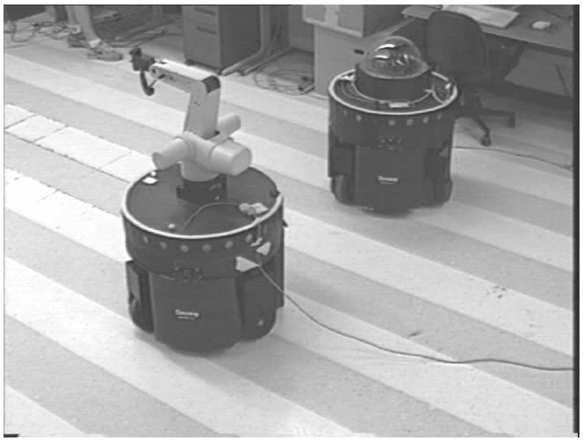
There are many robot programming languages with which CDL must compete, but several are rather loosely defined extensions to standard programming languages. The Robot Independent Programming Language (RIPL) from Sandia[34] is built on top of C++. The SmartyCat Agent Language (SAL) developed at Grumman[27], the Behavior Language (BL)[10] from MIT targeting the Subsumption architecture[8], and Kaelbling's



1. Robots in start location

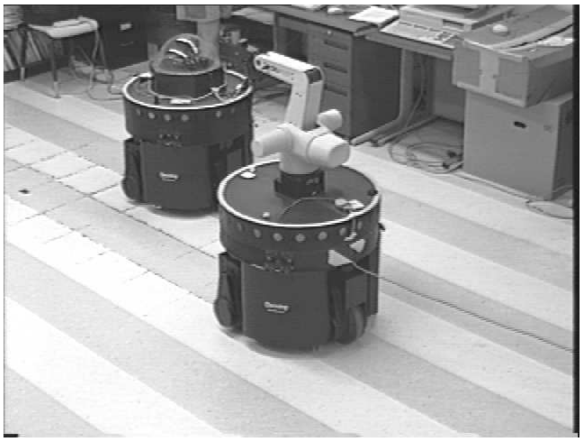


2. Moving towards `dest1`



3. Robots at location `dest1`

4. Moving towards `dest2`



5. Robots nearing location `dest2`

6. Completed mission

Fig. 28. Pictures of the robot executing the two agent mission

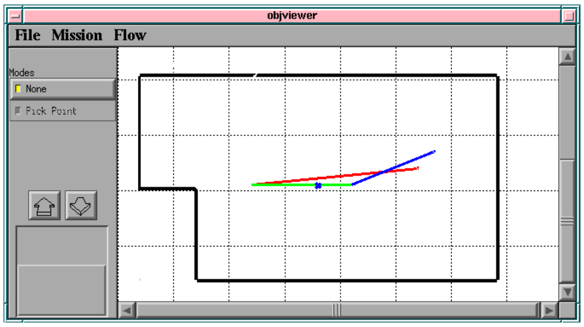


Fig. 29. Snapshot of SAUSAGES simulation display after executing the mission shown in Figure 23. Notice the same general route was taken by the robot.

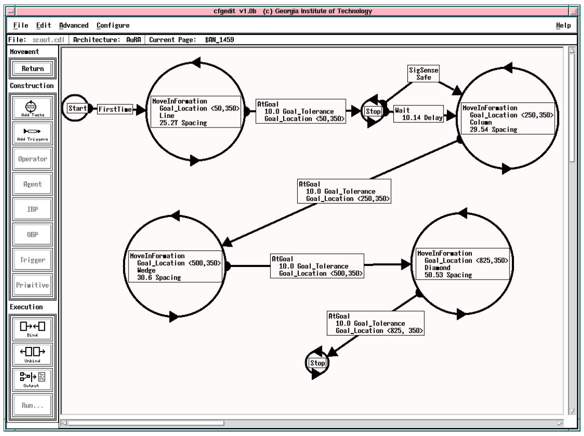


Fig. 30. The state transition diagram for the scouting mission

REX[22], [43] language all are based on Lisp[7]. These languages suffer from a co-mingling of the configuration with the specification of the primitives. They also bury hardware specific binding information within the implementations of individual primitives. This greatly increases the amount of effort necessary to change the configuration or to re-deploy it on other robots. REX does support semantic analysis to formally prove run-time properties[43] if a detailed environmental model is available.

