March 1994

How Animated Agents Perform Tasks: Connecting Planning and Manipulation Through Object-Specific Reasoning

Libby Levison
University of Pennsylvania

Norman I. Badler
University of Pennsylvania, badler@seas.upenn.edu

Presented at *Toward Physical Interaction and Manipulation, AAAI Spring Symposium Series, 1994.*

How Animated Agents Perform Tasks: Connecting Planning and Manipulation Through Object-Specific Reasoning

Abstract
Creating animations of a human figure performing a task requires that the agent interact with objects in the environment in a realistic way. Agent-object interaction is not completely specified from a task description alone. In this paper we sketch an architecture for the Object-Specific Reasoner (OSR), an intermediate planning module which tailors high-level plans to the specifics of the agent and objects. As plans are elaborated, the OSR generates a sequence of motion directives which are ultimately executed by a simulator. Descriptions of failures can be used to identify possible tools for the agent to use. An Object-Specific Reasoner is necessary in a system which allows an agent, equipped with a set of action behaviors, to interact in a semiautonomous fashion with the world.

Keywords
object manipulation, motion planning, animation, tools

Comments
Presented at *Toward Physical Interaction and Manipulation, AAAI Spring Symposium Series, 1994*.

How Animated Agents Perform Tasks: Connecting Planning and Manipulation Through Object-Specific Reasoning

Libby Levison & Norman I. Badler
Center for Human Modeling and Simulation
Department of Computer and Information Science
University of Pennsylvania,
Philadelphia, PA 19104-6389
{libby@linc & badler@central}.cis.upenn.edu

Keywords: Object Manipulation, Motion Planning, Animation, Tools

Abstract

Creating animations of a human figure performing a task requires that the agent interact with objects in the environment in a realistic way. Agent–object interaction is not completely specified from a task description alone. In this paper we sketch an architecture for the Object-Specific Reasoner (OSR), an intermediate planning module which tailors high-level plans to the specifics of the agent and objects. As plans are elaborated, the OSR generates a sequence of motion directives which are ultimately executed by a simulator. Descriptions of failures can be used to identify possible tools for the agent to use. An Object-Specific Reasoner is necessary in a system which allows an agent, equipped with a set of action behaviors, to interact in a semi-autonomous fashion with the world.

1 Introduction

Suppose you had a computer system which could animate a human figure by making it reach for things, grasp them, pick them up, or walk along a path in

a semi-autonomous fashion.[1] Now suppose that you want to generate realistic simulations of the character performing different tasks. Suppose further that you have a task-level interface to this agent, so that you can give commands such as `(pickup jack glass)` or `(open betty door)`.[2] These high-level descriptions of action must somehow be mapped to the agent's control language: the command `(pickup jack glass)` must be converted to fully-parameterized motor commands which specify details such as where on the glass to grasp, how high to lift the glass and where to stand in relation to the glass.

Two central observations motivate this work: 1) different expansions of a task arise for different agents and objects, and 2) different expansions of a task arise for different intentions. At the same time, controlling animated figures requires specifying details of the desired motions which are rarely present in high-level commands. Our research is concerned with building an intermediate reasoning system which maps a small set of high-level commands to the language of the animation system. We are investigating whether this mapping can be done by considering action commands in terms of each agent's resources, the object attributes, the intention, and the situation. The entire process of elaborating high-level commands can be seen as grounding symbolic action descriptions in 'numeric' motion descriptions.

We are building the Object Specific Reasoner (OSR), which is implemented in the SodaJack system, which itself is part of the **AnimNL** Project (**Anim**ation from **N**atural **L**anguage Instructions) at the University of Pennsylvania [BPW93, WBD+93]. The goal of the AnimNL project is to generate realistic animations of human figures carrying out tasks specified through natural-language instructions. The SodaJack system covers the plan expansion and simulation planning portions of the AnimNL project.

1.1 Bridging Planning and Action

High-level planning is performed by a module 'above' the OSR. The high-level planner is responsible for recognizing the task, and selecting from a library a plan to accomplish that task. This planner breaks the plan down into steps – *task-actions* – for the agent to perform. (While the high-level planner might consider

[1] By 'semi-autonomous' we mean to imply that instructions are given to the agent, who attempts the task, and reports back to the instructor with the result of the action. If the action failed it is the role of the instructor to select another course of action.

