Reactive Behavior Design Tools

Mark Bradakis, Thomas C. Henderson and Joe Zachary

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

1 Introduction

Designers of autonomous systems need to make sure that the controlling behaviors are coherent, robust, and adequate for the task at hand [2, 4]. Autonomous agents need to continuously interact with their environment, the essential element of reactive behavior[9]. The reactive behavior of an autonomous agent can be described as collections of logical behaviors, each member of the collection controlling some aspect of the agent and working in conjunction with all the other behaviors. A concrete example of such is the can collecting robot built by Jonathan Connell[5]. This example makes extensive use of Brooks' subsumption architecture approach[3].

Such collections of reactive behaviors can be defined as combined, synchronous finite state automata, using real time programming languages which have strong formal components. These language tools, such as COSPAN[8] and ESTEREL[1], require sophisticated users who have deep knowledge of both the syntax and semantics of the language.

The goal of this work is to make use of the simplicity of graphical FSA editing to specify concurrent, synchronous FSAs, and from those to produce COSPAN descriptions of these behaviors for analysis, and C language programs to implement the designed behaviors. The usefulness and validity of this approach is tested through the design, verification and implementation of several examples, including a controller daemon for a robot arm.

2 Behavior Architectures

Many researchers have the goal of building a mechanical device which exhibits intelligent behavior. Not all of them agree on the meaning of "intelligent behavior," and naturally most have their own approach to the problem. The more traditional approach is to follow the path of artificial intelligence research, with high level planning[6]. This planning and reasoning is based on constructing a representation of the world and making inferences about what to do in a given situation in the environment.

Proponents of this approach argue that representation and symbolic manipulation are necessary for intelligent behavior[10]. Without some sort of symbolic approach and the use of abstract concepts, an autonomous agent might not be able to tell, for instance, that a red soda can and a green soda can share attributes and do not need to be treated as two totally distinct types of objects, as they differ only in the concept of color.

Problems with the use of symbolic representations can occur when the computing resources needed to do the reasoning are not quick enough to keep up with real world situations. Improper actions might be taken, or actions necessary to keep the autonomous agent intact and functioning might not be done in time.

3 Graphical Editing

Since early researchers first got computers to make pictures, using non-textual, visual means of specifying programs has been a goal[14], such as the Graphical Programming Language work at the University of Utah [12]. Human beings do well at shape recognition and pattern matching, skills that were developed for thousands of years before text processing came along. Graphical displays of state machines can convey the flow of control directly, in an easy to grasp manner, giving a direct transfer of meaning [15].

A major goal of this work is to provide a basis for rapid development of state machine behavior descriptions. The quickness with which one can grasp pictorial representations over simple text is a definite aid in quickly laying out a design. In recent years similar systems have been developed, for instance the PetriFSA language within the Visual Programmer's Workbench[16]. This program performs similar functions as GI Joe, but not include facilities for formal verification as does the COSPAN aspect of GI Joe. The Task Control Architecture of Simmons[17] has a similar goal, that of taking high level behavior descriptions and converting then to running code. This system, though, views behavior differently and is oriented towards single purpose use, and is not a more general development tool as GI Joe could be.

The combination of graphical representation of state machines, followed by formal verification and code generation is the basic premise of the GI Joe work. It is hoped such a system will provide a useful tool to robotics and reactive behavior research.

4 COSPAN Programming Model

COSPAN is a coordination specification and analysis tool originally developed by R. P. Kurshan to assist in the development of large, control intensive programs. The methodology consists of formal verification, complexity management and formal refinement[11]. The GI Joe design process builds on this idea, and employs COSPAN as the means of formal verification of state machine designs.

Analysis of an FSA in the COSPAN model is a three part process. The FSA is described by the programmer with a textual description. This text is in the form of a *.sr* file, which is then run through the first part of the process producing a program in the C language. The next phase compiles this code into an executable file, and the final phase runs the program, monitoring the progress of the FSA and looking for inconsistencies, and checking for task acceptance defined by the user.

The COSPAN S/R example in Figure 1 shows a simple flip-flop state machine. If in the ON state and the selection variable takes the value off, it moves to OFF. When in the OFF state and the select variable has the value of on, it moves to ON. The textual description is certainly not overly complicated, and the syntax can be understood with a minimum amount of trouble. The exact same two states and transitions can be described graphically, as shown in figure 2, with a few mouse clicks and the keystrokes necessary to label the states and transition arcs, which can be done in less than a minute.