Another class of existing languages are logic based. Gapps[23], [24] is a declarative language providing goals for an off-line planner which generates REX programs for execution. Multivalued logic is used to control the robot Flakey, where the control program takes the form of a fuzzy logic rule-based system. Multivalued logic also has been used to analyze how behaviors combine[44]. Given that each behavior has an explicit applica-

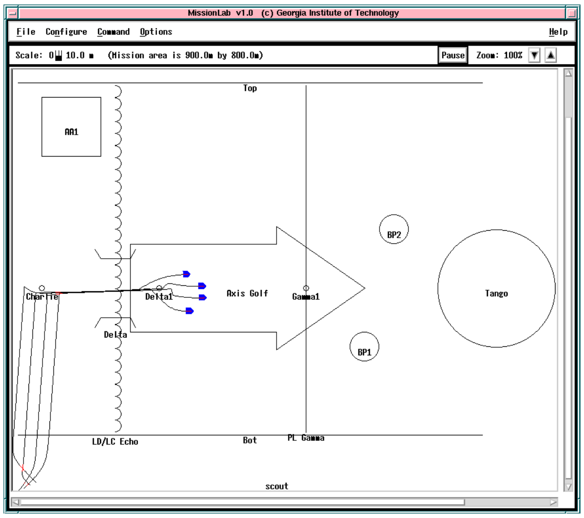


Fig. 31. The mission executing in the *MissionLab* simulator. The robots started in the bottom left corner moving up in line formation, then moved right in column formation, and are now moving to the right in a wedge formation.

bility context, multivalued logic can be used to determine the context of the resulting behavioral assemblage.

The Robot Schemas (RS) architecture[29] is based on the port automata model of computation. Primitive sensorimotor behaviors are called basic schemas and a group of schemas can be interconnected to form an assemblage, which is treated as a schema in subsequent constructions. The assemblage mechanism facilitates information hiding, modularity, and incremental development. The computational model that the RS language embodies is rigorously defined, facilitating formal descriptions of complex robotic systems. CDL expands on the concept of recursive composition of sensorimotor behaviors, apparent here in the assemblage construct.

The successor to RS is called RS-L3[28] and combines RS with ideas from the Discrete Event Systems(DES)[41], [40] community. DES models systems as finite state automaton where the perception-action cycle is broken into discrete events to simplify modeling. RS-L3 is able to capture the specification of a robot control program and the situational expectations, allowing analysis of the system as a whole.

CDL is a generic specification language which is robot and runtime architecture independent. We

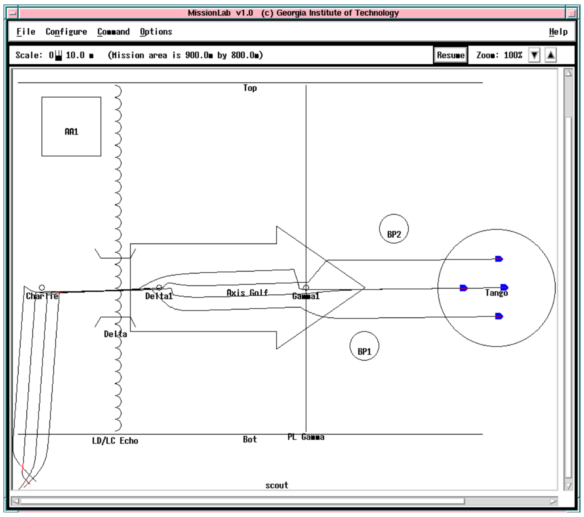


Fig. 32. The completed scout mission with the robots occupying the objective in a diamond formation.

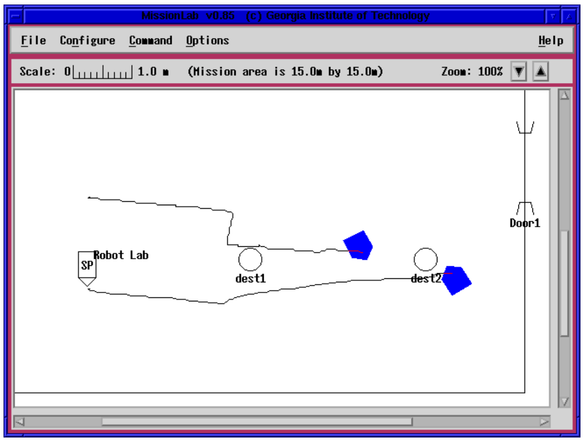


Fig. 33. *MissionLab* showing the operator console after execution of a simple two robot mission. The robots start on the left edge of the lab and proceed to the `dest1` point in line formation. They then continue to location `dest2` using column formation. They are shown in there final positions, with the trails marking the path they each traversed.

now survey the important robot architectures with an eye towards their suitability as targets for CDL.

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. AuRA is a hybrid system spanning the range from deliberative to reactive modes of execution. In configurations generated by *MissionLab*, the human replaces the deliberative system by crafting a suitable behavioral assemblage which completes the desired task.