[2] The first term indicates the action to take; the second term is the agent to do the action; the third term refers to the object of the action. Assume that object names are uniquely identified with an object in the animation scene.

these task-actions basic or primitive, they are not necessarily primitive in terms of the motions which must be performed.)

Below the OSR, a simulator manages motor-control issues for agents and objects. This reactive [KR90, Bro86] module provides situational knowledge to the OSR (and the high-level planner) and relieves the OSR and the high-level planner of details of motion planning and motor control. The reactive nature allows many motion control decisions to be made internally, such as how to go around obstacles in the reach or locomotion path.

The simulator is invoked with detailed motion commands or *motion directives*. The high-level planner might use a step like `(grasp jack glass)`; the motion control system, however, requires the motion directive `grasp` to indicate which hand to use, what type of grip to use, the approach vector for the hand, and the region of the object to grasp.[3]

The problem is this dichotomy between task-actions and motion directives. The high-level planner does not know enough about the physical description of an object to determine the grip-site; neither does the simulator know enough about the intention behind the action to select the grip-site. It is the role of the OSR to break task-actions into sets of fully-specified motion directives. To generate realistic behavior of the simulated agent manipulating objects, the agent's physical motions must be fully described.

1.2 Implementation of SodaJack and the OSR

A prototype of the OSR exists in Lucid Common LISP. The OSR takes a task-action, with the object references fully resolved, and generates a set of motion directives to send to the simulator. The modules sketched above are implemented by: the Intentional Planning System (ItPlanS) [Gei92] decomposes high-level plans into task-actions, and object reference resolution and object locating is performed by Search Plans [Moo93]. Finally, agent and object motion simulation, and perceptual requests about the current state of the world will be handled through the Behavioral Simulator [BB93]. The Behavioral Simulator is, in essence, an operating-system, as it manages both agent and control resources, thus allowing for simultaneous motions. The language is similar to McDermott's RPL [McD90], with an emphasis on managing a behavioral mechanism [Bec94].

ItPlanS, Search Plans and the OSR are integrated into the **SodaJack** system [GLM94]. Built on the *Jack*™ software platform [BPW93], SodaJack is named

[3] We use 'hand' through out this discussion, as the animated figure we work with has two hands.

† *Jack* is a registered trademark of the University of Pennsylvania.

after its first domain, a soda fountain in an ice cream shop. The animated agent takes orders, and manipulates objects such as bowls, glasses, ice cream scoops, and refrigerator doors. SODAJACK is not currently fully integrated with the Behavioral Simulator; the OSR writes out a set of motion directives which are run through the Behavioral Simulator separately.

SODAJACK is invoked with a command to ItPlanS: `serve soda`. ItPlanS retrieves from its plan library a plan to serve sodas, and expands the plan to task-actions. At the same time, ItPlanS invokes Search Plans to find a soda in the animation scene. ItPlanS sends each completely specified task-action, one at a time, to the OSR.

The OSR converts task-actions to motion directives which are in turn passed to the Behavioral Simulator. In addition to animating the human figure and all of the objects in the scene in a time-sliced manner, the Behavioral Simulator provides "sensory" feedback to ItPlanS, Search Plans and the OSR (e.g., the current location of the agent or an object); this knowledge is used for incremental decision-making and plan specification.

Within the SODAJACK system, the OSR distinguishes between checking the *feasibility* of the task-action and invoking the Behavioral Simulator. ItPlanS invokes the OSR with a task-action; if the OSR finds that a task-action is feasible, it returns this analysis to ItPlanS (the basis for this decision is discussed below.) ItPlanS then uses this knowledge in deciding which plan expansion to pursue. When ItPlanS elects a task-action to perform, the OSR generates the motions directives and invokes the Behavioral Simulator. If the Behavioral Simulator cannot successfully perform a motion, the OSR is alerted to the failure, and relays the error message to the high-level planner. The OSR does not do any replanning at the task level.

