Title

Experiences with the Subsumption Architecture

Contact Information

John E. Arnold
Digital Equipment Corporation
295 Donald Lynch Boulevard DLB12-2/D4
Marlborough, MA 01752

Net:	Arnold@istg.dec.com
Phone:	508.490.8011

Topic

Principles.
Task-specific Reasoning, Subsumption Architecture, Reactive Planning, Autonomous Robots.

Abstract

The subsumption architecture [Brooks 1985] has been proposed as an effective approach for the construction of robust, real-time control systems for mobile robots. To investigate its strengths and weaknesses, a simulation of the architecture was developed: the Subsumption Architecture Tool (SAT). This simulation allows various models of system behavior to be quickly built and tested. During the building and testing of the Subsumption Architecture Tool, issues related to some architectural features became evident:

- Level of commitment of each layer
- Code redundancy
- Problem decomposition and programming style
- Complexity of large systems
- Abstract reasoning capabilities

The effects of these issues are presented with respect to the design and implementation choices of two sample layers of behavior. These layers are used to illustrate considerations that should be taken into account (1) when a project team is considering the use of the subsumption architecture or (2) when a subsumption architecture-based system is being designed and implemented.

Status

Research

Domain

Simulation of an autonomous, mobile robot in a warehouse or factory environment.

Language

Common LISP

Effort

1 (one) person-year

Experiences with the Subsumption Architecture

John E. Arnold
Digital Equipment Corporation[1]
295 Donald Lynch Boulevard DLB12-2/D4
Marlborough, MA 01752

1 Introduction

The development of autonomous systems — systems that are able to act and attain goals without human intervention — continues to be a goal of much research in the "artificial intelligence" community. This research has shed light on the requirements for a truly autonomous system. Attempts to implement simple systems based on these requirements have resulted in proposed architectures that are designed specifically for autonomous robots.

The current work evolved from the desire to review the proposed architectures and develop a basis for their quantitative comparison [Dean 1987]. This research uses the Subsumption Architecture Tool [Arnold 1988], a simulation of Brooks' subsumption architecture [Brooks 1985], to investigate the architecture's utility in the development of a few simple behaviors. This investigation has resulted in the identification of five issues which should be considered when one is considering the use of the subsumption architecture in their system or when a subsumption architecture-based system is being developed:

- Level of commitment of each layer
- Code redundancy
- Problem decomposition and programming style
- Complexity of large systems
- Abstract reasoning capabilities

The domain for the sample behaviors is a simulation of low-level actions that a mobile robot in a warehouse would routinely encounter.

[1]This work was completed while the author was at the Brown University Department of Computer Science as a participant in Digital's Graduate Engineering Education Program. The author would like to acknowledge Tom Dean and others at the Brown CS Department for their assistance during the course of this work.

1.1. The Subsumption Architecture

The subsumption architecture [Brooks 1985, 1986, 1987] is a result of an investigation into the requirements for and implementation of a control system for an autonomous mobile robot. The architecture results in the creation of simple systems that react to complex environments. This type of system can be traced to Simon's "Parable of the Ant" [Simon 1969].[2]

Brooks has identified the following requirements for an intelligent, autonomous, mobile robot [Brooks 1985]: the ability to pursue multiple goals, the ability to respond to multiple sensors with varying degrees of accuracy, robust performance, and extensibility. To meet these requirements, Brooks has proposed and implemented a scheme where the tasks to be achieved (by the robot) are decomposed into levels of competence. Each level of competence represents a complete, working behavior which the robot can successfully execute. As more complex problems are tackled, new levels of competence are layered on top of the existing levels without altering the underlying layers. In this process, each added layer subsumes the existing layers. Thus, the term Subsumption Architecture.