The goal of the first phase of this work was to replace the time consuming process of creating the textual descriptions of an FSA with a simple to use graphical interface. The FSA editor allows for creating states and transitions, and grouping them into named modules. Often state information from one module is needed to effect a transition in another module. Such a condition is easy to describe using simple labeling schemes. The FSA editor will then compile the descriptions into S/R language acceptable by COSPAN. Compilation of the C code and the execution analysis will then proceed as in the standard COSPAN process.

```
proc ALT /* Alternator */

        selvar  #:(off,on)
        stvar   $: (OFF,ON)
        init    OFF

      trans

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

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

Figure 1: COSPAN S/R Example

5 Verification

COSPAN works by checking that the "language" accepted by the FSA described is a subset of the language which the user defined task describes [7]. In this context a language is a sequence of state transitions rather than a string of characters. The user can test the behavior of the design by checking to see if some transition sequence is accepted by the FSA. If the transition is a desired one, than one would want the FSA to accept it, indicated by a "Task Performed" result from COSPAN. If the transition is unwanted, such as having a vending machine dispense a product before payment is collected, then a result of "Task Failed" from COSPAN would be the result wanted.

The simplest method of verification is done by having the user defined acceptance task be a simple two state automata. There is one initial state and one acceptance state. The transition from the initial to the acceptance state occurs if the predicate under test occurs, otherwise the machine remains in the initial state.

In the case of a vending machine example, having goods delivered without proper payment is undesired behavior. That is, going to a "Deliver" state without a "Paid" transition is an undesired behavior. The predicate to test for this would be translated into a task definition such as that in Figure 3. And since the task tests for an undesired behavior, running it through the COSPAN process would yield a result of "Task Failed", which means the FSA as described will not perform this unwanted behavior.

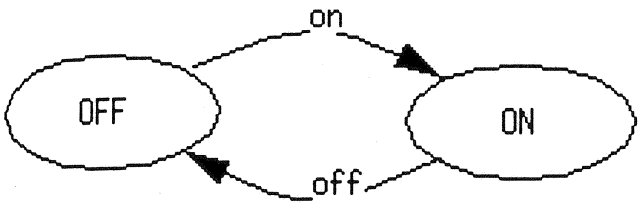

Figure 2: Alternator Graph