2 OSR Architecture

2.1 Assumptions

The previous sketch of how the OSR functions and the other modules it relies on serves as an introduction to some of the assumptions we make in our work.

The OSR is not a stand-alone module. It does not handle natural language input, does not resolve object reference, does not do plan decomposition, nor does it deal with the agent's motor control system. The OSR is defined as a necessary, intermediate reasoning process in a complex system.

We assume that all object references are already identified in the world. Lo-

cating an object gives us access to mechanical and functional knowledge about the object. (Object knowledge is discussed further below.)

We also assume that the agent is endowed with a certain number of skills or behaviors which it can perform. For example, we assume that the agent can locomote between two points (including planning the path), can place its end-effector on the surface of an object (without causing illegal collisions), and can take control of an object, (e.g., grasp the object, or bring an end-effector into contact with the object). How an agent's physical behaviors are performed and interleaved is handled by the underlying simulator.

2.2 Terminology

TASK-ACTION: A task-action is one of the steps in the high-level task plan sent to the OSR. Currently the list includes: `goto, move, grasp, release, look, co-locate`[4], `pickup, carry, open` and `close`.

A task-action is of the form: `((pickup jack soda) serve)`. The first term is the requested task-action itself, here `pickup`; the second term is the agent to perform the task-action. After this, each task-action requires different parameters: most task-actions require an `object` (but `goto` requires a destination and not an object). In the above example, `soda` is the object of the task-action. Other optional arguments are `destination, hand`, and `direction`. The final term in any task-action is the intention governing the action.

INTENTION: We take *intention* to be the goal that the agent is committed to achieve. Each task-action from the high-level planner specifies the intention governing the task-action. At the present time, the intention is usually the next task-action to be performed: the performance of a motion is influenced by knowledge of the subsequent motion. Thus, the set of intentions is the set of task-actions, plus two additional intentions: `use-function`: which captures the notion of manipulating an object with the intention of using it for its inherent function; and `poise-for-action`: which commands the agent to assume a neutral stance until the next task-action is known.

OBJECT KNOWLEDGE: The OSR distinguishes between two kinds of object knowledge. The first is TYPE, or category, knowledge of an object (it is a TOOL, its function is to contain things, what its part/subpart structure is). The second is TOKEN, or instance, knowledge which includes particular details of the object

[4] `Co-locate`: cause two objects to be at the same location.

instance being manipulated (it is 8 inches wide, ferrous, and red). This knowledge is stored in two knowledge bases: TYPE information is stored in a symbolic knowledge base; TOKEN knowledge is stored in the graphics database.

ACTION OUTLINE: The action outline is an underspecified description of the motions to be performed; they are scripts [SA77], but at a lower, behavior level. The task-action and the type of the object is used to select the action outline: for example, the task-action `move` has different definitions for objects that are containers and those that are tools (the former requires the additional constraint that the object should not be turned on its side when it is moved; the latter has a preferred motion path).

An action outline is a list of steps which are either other action outlines or OSR-motions (see below). Because every action outline is type sensitive, when an action outline is expanded, a step which is defined by another action outline automatically selects the correct version or set of motions based on the type of its object.

OSR-MOTION: The OSR uses an intermediate language to describe the required motions. OSR-motions are agent-independent, allowing the OSR to reason abstractly about the required motion before substituting a particular agent's behaviors. Examples of OSR-motions are `locomote` and `get-control-of`: these are general concepts which are performed differently by each agent to achieve the same goal.

BEHAVIOR: Each agent has a set of behaviors which it can perform; for each OSR-motion, each agent has an associated behavior. Behaviors are defined as a sequence of motion directives in the simulator. For example, the OSR-motion `get-control-of` can be achieved in different ways by different agents: one agent might have a `grasp` behavior, while another agent might have a complex behavior requiring pushing an object into a corner to get control of it.

MOTION DIRECTIVE: A motion directive is a fully specified call to the simulator. Currently the list of motions includes[5]: `arm-motion`, `com-motion`[6], `look-motion`, `foot-motion`, `torso-motion`, `figure-motion` (moves an inanimate object, e.g., a box sliding), `grasp-motion`, and `release-motion`.