Each level is built of simple finite state machines augmented with local memory. Conceptually (and in some cases, in reality), the finite state machines all operate in parallel; there is no central flow of control and there is no shared, global memory. These machines (called modules) are allowed to communicate by passing simple messages along wires.[3] Once programmed, debugged, and wired, a

[2]Other examples of reactive systems can be found in [Fox 1984], [Kaelbling 1986], [Firby 1987], and [Sanborn 1987].

[3] In [Brooks 1985], these messages are portrayed as arbitrary LISP data structures. In practice, however, it is best to think of them as somewhat more constrained. In keeping with the spirit of the architecture, the messages should be something that really could be passed along a wire. For instance, [Connell 1987] uses messages that are single bit logic signals.

level is not altered as the robot's behavior is extended. Instead of altering the existing levels, new levels are allowed to override existing behaviors. This is accomplished by either inhibiting the outputs or suppressing the inputs of the subsumed levels.

Inhibition and suppression are two methods of allowing a higher-level module to exert some control over a lower-level module. Inhibiting a module's output allows one module to prohibit another module from sending a message or activating an effector such as a drive motor. Suppressing a module's input allows one module to prevent another module from receiving a message that was sent to it and provides a means for a higher-level to control behavior into a lower-level module by injecting data (perhaps fictitious, perhaps slightly altered) into the lower-level module's input. Inhibitors and suppressors must be wired into the web of modules in a manner similar to the communication links described above.

2 Experiments with the Subsumption Architecture Tool (SAT)

BRIE (Brown Robotics Implementation Environment) is a simulated robotics environment developed at Brown University [Dean 1987a]. It is intended to be used as a test bed for various robot control and robot problem-solving techniques [Dean 1987a]. BRIE consists of subsystems that provide capabilities such as spatial representation and 3-D modeling, discrete-event simulation control, robot emulation at the sensor and effector level, and robot control system simulation. The Subsumption Architecture Tool is an example of one of the robot control systems provided to the BRIE user.

One of the experimental systems that has been simulated with the Subsumption Architecture Tool begins with two layers of primitive behavior. At the lowest level is a set of controller modules that provide access to the effectors for variables such as acceleration and steering direction. The next highest level is a wall-finding and avoidance layer in which messages are sent to the controller layer in order to move towards an object that seems to be a wall without hitting it.

These two levels are presented as an example of the types of interaction that one may want to implement in a system. This example shows one way in which a robot's behaviors can be decomposed into discrete layers and how intra- and inter-module communication can be used to build increasingly complex tasks. Source code for the SAT implementation of these layers can be found in [Arnold 1988].

2.1. The Controller Layer

The modules and the wires in the controller layer implement behaviors which process requests dealing with the robot's effectors. The level of competence represented here is the ability for the robot to attain a specified velocity and orientation. A "panic" stop that models a hard braking condition is also provided. In order to realistically represent the processes which are used to reach a target velocity and orientation, additional modules that model acceleration and turn radius are used. A schematic of this layer is shown in Figure 2-1.

2.2. The Wall Finding and Avoidance Layer

This layer provides a simple facility for finding possible walls and heading towards them. In case obstacles are met on the way or in case the robot's speed gets it to a wall more quickly than expected, a simple avoidance module is also provided. These modules <u>only</u> affect the robot controllers via messages sent to modules in the Controller layer. A schematic of this layer is shown in Figure 2-2.

3 Results and Observations

Building the examples presented in Section 2 provided an opportunity to investigate the utility of the subsumption architecture. One of the advantages of a <u>simulated</u> subsumption architecture is the ability to experiment with different approaches and the effects of different constraints without having to take the time to build the robot itself. One can successfully argue that the full ramifications of these design decisions are not fully understood until the fully realized system is built and operating in its intended environment. However, by experimenting with various strategies in an easy-to-manipulate environment such as BRIE, some ideas can be eliminated in the early stages thus saving time for a better implementation of the chosen strategy.

