Tapir: the Evolution of an Agent Control Language

Tracking Number: 322

Gary W. King and Marc S. Atkin and David L. Westbrook and Paul R. Cohen
University of Massachusetts
140 Governor's Lane
Amherst, MA 01003
{gwking, atkin, westy, cohen}@cs.umass.edu

Abstract

Tapir is a general purpose, semi-declarative agent control language that extends and enhances the Hierarchical Agent Control (HAC) architecture (Atkin, Westbrook, & Cohen 2000a). Tapir incorporates the lessons learned from developing HAC, making it easier and faster to create reusable and understandable actions. Tapir has been used in a battalion level wargame simulation, a robot simulator, a simulation of cellular dynamics and a simulation of rodent behavior. The language is built around constructs that define agents, sensors, actions, and messages. It has mechanisms for handling multiple agents, a flexible resource model, and multiple means for structuring concurrent actions. This paper provides an overview of HAC and its shortcomings and then explains how Tapir extends and improves upon it.

Introduction

Building on previous work in simulation and agent control (Atkin, Westbrook, & Cohen 2000a; Cohen *et al.* 1989; St. Amant 1996) we have created Tapir: an expressive agent control language with a simple syntax. Tapir has been used in multiple systems by experienced and novice agent programmers and has fostered dramatic increases in productivity.[1] Tapir constructs are self-contained and modular, making them human-understandable and machine-parsable. Tapir is a semi-declarative, general purpose agent control language implemented in Common Lisp. Tapir has constructs for defining agents, sensors, actions, and messages:

Agents Agents are viewed primarily as resources for actions. They connect actions to the physics of the real or simulated domain. As resources, agents come in many different types. For example, they can be serializably reusable, sharable, composite, consumable and so on. Individual resources may exist in a hierarchy. For example, a robot is an agent consisting of motor resources, camera resources and a gripper.

Sensors Sensors tie actions to the world. They can be "primitive", connecting to the world directly, or they can be abstract, amalgamating and processing data from multiple sources. Sensors can be shared by multiple actions.

[1] Using the admittedly imprecise Lines of Code metric, Tapir actions require from two to four times *less* effort.

Actions Actions use resources and sensors to carry out their tasks. Each action exists in the context of a control hierarchy and may extend the hierarchy by creating children of its own. This hierarchy is grounded in "primitive" actions that use resources to achieve their ends. Tapir has many constructs for structuring control, including sequential, parallel and repeated execution.

Messages Actions, sensors, and resources communicate via messages. For example:

- sensors send actions messages when events in the world occur.
- child actions send their parent messages when they complete or when they need to communicate some change in status.
- parents send their children messages when they wish to stop or redirect their activities.
- resources send their actions messages when they change or are destroyed.

Each message is an instance of a particular class and can carry additional information. For example, a FAILURE message carries the reason for its failure and a CHANGE-SPEED message carries the value of the desired speed.

Tapir is built upon and extends the Hierarchical Agent Control (HAC) architecture (Atkin *et al.* 1998; Atkin, Westbrook, & Cohen 2000a). The rest of this paper provides a brief overview of HAC (section), describes Tapir's main constructs (section), and compares Tapir with existing languages (section). We conclude by discussing current applications and our future work.

The HAC architecture

HAC is a framework for controlling agent behavior. HAC takes care of the mechanics of executing the code that controls an agent, passing messages between actions, coordinating multiple agents, arbitrating resource conflicts between agents, and updating sensor values. HAC structures agent control into three hierarchies: control, sensor and context. These determine what to do, what is happening, and how to interpret it. The last is especially important in planning since the correct response to an event depends on the goals and interests of an action's ancestors. HAC is action-centered, not agent-centered. Rather than specifying what an agent is

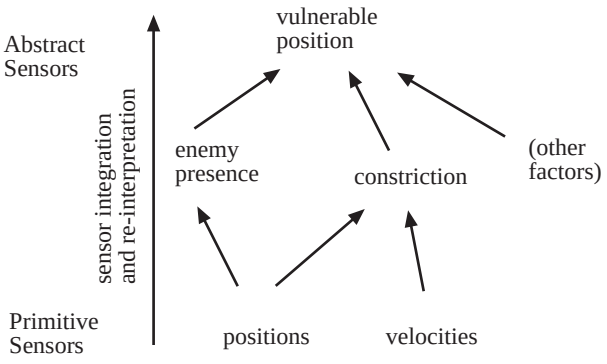

Figure 1: Raw sensor data is transformed into more complex concepts via the abstract sensor hierarchy.

to do, one specifies what is to be done. Actions then use whatever resources are available to accomplish their ends. We found that this inversion of the usual framework made it easier to incorporate both planning and multi-agent control into the architecture.