[5] These motion directives are defined by, and are the input language of, the Behavioral Simulator. As the Simulator's language expands, so too can the language of the OSR.

[6] "com": center of mass.

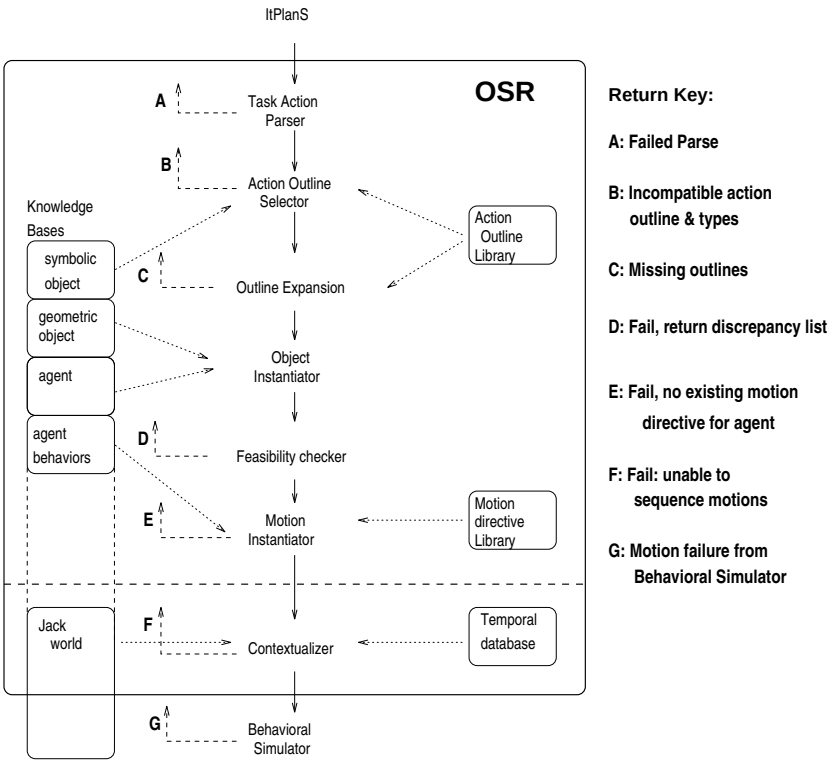

Figure 1: Detailed OSR system diagram, within SodaJack.

An example motion directive is: `(arm-motion jack start end arm? site)`. The first term is the motion directive to be performed, the second term identifies the agent. The start time and end time are next, followed by a constant indicating which of the agent's arms to move. Finally, the destination is given as a site (coordinate system) in space. Each motion directive requires slightly different parameters. These are defined by the Behavioral Simulator.

2.3 The OSR System

The following two subsections explain the current architecture (see Figure 1).

2.3.1 Checking Action Feasibility

Checking the feasibility of a task-action is performed in five stages: selecting an action outline, conditionally expanding all steps of the outline, using details of

the object of the current task-action to refine the outline, verifying that the agent can perform the specific action on this specific object, and supplying the specific agent's behaviors for each motion in the task. If an outline can be found and tailored to the details of the agent and the object, the task-action is judged to be *feasible*.

(In a preprocessing stage (Task-action Parser), the input from the high-level planner is parsed and converted to a representation which the OSR can use.)

In the first stage (Action Outline Selector), the task-action and the type of the object are used to select an action outline from a library of outlines. This library is indexed by both the task-action and a taxonomy of object types. For each task-action, there may be separate action outlines for each type of object that can appear as an argument. This step is implemented as a lookup table.

The second stage (Outline Expansion) expands all the steps in the action outline. Each step is either another action outline or a motion directive. Each step is sensitive to the type of its object: expanding an action outline implies interpreting each of its steps in terms of the object. The process continues until sets of OSR-motions have been substituted for all action outlines. Action outlines may specify temporal ordering constraints on the motions [All84].

The third stage (Object Instantiator) binds parameters of the OSR-motions based on information about the specific object instance. This is also the stage at which OSR-motion specific parameters (such as grasp-site, grip-type or approach-vector) must be supplied. Currently a knowledge base is used to determine – from the OSR-motion, the object TYPE and the intention of the task-action – these additional parameters. This is similar to approaches taken by [IJLZ88, TBK87, RG91].