The examples of Section 2 are not intended to be the epitome of subsumption architecture programming. The strategies implemented in the Controller and Find Wall/Avoidance layers were chosen because they are different from the examples presented in [Brooks 1985]. By presenting an alternative view of the architecture's use, it is hoped that the reader familiar with the existing subsumption architecture literature will be able to recognize the utility of the SAT as an investigative tool. A different approach was also chosen in hopes of getting a better understanding of the strengths and limitations of the subsumption architecture.

During the building and testing of the Subsumption Architecture Tool, some issues related to the architecture became evident. These issues, discussed in the next five sections, are related to the following areas:

- Level of commitment of each layer
- Code redundancy
- Problem decomposition and programming style
- Complexity of large systems
- Abstract reasoning

3.1. Level of Commitment of Each Layer

One of the distinguishing features of the subsumption architecture approach to system building is the strong commitment made by the implementor when each layer is deemed "complete." The strength of this commitment is stated as follows:

"We start by building a complete robot control system which achieves level 0 competence. It is thoroughly debugged. We never alter that system." ([Brooks 1985], p. 7)

The idea that a completed layer will not be altered in future revisions (<u>except</u> via suppression and inhibition connections) is critical to the idea that the use of a subsumption architecture offers a robust control system. With this commitment, it is argued that it is rare to experience a loss of competence as new layers are added. Without this commitment, it is much more difficult to verify that implementation changes have not altered previous functionality.

By disallowing the alteration of the code once a layer has been debugged, the designer of each layer must exhibit a degree of foresight due to the flexibility that is lost. This was noted early in the development of the subsumption architecture[4] and was encountered again while using the SAT. It should be noted that there is only one input line to each of the three controller modules that are expected to receive requests from higher modules (`request-velocity`, `request-orientation`, and `stop`). Since the architecture only allows one source for each input, it will not be possible for other modules to place requests to these modules.

In retrospect, it would have been wiser to design an interface module for each of these behaviors which would have more input wires (to be used as future needs arose) and which would initiate the applicable operation when a new message was received at any of those lines. This solution is still not optimal, however, since a predefined limit of the number of sources for requests is fixed at the time the `controller` layer is implemented.

[4] In [Brooks 1985], Brooks notes his dissatisfaction that a level 0 module included a certain output only because it was known to be needed in level 2. Had this need not been known ahead of time, another strategy for obtaining the desired behavior would have been required.

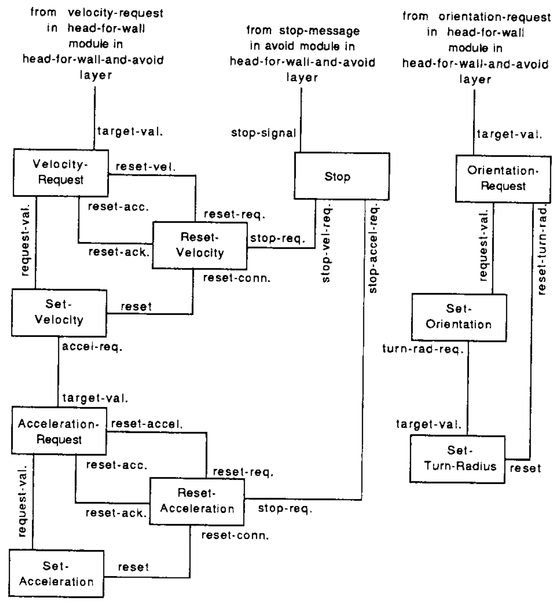

Figure 2-1: The Controller Layer

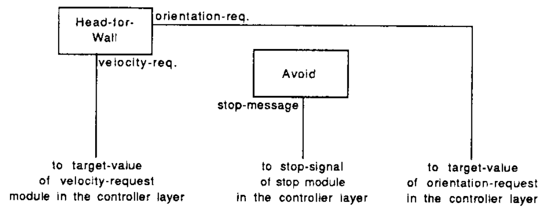
Head-for-Wall
orientation-req.
velocity-req.
Avoid
stop-message
to target-value of velocity-request module in the controller layer
to stop-signal of stop module in the controller layer
to target-value of orientation-request in the controller layer

Figure 2-2: The Head for Wall & Avoidance Layer

A better solution that still obeys the architecture constraints is needed. One approach would call for the implementation of a "request bus" that would look like a wire shared by many modules. However, this approach would seem to violate the assumptions that an input can have only one source and that there is no shared global memory. Another solution would allow us to eliminate the `controller` layer entirely by having each layer access the effectors as needed. This leads to the issue of code redundancy in the subsumption architecture.