Agent control in HAC

HAC is a *supervenient* architecture (Spector & Hendler 1994). It abides by the principle that higher levels should provide goals and context for the lower levels, and lower levels provide sensory reports and messages to the higher levels ("goals down, knowledge up"). A higher level action cannot overrule the sensory information provided by a lower level, nor can a lower level interfere with the control of a higher level. Supervenience structures the abstraction process; it allows us to build modular, reusable actions.

HAC is an architecture; other than enforcing a general form, it does not place any constraints on how actions are implemented. Every action can choose which messages it will respond to. Although actions lower in the hierarchy will tend to be more reactive, whereas those higher up tend to be more deliberative, the transition between them is smooth and completely up to the designer. Unlike other architectures (Georgeff & Lansky 1987; Cohen *et al.* 1989; Gat 1997b), we do not prescribe a preset number of behavioral levels. Parents can run in parallel with their children or only when the child completes.

The Sensor Hierarchy

The sensor hierarchy provides a principled means of structuring the complexity of reading and transforming sensor information. It functions analogously to the HAC action hierarchy and reduces the complexity of sensor fusion. The sensor hierarchy is grounded by the low level primitives available from the physics of the world (real or simulated). These primitive sensors include the location of terrain features, the current speed and location of agents in the world, the status of a robot's bump sensors and so forth.

Each level in the hierarchy integrates and extends the level below it by compiling the available information and providing additional structure. We call these higher levels *abstract sensors*, since they do not sense anything *directly* from the world. For example, enemy location information can be combined into a sensor that specifies overall enemy presence; terrain information can be combined into a sensor that specifies passes and movement corridors. Furthermore, these two sensors can be combined to show enemy vulnerability: areas where enemy units are concentrated and cannot move quickly (see Figure 1).

The sensor hierarchy shares the control hierarchy's syntax and structure. Each sensor is analogous to a HAC action. It sends and receives messages and performs sensor computation each time it runs. One advantage of this is that the same principles learned in building actions carry over directly when building sensors. This linkage also makes it easy for actions to use sensors as part of their control mechanism. An action can react to sensor messages the same way it reacts to those from child actions. Each sensor is associated with the set of actions that request it and completes when this set becomes empty.

Like actions, sensors abide by the principle of supervenience. Higher-level sensors integrate and interpret lower-level ones but they do not change the lower-level information. Lower-level sensors provide information to the higher-level ones but they do not tell them what to say. One advantage of this is that each level of the hierarchy can be viewed independently without worrying about the levels coming into it or the levels that are using it.

In summary, HAC provides a simple and flexible architecture for agent control and sensor fusion. Tapir extends the HAC framework by codifying standard idioms, supporting complex action constructs, simplifying resource use, adding declarative meta-information and making actions easier to build, modify and debug.

Tapir Architecture

Tapir gains some of its power and simplicity by limiting the flexibility of the underlying HAC framework and clearly specifying the model that it implements. In particular, Tapir specifies a simple process model, a resource description language, and a powerful set of control constructs. Tapir also adds support for debugging, data collection, interactive design and an extended syntax that helps clarify the action writer's intent. Each of these will be discussed in turn after we first provide examples of some simple Tapir actions.

Actions are defined with **defaction**, a Lisp macro that defines a **CLOS** class (Steele Jr 1990; Kleene 1988) and the methods that implement it. Actions can be used as building blocks for the creation of further actions. For example, figure 2 shows how MOVE-TO-RANDOM-POINT can be used to build SWARM. SWARM's clauses specify that it should bind all of the resources available to it in its **agents** slot and create a MOVE-TO-RANDOM-POINT action for each of them in its **do** phase. The :FOREACH control construct is expanded at run-time and allows an unlimited number of children to be executed in parallel or sequentially. MOVE-TO-RANDOM-POINT's clauses specify that it should bind only a single resource (the default) to its **agent** slot and that its **do** phase consists of starting a single MOVE-TO-POINT action with a random destination. Finally, the :ON-

```
(defaction move-to-random-point ()
  :documentation "Move an agent to a randomly
                  selected location."
  :resources (agent)
  :do
   (move-to-point
    :target-geom (find-random-location-for-agent
                  the-simulation agent))
  :on-message (message (stop-children) :restart))

(defaction swarm ()
  :documentation "Move any number of agents to
                  random locations repeatedly."
  :resources ((agents :count :all))
  :do
   (:foreach agent in agents :in-parallel
             (move-to-random-point :agent agent))
  :ignore-messages)
```

Figure 2: Implementation of multi-agent "swarm" behavior in Tapir.