The fourth stage (Feasibility Checker) involves checking dependencies between the agent resources and object attributes. Each OSR-motion includes a predicate which specifies those pairs of resources and attributes to check. For example, the OSR might check whether the agent's hand is large enough to grip the handle of a scoop. If all the dependencies for all the motions in the current outline are within tolerance, the OSR reports that the task-action is feasible.

If the agent and object attributes fail the tolerance test, then control is returned to ItPlanS along with a *discrepancy list* of those resource/attribute pairs that are out of tolerance. How this list can be used is discussed in the next section.

The fifth stage (Motion Instantiator) supplies the agent's behavior for each OSR-motion. At this point the task-action has been completely converted to a set of motion directives.

2.3.2 Action Execution

When called upon to output motion directives to *Jack*, the OSR must first provide a start time and an approximate duration for each motion (Contextualizer). Durations are calculated from a temporal database which contains both rules to generate times for parameterized motions (such as a `reach`) [EBJ89] and fixed values for other motions. When the temporal information is added, task-action refinement is finished, and the motion directives can be sent to *Jack* for animation. Errors (in the form of failed actions) may occur during animation; the OSR aborts the remaining motions in the task-action expansion and relays errors back to ItPlanS for replanning.

2.4 Summary

Using TYPE knowledge about the object, the OSR first builds action outlines for the task-action; it then refines these partial plans with TOKEN knowledge. Adding agent-specific knowledge, and selecting agent-specific behaviors, allows the OSR to tailor a general task-action to the specific context and check its feasibility.

The number of possible variations of task-actions that an agent might be asked to perform makes a strictly case-based approach (such as [Ham86], which enumerates all possible cases) impractical. The multi-stage OSR provides a robust, mid-level planner which adapts task-actions to the agent, object, intention and situation.

3 Action Failure May Not Be Harmful or, Understanding Tools and their Usage

A *mediated* task-action is a task-action which requires the agent to employ a tool. The term *mediated* comes from the fact that the tool mediates between the agent and the object [WBD+93]. Mediated task-actions include `med-open`, `med-grasp` and `med-release`.

3.1 Feasibility Failure

An agent is unable to manipulate an object when the agent lacks some resource (e.g., is not strong enough) or when some set of the object's attributes rule out the action (e.g., a pot is too hot to be grasped, the object does not move). Task-actions are not feasible, when the agent resources and the object attributes are

not in tolerance. The OSR places any resource/attribute pairs which are out of tolerance on a *discrepancy list*, which is returned to the high-level planner.

Tools augment the agent's resources, or filter the object attributes adversely affecting the OSR-motion. In the OSR, tools are regarded as sets of attribute modifiers which can be used to make the agent resources and the object attributes compatible. An object whose attributes are described by the discrepancy list might serve to make the motion feasible. Locating and using such an object – such a tool – might license the task-action.

3.2 Determining Tool Types

We present three possible extensions to the OSR algorithm. First, as suggested above, when there are discrepancies between resources and attributes, the discrepancy list can be returned to the high-level planner as a way of describing the failure. In this case, it is possible that the discrepancies can be used to select a tool that can be used in performing the task.

The next two extensions are variations with mediated task-actions, and are not presently implemented. In the first case, a mediated task-action is requested, and a tool is specified, e.g. `((med-open jack bottle bottle-opener) serve)`. Here, the tool's attributes are added to the agent resources before the resources are checked against the attributes by the dependency checker. If the tool has the correct attributes, then the incompatibilities are reduced to the point that the dependencies between agent and object are met when the motion is performed.

Third, a mediated task-action is requested, but no specific tool is specified, e.g. `((med-open betty bottle dummy-tool) serve)`. In this case, the discrepancies are discovered and used to describe the attributes of the required (dummy) tool. The difference between the first alternative and this, however, is in control: the high-level planner calls the OSR with a tool requested but unidentified – the OSR is invoked specifically to build the discrepancy list describing a possible tool. The description is returned to the high-level planner, which attempts to locate such a tool. If one exists, the high-level planner then has the option of requesting that the OSR check the feasibility of the task-action with the discovered tool. Should the high-level planner commit to the task-action, the OSR is called a third time to invoke the motion directives.