3.2. Code Redundancy

The implementation of the examples shown in Section 2 resulted in the identification of two areas in which code redundancy may become an issue in the subsumption architecture. These areas are the need for parameterized behaviors (those that follow the same "rules" but whose exact behavior is determined by the values of various parameters) and the need for behaviors that are subsets of a those found in other layers.

Parameterized Behaviors

In terms of code reusability or integrated circuit "real estate", it is desirable to implement a common behavior only once and then give other layers access to it. However, providing this functionality within the constraints of the subsumption architecture requires a great deal of foresight by the designer. The initial design must provide sufficient means of access to the behavior in the manner needed by all future modules in order to eliminate redundancy.

The decision to implement a `controller` layer was based on the desire to develop a fully tested library of common controller routines. If a good design for sharing these behaviors cannot be found, the alternative is to implement identical code within each layer that needs one of the routines. This redundancy would greatly add to the complexity of the code, increase the opportunity for inducing errors into the system, and increase the storage/hardware necessary for an on-board implementation of the control system.

As previously mentioned (in Section 2), the current implementation of the `controller` layer must be changed to meet the goal of minimized code redundancy if new modules want to use the behaviors in that layer. Since only two layers have been shown in the example system, changing the `controller` layer is not that great a task (even though it violates our commitment to freeze a layer once it has been tested and deemed complete). In a more complex system simulation, however, redesigning a lower layer may be neither easy nor advisable.

Subset Behaviors

Consistently and efficiently providing behaviors that are subsets of behaviors found in other modules can also raise the issue of code redundancy in the subsumption architecture. Consider the following situation that arose during the implementation of a layer higher than those presented in Section 2. Presume that a lower level has implemented an avoidance behavior such that objects within a certain range will cause the robot to stop. Now, suppose a higher level wants to get the robot to maneuver through a doorway that is sufficiently narrow to trigger the lower level's avoidance behavior. An obvious solution to this situation is to inhibit the output of the lower level module that would cause the robot to stop.

Due to the level of abstraction chosen in the avoidance behavior described above, avoidance in this example becomes an all-or-nothing proposition. The higher level really wants to turn off the avoidance only for the objects that it is tracking. (In this case, we assume that the module is deactivating avoidance of the doorway because it has implemented its own mechanism for making sure that the robot does not hit it.) It would be desirable to have the lower level avoidance behavior take effect if another object came into view.

One can see the problems that arise in this situation due to the incompatibilities between the goals of the two layers. The lower layer exhibits a behavior that restricts the higher layer from achieving its level of competence. Therefore, the higher layer has no choice but to inhibit the lower layer. The higher layer still wants to avoid objects that are not the doorway, however, so it must re-implement an avoidance procedure for any object it detects that is not presumed to be part of the doorway. Thus, code redundancy has entered into the scenario due to the similarity (but not an exact match) of goals in multiple layers.

One could argue that this type of code redundancy enters into the system due to a design that failed to provide the right level of abstraction for common behaviors. Perhaps an omniscient designer could account for every view of common problems and provide a correct solution. Since the subsumption architecture embodies a philosophy that encourages orderly evolution of system functionality, it should be expected that parts of the system will be implemented before all future ramifications of the design and implementation decisions are known. It is to the architecture's credit that a well-defined means of overriding existing system behavior is available to the implementor. The costs of this flexibility are the effort and resources used to re-implement and re-test behaviors that are very similar to those found in other layers.