The Subsumption Architecture[8] is probably the most widely known behavior-based mobile robot architecture. It uses a layering construction where layers embody individual competencies and new skills are added by adding new layers on top of the existing network. The layers can take control when appropriate by overriding layers below them. The subsumption architecture has been used to construct complicated mobile robots[9] as well as societies of robots[32], [33]. All coordination in subsumption occurs via prioritized competition, precluding any cooperative interaction between behaviors. Due to the lack of support for cooperative coordination in Subsumption, only a subset of the possible CDL configurations can be targeted to this architecture.

There are a number of architectures which have grown out of subsumption and share similar constraints on targeting from CDL. Connell's colony architecture[12] and Parker's ALLIANCE architecture[36], [37], [38], [39] are two examples. Connell's efforts with the can collecting robot demonstrated that a collection of behaviors can perform complex tasks. Parker's work went a step further to show that cooperation can also occur between robots without explicit coordination strategies.

Other researchers have evaluated certain types of coordination strategies. Maes has used spreading activation[31], [30] to arbitrate which behaviors are allowed to control the system and to interject goals into reactive systems. Behaviors are connected via activation and inhibition links with activation flowing into the system from both sensors and system goals, tending to activate agents which are both currently applicable and useful in achieving the system goals. The behavioral coordinator for the ARPA UGV robots is called the Distributed Architecture for Mobile Navigation (DAMN) arbiter and uses a fuzzy logic approach to cooperative coordination. Each behavior has a 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 a fine-grained alternative to the subsumption architecture [42].

The System for AUtonomous Specification, Acquisition, Generation, and Execution of Schemata (SAUSAGES)[17], [16] provides a specification language as well as run-time execution and monitoring support. A variant of SAUSAGES called MRPL is used in the ARPA Unmanned Ground Vehicles (UGV's). In a SAUSAGES program behaviors are operations which move along a link in the plan. SAUSAGES is supported as a target architecture from CDL, allowing testing configurations constructed with this system on the SAUSAGES simulation system available from CMU.

The UM-PRS system[26], [21], [14] is a general purpose reasoning system able to exploit opportunism as the robot moves through the environment. UM-PRS is important since it has been considered for inclusion as the behavioral controller in the UGV architecture.

Reactive Action Packages[13] (RAPs) are intended to be used as a set of primitive actions by a deliberative planner. Several different methods for accomplishing an action will exist within a given RAP and at execution time, one of the methods is chosen as most applicable based on precondition tests. Each RAP coordinates itself until failure or success when the planner regains control.

Supervenience[46] 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 targeted for use in dynamic-world planners. Supervenience is the formalization of the process of partitioning a control structure into abstraction levels.

The inspiration for the graphical construction of configurations in *MissionLab* was the Khoros[49] 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". The program can be executed and the results be displayed within the system.

The Onika system[47], [15] from CMU is optimized for the rapid graphical construction of control programs for robot arms. It is tightly integrated with the Chimera real-time operating system, also from CMU. Programs are constructed at two levels: The Engineering level uses an electronic schematic style of presentation to build high level primitives which appear as puzzle pieces when iconified. At the user level, programs are constructed by placing a linear sequence of puzzle piece icons in the workspace. Compatibilities between primitives are represented on the input and output side via different shapes and colors. 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. 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.

ControlShell[45] is a commercial graphical programming toolset from Real-Time Innovations used to construct real-time control programs. It is very similar in presentation to the Engineering level of Onika, having a similar schematic-like look and feel. A dataflow editor is used to graphically select and place components into the workspace and connect them into control systems. The state programming editor supports graphical specification of state transition diagrams. ControlShell supports 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. The lack of support for arbitrary recursive construction limits reuse and information hiding in complicated designs.

8. Conclusions and future work

The "Society of Mind"[35] 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** theory has been presented which formalizes this view-point. 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 Configuration Description Language was developed to capture 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.

CDL strictly partitions specification of a configuration from the implementation of the underlying primitives. This separation in structure supports the separation of presentation required to empower non-programmers with the ability to specify complex robot missions. 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.

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.

To validate the usefulness of the concepts and implementations presented in this paper, simulated and real runs of configurations were presented. The *MissionLab* system was demonstrated as part of ARPA UGV Demo C. This tech demo involved creating a configuration specifying the behavior set and mission sequence for a pair of automated off-road vehicles conducting several scouting tasks.

Notes

1. *MissionLab* is available in source and binary form at *http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab*
2. Due to hardware limitations the robots in Figure 9 only placed the cans near the wastebaskets

References

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*. PhD 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. MacK enzie, 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 and 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. R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.
9. 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.