```
monitor TASK

stvar $ : (OK,NOK)
init OK
trans

true -> NOK : ( Vend.$ = Deliver) * (Vend.# != Paid)
-> OK : else;
end
```

Figure 3: COSPAN Task Example

6 A Robotics Application: Control of the Rhino

The entire design process was applied to the following problem: the goal was to design and implement a command interface to a Rhino robot arm with limited sensors and actuators. As the Rhino is used by students in various robotics classes, a good robust interface to the device will be of practical value.

The design and verification process is similar to the earlier examples, with a behavior module for the arm built up from simpler modules describing the actions of the individual parts of the system.

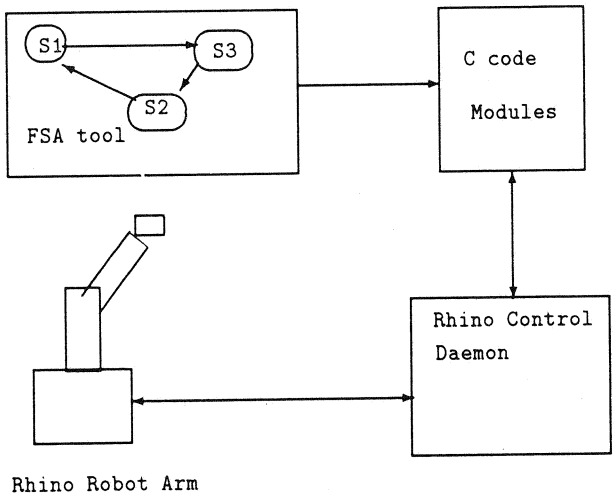

Figure 4: Rhino Model

6.1 Design Considerations

The Rhino is an arm with six degrees of movement, each with its own actuator and, where applicable, limit sensor. Commands are sent to the arm and status returned over a single serial communication line. This situation could lead to difficulties if a behavior module were composed of simple controllers for each of the six actuator and sensor pairs, and each of these assumed there was no contention for the communication path. Status returned by one controller's query could be intercepted by another, leading to undesirable behavior.

To counter this possibility the top level Rhino behavior module will not communicate directly with the arm, but instead with an intermediate controlling process (See Figure 4). This controlling daemon will run in the background. The Rhino must accept input from a top level user program and pass it on as to the appropriate motors, as well as coordinate motor position queries and make sure that no commands which could lead to damage of the robot are executed.

6.2 Main Controller

The basic premise of the Rhino controller is straightforward. It needs to accept user input, and send valid commands to the Rhino. If the command is a motor move, it passes the command to the motor controller. If the command is a query, then the query subsystem will take care of it. This top level view is shown in Figure 5.

The Rhino controller daemon, after setting up communication ports with the Rhino device and user interface sockets, sits in the *Idle* state. When a command comes in from the user program, the controller moves to the *Act* state. The command is parsed and acted upon according to the type. If a query, the controller goes to the *Query* state, if a motor control command, the *Motor* state, or if the command is not recognized, it returns to the *Idle* state. The three states *Act*, *Query* and *Motor* represent state machines implemented from designs done with GI Joe.

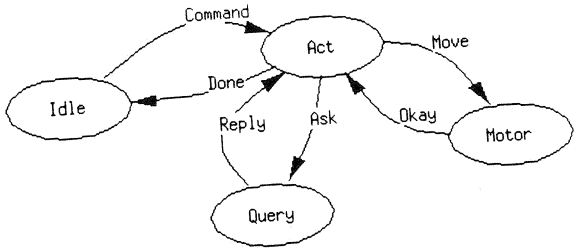

Figure 5: Rhino Top Level

6.2.1 Act

Once a command has been received by the Rhino controller, it must be acted upon. If it is a recognized command the appropriate action must be taken, if it is not then the controller simply returns to the *Idle* state.

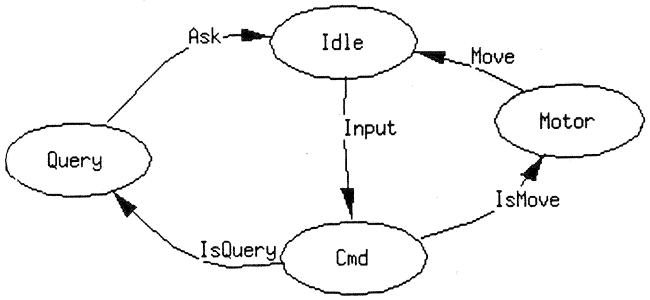

Figure 6: Act State Transitions

Commands acceptable to the Rhino are of four types. The first is the Start command, which consists of a two to five character string. The first character is the motor identifier, "A" through "H". The rest of the command is a string of 1, 2 or 3 digits, with the digits preceded by an optional "-" for reverse motion or a "+" for forward motion. If there is no sign character, "+" is assumed.

The second command is the Question command, which queries the motor's position. This command is a two character string with the first character being the motor identifier and the second character is a "?". When this command is received the Rhino responds with a single character encoding the motor's distance from the zero position.

The third command is the Information command, consisting of the single character "I". The Rhino returns a single byte which consists of 01xx xxxx where the "x" corresponds to a specific motor. If the limit switch for that motor is open, the value will be a 1, if the switch is closed the value will be 0.

The last command is Stop. This is a two character command with the motor identifier followed by "X". This command shuts off power to a motor and sets it error code to zero. This is useful to stop a stalled motor from burning itself up. All the commands are terminated with a carriage return.