MESSAGE clause indicates that MOVE-TO-RANDOM-POINT should restart whenever it receives any kind of message.[2] This might be a message from the MOVE-TO-POINT child action or one from the agent it is using as a resource.[3]

Process Flow in Tapir

Figure 3 displays a schematic of an action's life cycle. In brief: an action is created, enters its **do** phase (perhaps many times) and then completes. The details of this simplified description are filled in by special initialization and finalization code, sensors, process monitors, child actions, resource utilization and more. Figure 3 provides both a schematic of the process flow and the Tapir language constructs that can be used to influence and describe what an action should do. We summarize the phases below.

Initialization Actions maintain internal state in their slots; they also may require resources to do their job. When an action is created, its slots are initialized and any required resources are found and bound. Created actions do not do anything until they are actually scheduled to be run and the time for their activation arrives.

The Do Phase When activated, the action enters its **do** phase. Actions may execute special code at the beginning of the very first **do** phase; any sensors required by the action are also found or created at this time. The **do** phase itself consists of pre-checks to see if any messages should be sent to the action's parent (this may possibly complete the action without it ever having executed); the actual execution; and then post-checks to see if any messages should be sent.

Sleeping Once an action executes, it will go to sleep unless it has been rescheduled. A sleeping action remains asleep until it is explicitly rescheduled (perhaps by a parent) or until it receives a message.

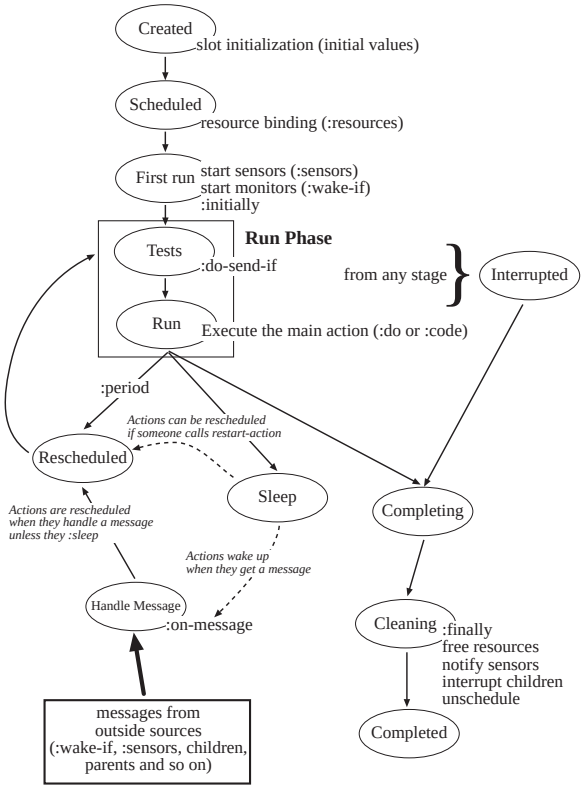

Figure 3: Process Flow in Tapir.

Message Handling Actions specify how to respond to each message type. These responses include: running code, reentering the **do** phase, sending the message on up to the action's parent, and ignoring the message completely.

Completion and Cleaning When an action is sleeping and it has no sensors or children that can wake it up, it automatically completes. Actions can also be interrupted at any time. In both cases, the action enters a cleaning phase where resources are unbound, finalization code is run, children are interrupted and so on.

As mentioned above, messages are used to control the flow of execution between parents and their child actions. Children can complete themselves explicitly by sending a message of type completion (e.g., success or failure) to their parents. If a child completes for some other reason (e.g., a required resource is destroyed) then it will automatically send a completion message during its clean-up phase. Message passing is the only means of control transfer.