3.3. Problem Decomposition and Programming Style

Before programming a subsumption architecture-based system, it is important to determine the general approach to the problem at hand. The approach that is chosen will guide the decomposition of the problem into layers and will help specify the level of competence that should be implemented with each layer. For instance, Section 2 showed the two lowest layers from a decomposition that calls for a wall-following layer and a doorway-detect-and-traverse layer to be added next. Even though the subsumption architecture's evolutionary capabilities allow one to pick a level of competence and implement it, early use of the SAT showed that rushing into implementation of the first levels without attending to the overall problem decomposition leads to an ad hoc design that is neither efficient nor easy to extend.

One of the difficulties that arose during the early use of the SAT concerned the lack of documented criteria on which to base the decomposition decisions. The examples in the literature (e.g., [Brooks 1985], [Brooks 1986b], [Connell 1987]) show successful decompositions in varying degrees of detail. Unfortunately, there is little explanation of <u>why</u> the chosen decompositions were used. It would be useful to know if other decompositions were attempted and what features and/or decisions led to the better designs. For instance, all of the examples (including those presented in Section 2) include some form of avoidance as a very low level behavior. It would be helpful to know if there are reasons other than the intuition of the designer that led to this decision.

Once the decomposition has been chosen, the programming can begin. Programming a subsumption architecture-based system involves a local view of activity (i.e., the finite-state machine within a module) and a more global view (i.e., the communication between modules and layers). As with any programming activity, a style of programming emerges with continued use. The examples in the subsumption architecture literature are useful guides for a beginner but do not directly advocate a particular style.

The small number of language constructs for the finite-state machine require each user to develop a method for concepts such as do-while, do-until, and wait-for-condition. While the finite-state machine language offers a great deal of expressive power in a small language, the fact that each user must adopt or develop a consistent style places a somewhat larger burden on the programmer than programming languages that encourage a program style by providing commonly used program constructs. This point may be minor but in a complex system that is being implemented over time by a team of people, it is important that one's code be easy to understand (or that accurate documentation of the intention of the code is provided; preferably both). This encourages others to understand the original intention and will allow better decisions about the future use (or suppression or inhibition) of the functions provided.

The subsumption architecture provides some flexibility in terms of inter-module communication. The semantics of the `event-dispatch` and `conditional-dispatch` forms are well defined in terms of how new messages are handled by a module. There are no rigid requirements that dictate the style of message sending, however. In fact, two different communication styles have emerged [Cudhea 1988]. One style sends a message over a wire only once. The alternate style "strobes" (i.e., repeatedly sends) the message for a fixed duration or until some desired activity has been noticed.

The former style, sending a message only once, is used in the examples shown in this paper. It provides an economy of messages and thus reduces the use of system or module communication resources. This style also requires more attention to the possibility of losing messages. Thus, the design may call for tighter loops around the states in which messages are detected. To accommodate these tight loops, the problem may be decomposed further to provide for minimized message loss (as in the case of the `velocity-request` and `orientation-request` modules). The latter style, repeatedly sending messages, minimizes message loss but uses more system or module communications resources. This style may result in fewer modules (since decompositions such as those explained above are not as necessary); but this can result in more complex modules.

The Subsumption Architecture Tool leaves the choice of communication style to the user. The choice of style, however, will affect the programs written for the modules. It should also be noted that the styles do not necessarily mix well. Consider the `orientation-request` module. It is implemented as a tight loop in order to lose as few messages on the `target-value` input as possible. Whenever a new message is detected on the input, the `set-turn-radius` module is reset and the new target value is passed on to the `set-orientation` module. This works fine for the expected circumstances; that is, situations in which new target orientations are received infrequently.

If a higher layer was "strobing" the orientation request on the `target-value` wire, the behavior could be noticeably (and detrimentally) affected. As the higher layer strobes its orientation target value, the `orientation-request` module would be seeing the repeated messages and constantly resetting the `set-turn-radius` module. This resetting could result in a situation where the turn-radius controller never steps to the next value. The result of this is that the robot's current turn radius would never be altered. Thus, the desired effect would never be attained.