With only these few simple commands available, it is trivial to determine whether or not a command is legal, according to the following:

command	→	start | stop | query | info
start	→	[A..H]+—-[0...127]
stop	→	[A..H]X
query	→	[A..H]?
info	→	I

The start and stop commands elicit no return response from the Rhino over the serial line, and can be forwarded to the motor controller. The query and info commands both return values, which require different handling, which will be done in the Query state.

6.2.2 Motor

Each of the up to 8 motors on the Rhino can be controlled separately. Motor movement is done by sending a start or stop command to the Rhino as described above. The design of the Rhino card limits the size of each individual move to 127 steps or less, even though a particular motor may have a limit from 300 to 4,700 steps. So to move a motor for 500 steps, say, it is necessary to move it in 5 steps of 100 each. Rather than having the user program do incremental motor movements in this fashion, the controller will provide this movement partitioning.

Any large movement broken down into several steps may appear as a single smooth action, since the Rhino will accept another command for a motor before it has stopped moving. Getting a smooth, continuous movement is done by checking the position of the motor, and sending another incremental move command while running. This send and check process is repeated until the entire motion requested has been completed.

The position checking portion of this is where the possibility of conflict may arise. If a user sends a query command for one motor, and the motor controller sends a query command to a second motor at about the same time, it would be possible for the results to be mixed up without careful control. So if there is a user query pending, the motor controller must wait for it to complete, possibly stopping the motor while waiting, if necessary.

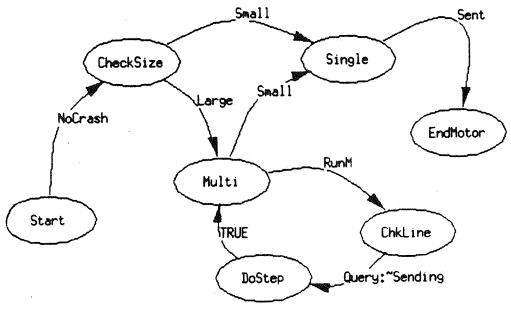

Figure 7: Motor Control States

To keep track of the position of all the Rhino movements, in order to return to a known state or keep from damaging the device, all movements will be recorded. If a command is given which would move a particular motor past its limit, the controller will only move the motor up to the limit, and will only act upon commands for that motor which move it away from the limit.

6.2.3 Query

Reading back status from a motor provides more difficulty than sending motor movement information. The same facility that allows one to send commands to a moving motor also allows reading the status or location of the motor while it is running. And unless this is taken into account, position readings may be inaccurate. It will be assumed that if the programmer requests status information they are aware of this. If the motor status is needed for the daemon this is taken into account.

The sequential nature of the query subsystem can be modeled by a simple two state machine, as shown in Figure 8. It sits idle in the "QIdle" state, until a query request is received. At that time, it moves to the "Sending" state, where it waits for the reply before returning to the idle state.

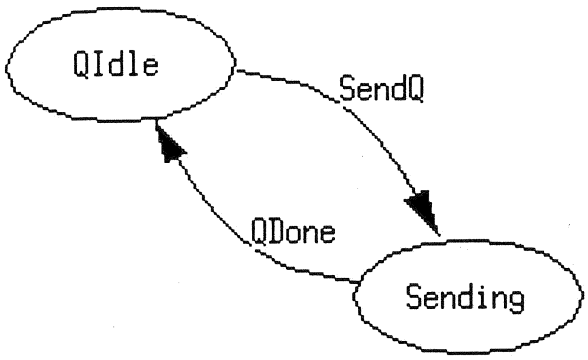

Figure 8: Query Control States

6.3 Task Acceptance

The goal of the Rhino program is to move the Rhino the amount and in the direction desired by the user, without moving too far and causing damage. Also, status returned from a motor must be matched to the proper query to avoid confusion. The first property can be checked by writing a task which will ask if it is possible to move a motor too far. Given the top level motor subsystem of Figure 7 it is easy to see that if the Boolean "NoCrash" is false, the motor will not respond to a move command from the top level controller.

Rephrasing this point in a COSPAN task definition is not that difficult. Basically, we ask if the motor FSA will move to the "CheckMove" state with "NoCrash" being false. In COSPAN syntax, we can write the expression:

(Motor.# = NoCrash) * (Motor.$ = CheckMove)

A complete task definition uses this phrase as the test for failure. That is, the task will proceed, and remain in an acceptable state as long as the motor process state doesn't become "CheckMove" without the corresponding selection variable "NoCrash" being true. To check for compliance, the following COSPAN code is added to the description of the motor FSA:

```
monitor TASK

import  Query, Motor
stvar $ : (OK,CRASH)

cyset { OK }
init OK
trans
true -> CRASH :  ~( Motor.# = NoCrash)
          * ( Motor.$ = CheckMove )
-> OK : else;
end
```

Running the FSA description through the COSPAN process yields "Task Performed!", meaning that our task has stayed within the defined acceptable states. In this case, all states except going from "Start" to "Checkmove" without the "NoCrash" predicate being true.

The second property is similar to the serialization of the Votrax controller discussed earlier. The COSPAN task asks if a second query will be processed before the first query has been acknowledged. If this is not possible, then queries and their responses will be correctly matched.

Examination of Figure 7 shows that the motor control subsystem communicates with the query subsystem when the motor is moving a large amount, and has to repeatedly check progress before sending another incremental move command. Checking the progress consists of sending a query command to the Rhino, which is a string of the form "D?", where the first character is the motor ID to be questioned.

If another subsystem has sent a query to the Rhino and is awaiting a reply, we must be sure that before the second query is sent the first one is answered. The motor FSA checks the state of the query FSA, and if it's busy, waits until it is free. A COSPAN task definition to check for this behavior can be constructed by saying that all motor FSA transitions are OK, except for moving to the "DoMove" state when the query FSA is in the "Sending" state. Casting this question in the COSPAN syntax yields the following:

```
monitor TASK

import  Query, Motor
stvar $ : (OK,Conflict)

cyset { OK }
init OK
trans

true -> Conflict : ( Query.$ = Sending )
* ( Motor.$ = DoMove )
-> OK : else;
end
```

The task has two states, "OK" and "Conflict". We want it to stay in the "OK" state, signified by the state "OK" being the only member of the cycle set. As long as the task stays within the elements of the cycle set during COSPAN analysis, the task will be performed. The following output of the analysis shows that the task as defined did indeed perform as specified.

```
cospan: Version 6.5.14 (AT&T-BL) 8/4/91
+ sr -DQTEST mq.fsa.sr -o mq.fsa.c
mq.fsa.sr: Sun Dec  8 20:52:36 1991
1.h: Sun Nov 17 16:49:16 1991
+ cc -o mq.fsa.an  mq.fsa.c ../cospan/lib/libsr.a -lm
+ ./mq.fsa.an
```

```
./mq.fsa.an: Synchronous model
./mq.fsa.an: Initialization complete.
1 initial state.
./mq.fsa.an: Search complete.
10 states reached.
10 states searched.
3 DFS trees generated.
4 SCC checks done.
12 edges transversed:
        4 plus, 7 tree, 0 self, 0 forward,
        1 back, 0 cross-intra, 0 cross-inter.
14 resolutions made.
1+1 boundary frames allocated.
./mq.fsa.an: Task performed!
```

6.4 Implementation Details

The Rhino daemon is written using the interprocess communication (IPC) facility of UNIX. This daemon has access to the communication path with the arm itself, and services requests from the behavior components. Adding this arbitration stage allows the behavior module to assume that the communication among components and the Rhino is going to work, with messages getting delivered reliably and in proper sequence.

6.4.1 C Code Production

GI Joe at this stage provides only stubs and basic flow of control code. Implementation of the FSA as working C code requires these stubs to be defined fully. This step must be done by the designer outside of GI Joe. The **Compile** command in GI Joe takes an FSA that had an initial state *Init* and a transition to the next state *Idle* when *Ready* (See Figure 7) becomes true and produces this code:

```
switch (CURRENT) {

case _Init:

    if (Ready())
    CURRENT = _Idle;
    return;
  }

case _Idle:

    if (Command())
    CURRENT = _Run;
    return;
  }
```

It is readily apparent that the above code won't really do anything useful unless the designer writes definitions for the Boolean functions *Ready()* and *Command()*. The corresponding SR description for this fragment looks similar:

```
    init Init
    trans

Init        { Ready }
     -> Idle :  # = Ready
     ->$     : else;