10. R.A. Brooks. The behavior language: User's guide. AI Memo 1227, MIT, 1990.
11. 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.
12. J. Connell. A colony architecture for an artificial creature. AI Tech Report 1151, MIT, 1989.
13. J. Firby. Adaptive execution in complex dynamic worlds. Computer Science Tech Report YALEU/CSD/RR 672, Yale, January 1989.
14. Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings AAAI Conference*, pages 677–682, 1987.
15. 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.
16. 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.
17. J. Gowdy. Sausages: Between planning and action. Technical Report Draft, Robotics Institute, Carnegie Mellon, 1994.
18. Thomas C. Henderson. Logical behaviors. *Journal of Robotic Systems*, 7(3):309–336, 1990.
19. Thomas C. Henderson and Esther Shilcrat. Logical sensor systems. *Journal of Robotic Systems*, 1(2):169–193, 1984.
20. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, page 79. Addison-Wesley, 1979.
21. 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.
22. L. P. Kaelbling. Rex programmer's manual. Technical Note 381, SRI International, 1986.
23. L. P. Kaelbling. Goals as parallel program specifications. In *Proceedings AAAI Conference*, volume 1, pages 60–65, St. Paul, MN, 1988.
24. 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.
25. B. Lee and A.R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, pages 27–39, August 1994.
26. 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.
27. 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.
28. 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.
29. 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.
30. P. Maes. The dynamics of action selection. In *Proceedings Eleventh International Joint Conference on Artificial Intelligence, IJCAII-89*, volume 2, pages 991–997, 1989.
31. 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.
32. 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.
33. 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.
34. 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.
35. M. Minsky. *The Society of Mind.* Simon and Schuster, New York, 1986.
36. 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.
37. Lynne E. Parker. Local versus global control laws for cooperative agent teams. Technical Report AI Memo No. 1357, MIT, 1992.
38. 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.
39. Lynne E. Parker. *Heterogeneous Multi-Robot Cooperation.* PhD dissertation, MIT, Department of Electrical Engineering and Computer Science, 1994.
40. 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.
41. P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77-1(1):81–97, January 1989.
42. 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.
43. 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.

44. 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.

45. 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.

46. L. Spector. *Supervenience in Dynamic-World Planning*. PhD dissertation, University of Maryland, Department of Computer Science, 1992. Also Tech Report CS-TR-2899 or UMIACS-TR-92-55.

47. 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.

48. N. Tinbergen. *The Study of Instinct*. Oxford University Press, London, second edition, 1969.

49. University of New Mexico. *Khoros: Visual Programming System and Software Development Environment for Data Processing and Visualization.*

Douglas C. MacKenzie received the B.S. degree in Electrical Engineering in 1985 and the M.S. degree in Computer Science in 1989, both from Michigan Technological University. He was employed as a Software Engineer in Allen-Bradley's Programmable Controller Division in Cleveland, Ohio from 1988 until 1990. He has been a Ph.D. student in the College of Computing at Georgia Tech since 1990. His research interests center around artificial intelligence as applied to mobile robotics. While at Georgia Tech, he has been working as a graduate research assistant in the mobile robotics laboratory.

Ronald C. Arkin is Associate Professor and Director of the Mobile Robot Laboratory in the College of Computing at the Georgia Institute of Technology. Dr. Arkin's research interests include reactive control and action-oriented perception for the navigation of mobile robots, robot survivability, multi-agent robotic systems, learning in autonomous systems, and cognitive models of perception. He has over 80 technical publications in these areas. Funding sources have included the National Science Foundation, ARPA, Savannah River Technology Center, and the Office of Naval Research. Dr. Arkin is an Associate Editor for IEEE Expert, and a Member of the Editorial Board of Autonomous Robots. He is a Senior Member of the IEEE, and a member of AAAI and ACM.

Jonathan M. Cameron received his B.S. degree in Mechanical Engineering in 1979 and his M.S. in Engineering Science and Mechanics in 1980, both at Georgia Institute of Technology. In 1981, he was employed by Jet Propulsion Laboratory (JPL) in Pasadena, California, where he was involved in spacecraft and ground vehicle control research. In 1988, he left JPL on an educational leave of absence to pursue a Ph.D. in Mechanical Engineering at Georgia Institute of Technology. He completed his Ph.D in December, 1993. From then until 1995, he was employed by the College of Computing at Georgia Tech as a research scientist investigating multiagent robotics. In July of 1995, he returned to JPL and now works in the Automation and Control section where is is working on several advanced space exploration concepts. His research interests include robotics, dynamics, kinematics, controls, and software development.