From this example, one can see that it is necessary for a single style of communication to be used on each wire. To minimize errors and confusion, it is probably best to use a single style for the entire system.

3.4. Complexity of Large Systems

Figures 2-1 and 2-2 indicate the complexity of the systems that can be easily built with the subsumption architecture. The two layers in Section 2 accomplish very little when compared to the requirements of an autonomous system that attempted to solve complex problems in a real environment. It is easy to imagine the size of a module schematic diagram growing to wall size or larger for a complex system. This reveals a fundamental dichotomy in the architecture: the simplicity of each module and the complexity of the overall system.

When implementing an individual layer, it is possible to decompose the level of competence into a set of modules that simply reflect the behavior of that layer. Programmers are able to attain reasonably good results at this level of detail in a complex design. Thus, the layer-level design, although requiring discipline and care, is not the critical issue in managing system complexity.

Managing the relationships between layers is a critical issue as the number of layers and the scope of the layers grows. In order for a new layer to be added to the system, it is important that the designer/implementor of that layer understands the existing layers. This knowledge is required for two reasons: (1) the user may be able to use an existing behavior to his/her advantage and (2) the user may have to suppress or inhibit some modules in order to attain the goal of the layer he/she intends to implement.

Since the level of information presented in a layer is still quite detailed (e.g., modules and wires), something more abstract is needed to communicate the intentions and effects of each layer. At the very least, this information must be made available to the programmer. To accomplish this, it is desirable to determine a means to convey the system's knowledge in a manner that maps easily to the user's "mental model" of the system. This investigation into higher-level system modelling and representation could provide a transition to the longer-term needs of increasingly intelligent systems. As suggested in [Georgeff 1987], increasingly capable autonomous systems will require more explicit representation of module/layer goals, intentions, knowledge, and ability.

3.5. Abstract Reasoning

All of the subsumption architecture applications known to the author involve relatively low-level robot control. Admittedly, the behaviors are still quite impressive given the limited machinery required. Whether the architecture can be applied effectively to more abstract problem solving remains an open question. This question will be answered in the future as the use of the subsumption architecture continues to grow and larger problems are tackled.

The successful use of the subsumption architecture in higher-level planning will require the effective solution to a number of constraints imposed by the architecture. The experience to date with the Subsumption Architecture Tool would indicate that at least two areas will need to be addressed as the architecture is used to tackle more abstract problems: (1) communications between modules and (2) shared global memory.

The efficiency of the communications scheme (i.e., the use of dedicated wires between modules) will be challenged when higher-level problem-solving modules need to reason about complex representations of the world, other agents, etc. The current hardware implementations of subsumption architecture-based systems ([Brooks 1986a], [Connell 1987]) are built with very low bandwidth (sometimes even single bit) communications. More complex representations will require much higher bandwidth. At a higher bandwidth, the issue of predefining reasonable interconnections at lower layers for use by not-yet-implemented higher layers will become even more critical than it is currently.

Complex representations will also challenge the requirement that there be no shared global memory. This restriction will require each module to keep its own copy of any needed representation (e.g., a map). In addition to the communication problem addressed above, this constraint also could present a space problem as more powerful modules need to keep more knowledge and/or data at hand in order to make appropriate decisions. This restriction also precludes the use of traditional database technology that could provide shared, persistent objects to the various modules. A tradeoff between the robustness offered by the lack of shared, global memory and the need to minimize redundant data storage may be necessary. In one possible scenario, the minimal "survival" layers that require the most robustness could obey the restriction while higher layers that perform more abstract computations could relax the restriction.