4 Task Designs

Below are sketches for how the OSR might handle three tasks. Recall that the high-level planner breaks tasks into sets of task-actions which the OSR handles one at a time.

4.1 Example 1: Get a soda

The high-level planner is given the goal `get soda`. It selects a plan and breaks that plan down into the following task-actions:
1. `((goto jack fridge) open)`. The intention on this task-action will restrict where in front of the fridge Jack will stop – knowing that he is about to open it, he will stop slightly to one side of the door.
2. `((open jack fridge) (get soda))`. The intention helps to determine how much the door of the fridge must be opened. The agent must both be able to 'see' inside to get the soda, and the door must be open wide enough that the soda can fit through the opening.
3. `((get-control-of jack soda) close)`. Jack must reach to the soda and grasp it. Note that, while not yet implemented, the intention to `close` the fridge should force the agent to first move the soda out of the fridge.
4. `((close jack fridge) poise-for-action)`. There is no other task-action in this plan expansion, so the agent must `poise-for-action` and wait for the next task-action.

4.2 Example 2: Tool Use

The OSR is given a mediated task-action which requires a specific tool: the task is to get a scoop of ice cream with the ice cream scoop.

The OSR is given the task-action `((med-get-control icecream-ball scoop) serve)`. The action outline is selected and expanded; the agent resources and the object attribute values are set after inquiring their values from the Behavioral Simulator and the knowledge bases.

Before dependencies are checked, the attributes from the scoop are also added to the dependency template. If the agent resources and object attributes are within tolerance for all OSR-motions, the OSR returns confirmation to the high-level planner that the task-action is feasible.

4.3 Example 3: Tool Selection

The high-level planner uses the OSR to build a discrepancy list which will describe a tool that might be used. The OSR is given a mediated action with the tool not specified: the task is to open the soda bottle with a tool.

The OSR is called with the task-action: `((med-open betty sodabottle dummy-tool) serve)`. The high-level planner recognizes that opening something of type `sodabottle` requires a mediated action, but does not know what tool is appropriate in this case. The high-level planner uses the OSR to build a description of a possible tool.

The OSR checks the dependencies between the agent resources and the object attributes. The discrepancy list indicates why the motion fails.

The high-level planner uses the discrepancy list to look for a tool, and finds a bottle opener which matches the required attributes. The high-level planner now re-invokes the OSR with the task-action: `((med-open betty sodabottle opener) serve)`.

At this point, the OSR must still ascertain that the action outline can be expanded to motion directives, and proceeds to do so. The OSR continues as in the first example and verifies that the opener can be used to `med-open` the `sodabottle`, and that `((med-open betty sodabottle opener) serve)` is feasible.

5 Conclusion

While the current prototype is built on top of an animation system, we believe that this work is of more general relevance. We have found few other systems developing general methods to build plans to manipulate objects: the issue is avoided either by selecting domains and tasks where manipulation is not crucial [VB90, LR90, McD93, Fir87, ZJ91], or by concentrating on general manipulation strategies (e.g., the "peg in hole" problem) when the task is generically specified ([IJLZ88, LPMT84]).

The OSR performs mid-level planning, tailoring high-level, "cognitive" plans to the specific agents and objects of the particular task-actions. Further, the OSR isolates the high-level planner from reasoning about object instance particulars, and is in turn isolated from details of motion directive execution by the simulator. This is necessary in a system which allows an agent, equipped with a set of behaviors, to act in a semi-autonomous fashion in his world.

By checking agent-object dependencies, the OSR can estimate whether a task-action is feasible for an agent. The resulting discrepancy list can be used to determine if there is a viable tool for the agent to employ. The strength of the proposed

OSR architecture is that it allows an agent, operating with partial knowledge and a symbolic action plan, to function in its world.

6 Acknowledgements

We would like to thank the members of the AnimNL project for discussion and reading earlier drafts of this paper. Thanks also to Bonnie Webber, Sandee Carberry, Mitch Marcus, Max Mintz, Tilman Becker and Beryl Hoffman.