Idle        { Command }
     -> Run  :  # = Command
     ->$     : else;
```

7 Summary and Future Work

Rapid design, verification and implementation of behaviors which can be described by finite state machines is a worthwhile goal. GI Joe and related tools can help in laying out and understanding the flow of control in state machines. With the proliferation of workstations supporting standardized graphical user interfaces such as X, programmers are expecting environments which make use of this pixel power.

Verification of the resulting FSA using COSPAN is a trickier problem. While it is easy to put circles and arrows on the screen to represent the FSA, the concept of "correctness" is difficult to describe in pictures. This area certainly warrants further investigation.

While the initial stages of FSA design are made simpler by the graphical editing, code generation is still time consuming. GI Joe currently does produce enough code to guide the actual implementation, but there is the possibility of having the programmer stray from specification as the lower levels of routines are filled in. If the states and transitions are detailed enough, though, the extra code required as fill in should not be that extensive. Still, following the specification requires discipline on the part of the programmer.

At this point only initial development and testing has been done on GI Joe, with no use by others for design work. The concepts appear reasonable, and the graphics libraries allow good support for putting ideas into the programs, but the general applicability of GI Joe to reactive behavior design remains to be seen. As it seems with any programming system, there is always much more to do. GI Joe could benefit from further work on the user interface, and some useful command additions. One useful addition would be the ability to "stack" the display, like a deck of cards. Suppose one had a state shown in the GI Joe drawing window as a single state, when it really represents an entirely separate FSA in itself. Clicking on that state could bring the FSA it represents to the foreground for editing. This would aid in creating and easily navigating hierarchical designs.

In the current model, the user designs behaviors in GI Joe, then must run COSPAN from a separate shell. Having the FSA editor run COSPAN automatically would be an improvement. Also, COSPAN produces textual output that can be rather daunting for the user. The states and transitions of the FSA are printed during the analysis. In addition to having GI Joe call COSPAN on the design under test, having it also parse this output and relate it to the on screen display could be most useful in tracking down errors. This verbose output trace could form the basis of an on-screen automation of the behavior.

Acknowledgements

This work was supported in part by DARPA (N00014-91-J-4123) and NSF grant CDA-9009026. All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Geŕard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. INRIA Research Report 327, INRIA, Sophia Antipolis, France, September 1984.

[2] M. Bradakis. Reactive behavior design tools. Master's thesis, University of Utah, Salt Lake City, Utah, June 1992.

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

[4] R.A. Brooks. Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence.* MIT Press, 1987.

[5] Jonathon H. Connell. *Minimalist Mobile Robots.* Academic Press, Boston, Massachusetts, 1990.

[6] Arun Guha and Martin Dudziek. Knowledge based controllers for autonomous systems. In *Proceedings of the IEEE Intelligent Control Workshop*, pages 134–138, Troy, New York, August 1985.

[7] Z. Har'El and R.P. Kurshan. Automatic verification of coordinating systems. In *Workshop on Automatic Verification Methods for Finit State Systems*, pages 1–16, 1989.

[8] Z. Har'El and R.P. Kurshan. Software for analytical development of communication protocols. *ATT Technical Journal*, page to appear, January 1990.

[9] Thomas C. Henderson and Rod Grupen. Logical behaviors. *Journal of Robotic Systems*, 7(3):309–336, 1990.

[10] David Kirsh. Today the earwig, tomorrow man? *Artificial Intelligence*, 47:161–184, 1991.

[11] R.P. Kurshan. Analysis of discrete event coordination. In *Discrete Event Systems: Models and Applications*, LNCIS, page to appear, Berlin, 1990. Springer-Verlag.

[12] G. Q. Maguire. *A Graphical Workstation and Programming Environment for Data-Driven Computation.* PhD thesis, Univserity of Utah, Salt Lake City, Utah, March 1983.

[13] Nils J. Nilsson. Shakey the robot. Technical Report Technical Note 323, SRI International, April 1984.

[14] Georg Raeder. A survey of current graphical programming techniques. *IEEE Computer*, pages 11–25, August 1985.

[15] Shane Robison. Data structures and system support for a graphical programming language editing and program development environment. Master's thesis, University of Utah, Salt Lake City, Utah, June 1983.

[16] James Waller II Rubin, R.V. and E.J. Golin. Early experience with the visual programmer's workbench. *IEEE Transactions on Software Engineering*, 16(10):1107–1121, 1990.

[17] Reid Simmons. Coordinating planning, perception and action for mobile robots. In *AAAI Symposium on Integrated Intelligent Architectures*, pages 146–150, March 1991.