4 Summary

This research has resulted in the production of a tool that allows experimentation with the subsumption architecture in a simulated setting: the Subsumption Architecture Tool. The use of the simulated environment encourages the user to attempt many different decomposition and implementation strategies before committing the design to hardware. Since the SAT implements the complete syntax and semantics of the architecture as originally described in [Brooks 1985], the experimenter familiar with existing implementations can transfer the results of the simulation to actual system implementation with minimal conversion difficulties.

The use of the Subsumption Architecture Tool has highlighted some areas in which the architecture will be challenged as the goals of systems being built are progressively extended to include more abstract reasoning capabilities. These areas are (1) the level of commitment made by each layer, (2) the code redundancy that can occur, (3) the effect of problem decomposition and programming style decisions, (4) managing the complexity of large systems, and (5) the ability to provide abstract reasoning capabilities within the constraints of the architecture.

5 References

[Arnold 1988] Arnold, John E., "SAT: A Subsumption Architecture Tool for Simulated Robot Control", Master's Thesis, Brown University Department of Computer Science, May 1988.

[Brooks 1985] Brooks, Rodney A., "A Robust Layered Control System For a Mobile Robot", MIT AI Memo 864, September 1985.

[Brooks 1986] Brooks, Rodney A., "A Robust Layered Control System For A Mobile Robot", IEEE Journal of Robotics and Automation, IEEE Press, April 1986.

[Brooks 1986a] Brooks, Rodney A. and J. Connell, A. Flynn, "A Mobile Robot with Onboard Parallel Processor and Large Workspace Arm", Proceedings of *AAAI-1986*, Morgan Kaufmann Publishers, Inc., August 1986.

[Brooks 1986b] Brooks, Rodney A. and J. H. Connell, "Asynchronous Distributed Control System for a Mobile Robot", Proceedings of SPIE *Cambridge Symposium on Optical and Optoelectronic Engineering*, Cambridge MA, October 1986.

[Brooks 1987] Brooks, Rodney A., "A Hardware Retargetable Distributed Layered Architecture for Module Robot Control", Proceedings of *IEEE 1987 International Conference on Robotics and Automation*, Raleigh NC, IEEE Press, April 1987.

[Connell 1987] Connell, Jonathan H., "Creature Design with the Subsumption Architecture", Proceedings of *IJCAI-1987*, Morgan Kaufmann Publishers, Inc., August 1987.

[Cudhea 1988] Cudhea, Peter and J. Connell, MIT, personal communication, January 1988.

[Dean 1987] Dean, Thomas L. "Benchmarks for Research in Planning", Proceedings of *AAAI-1987 Workshop on AI and Simulation*, Seattle WA, July 1987.

[Dean 1987a] Dean, Thomas L. and K. Basye, M. Lejter, T. Engel, J. Arnold, "BRIE: The Brown Robotics Implementation Environment", Brown University, Department of Computer Science Technical Report, Providence RI, August 1987.

[Firby 1987] Firby, R. James, "An Investigation into Reactive Planning in Complex Domains", Proceedings of *AAAI-1987*, Morgan Kaufmann Publishers, Inc., July 1987.

[Fox 1984] Fox, Mark S. and S. Smith, "The Role of Intelligent Reactive Processing in Production Management", In *13th Meeting and Technical Conference, CAM-I*, November 1984.

[Georgeff 1987] Georgeff, Michael and A. L. Lansky, "Reactive Reasoning and Planning", Proceedings of *AAAI-1987*, Morgan Kaufmann Publishers, Inc., July 1987.

[Kaelbling 1986] Kaelbling, Leslie Pack, "An Architecture for Intelligent Reactive Systems", Proceedings of *Workshop on Planning and Reasoning about Action*, June-July 1986.

[Sanborn 1987] Sanborn, James C. and J. A. Hendler, "A Model of Reaction for Planning in Dynamic Environments", Proceedings of *Knowledge-Based Planning Workshop*, Defense Advanced Projects Research Agency, Austin TX, December 1987.

[Simon 1969] Simon, Herbert A., *The Sciences of the Artificial*, MIT Press, 1969.