This research is partially supported by ARO Grant DAAL03-89-C-0031 including participation by the U.S. Army Research Laboratory (Aberdeen), U.S. Air Force DEPTH contract through Hughes Missile Systems F33615-91-C-0001; DMSO through the University of Iowa; and NSF CISE Grant CDA88-22719.

References

[All84] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[BB93] Welton Becket and Norman I. Badler. Integrated behavioral agent architecture. Conference on Computer Generated Forces and Behavior Representation, 1993.

[Bec94] Welton Becket. The Jack LISP API. Technical Report MC-CIS-94-01, University of Pennsylvania, 1994.

[BPW93] Norman I. Badler, Cary B. Phillips, and Bonnie L. Webber. *Simulating Humans: Computer Graphics Animation and Control*. Oxford University Press, 1993.

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

[EBJ89] Jeffery Esakov, Norman I. Badler, and M. Jung. An investigation of language input and performance timing for task animation. In *Graphics Interface '89*, pages 86–93, San Mateo, CA, June 1989. Morgan-Kaufmann.

[Fir87] R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202–206, 1987.

[Gei92] Christopher W. Geib. Intentions in means-end planning. Technical Report MC-CIS-92-73, University of Pennsylvania, 1992.

[GLM94] Christopher Geib, Libby Levison, and Michael B. Moore. Sodajack: an architecture for agents that search for and manipulate objects, Submitted to AAAI, 1994.

[Ham86] Kristian Hammond. Chef. In *Proceedings of the* 5^{th} *National Conference on Artificial Intelligence*, 1986.

[IJLZ88] Thea Iberall, Joe Jackson, Liz Labbe, and Ralph Zampano. Knowledge-based prehension: Capturing human dexterity. In *IEEE Intl. Conf. on Robotics and Automation*, pages 82–87, 1988.

[KR90] Leslie Pack Kaelbling and Stanley J. Rosenschein. Action and planning in embedded agents. *Robotics and Autonomous Systems*, 6:35–48, 1990.

[LPMT84] Tomás Lozano-Pérez, Matthew. T. Mason, and Russell H. Taylor. Automatic synthesis of fine-motion strategies for robots. In Michael Brady and Richard Paul, editors, *Robotics Research*, pages 65–95. MIT Press, 1984.

[LR90] John Laird and Paul Rosenbloom. Integrating execution, planning and learning in SOAR for external environments. In *Proceedings of the* 8^{th} *National Conference on Artificial Intelligence*, pages 1022–1029, 1990.

[McD90] Drew McDermott. Planning reactive behavior: A progress report. In *ARPA workshop*, pages 450–4588, 1990.

[McD93] Drew McDermott. Transformational planning of reactive behavior. Technical Report RR-941, Yale University, 1993.

[Moo93] Michael B. Moore. Search plans. Technical Report MS-CIS-93-56, University of Pennsylvania, 1993.

[RG91] Hans Rijpkema and Michael Girard. Computer animation of knowledge-based human grasping. In *ACM: Computer Graphics*, pages 339–348, July 1991.

[SA77] Roger Schank and Robert Abelson. *Scripts, Plans, Goals and Understanding*. Lawrence Erlbaum, Hillsdale, N.J., 1977.

[TBK87] Rajko Tomovic, George A. Bekey, and Walter J. Karplus. A strategy for grasp synthesis with multi-fingered robot hands. In *IEEE Intl. Conf. on Robotics and Automation*, pages 83–89, 1987.

[VB90] Steven Vere and Timothy Bickmore. A basic agent. *Computational Intelligence*, 6:41–60, 1990.

[WBD+93] B. Webber, N. Badler, B. Di Eugenio, C. Geib, L. Levison, and M. Moore. Instructions, Intentions and Expectations. Technical Report MS-CIS-93-61, University of Pennsylvania, 1993. To appear 1994: *Artificial Intelligence Journal*, Special Issue on Computational Theories of Interaction and Agency.

[ZJ91] David Zeltzer and Michael B. Johnson. Motor Planning: an Architecture for Specifying and Controlling the Behavior of Virtual Actors. *Journal of Visualization and Computer Animation*, 2:74–80, 1991.