Child Control Constructs

An action's :DO clause specifies the potential sub-tasks of an action. It describes both the sub-tasks and how they will be executed. The :DO specification is grounded in the sub-tasks which can be either Lisp code (using the :CODE construct) or an action specification consisting of the child action's name and any of its parameters. The sub-tasks can be joined to-

[2] Messages are typed and MESSAGE is the root superclass for all other message types.

[3] It is instructive to compare the Tapir version of SWARM with its HAC counterpart in (Atkin *et al.* 2000). The Tapir version is significantly simpler and shorter.

gether using the following constructs:

:case This is analogous to the case or switch statements found in many languages. A form is evaluated and the result is used to determine which action specification to execute.

:foreach Dynamically creates a list of children and runs then. The children may be run using any of the child combination types (e.g., in parallel, in sequence and so on).

:in-parallel Runs a list of children in parallel. Each child in the list is started at the same time and runs independently. The order in which the children are initially started is unspecified.

:in-sequence Runs a list of children in sequence. Only one child at a time is active.

:repeat Runs a child action repeatedly for a count or until some predicate becomes true.

:one-of This is Tapir's version of a the Lisp **cond** form. It takes a list of predicates and action specifications. ONE-OF runs the first action specification whose predicate evaluates to true. An optional :OTHERWISE clause can be included to run a specification when none of the other clauses is applicable.

:unordered Runs a list of children in some arbitrary sequence. Like :IN-SEQUENCE, only one child is active at a time but the order of activation is unknown.

:when Each when clause consists of a predicate and an action-specification. If the predicate evaluates to true, the action-specification is run.

Simple actions are easy to build because of Tapir's straightforward syntax. The action combination mechanisms mean that it is also easy to build complex actions out of these building blocks. Control hierarchies that would have been difficult to build in raw HAC can be written easily in Tapir.

Miscellaneous

Tapir provides many facilities that speed action creation, debugging and use.

Superclass inference Action behavior is specified in part by the HAC infrastructure classes from which it inherits. For example, there are classes for child actions, parent actions, agent using actions, agent managing actions, periodic actions and so on. Part of defining an action in HAC is determining its proper superclasses. This is often a non-trivial problem due to the number of superclasses and the interactions between them. Tapir removes this problem entirely by automatically inferring the proper HAC superclasses based on the action specification.

Debugging and Instrumentation Tapir's :DEBUG and :INSTRUMENT clauses make it easy to add debugging code to an action and to collect information while an action is running. Each :DEBUG clause creates switchable code or print statements. By default, the name of the debug switch is the same as that of the action although the :DEBUG-NAME clause makes it possible to change this. Debugging switches can be turned on and off with the **debug** and **undebug** commands. When the switch for an action is on, the debug code and print statements will be executed. This low-level facility makes it very easy to see exactly what is going right–or wrong–during an action's execution.

Action instrumentation relies on the EKSL CLIP package (Westbrook *et al.* 1994). CLIP allows data to be collected from running systems by hooking metaphorical alligator clips onto variables just as one can clip wires onto an electric circuit. The :INSTRUMENT clause adds CLIP instrumentation to individual action variables (:PARAMETERS and :LOCALS) or to the action as a whole. The author can specify how often the CLIP's should be collected (for example periodically or only when a certain trigger condition obtain). This facility makes it easy to build hooks into a system for later experimentation and analysis.

An extended example

We conclude our discussion of Tapir's architecture with a substantive example (figure 4). The MOVE-IN-FORMATION action is used in the Capture the Flag wargaming simulator to move a group of agents in formation. Like SWARM, this action binds all of the resources available to it. It uses the :INITIALLY clause to determine which agent would make the best leader (using **select-leader)** and makes the rest of the agents followers. The :DO clause creates two parallel hierarchies—one for the leader and one for the followers. The former runs a leader move whereas the later determines formation criteria and then creates a separate move action for each follower and runs all of these in parallel. The action completes either when the leader move completes or if all of the agents are destroyed. The :ON-MESSAGE clause handles **resources-gone** messages from the agents of the action. A **resources-gone** message is generated when a resource is destroyed. The MOVE-IN-FORMATION action handles this message by selecting a new leader, stopping all of its children and restarting. For comparison, the original HAC code for MOVE-IN-FORMATION required more than three times the number of lines! Furthermore, the HAC code was spread out over four classes and seven methods.

Related Work

Although motivated largely by its HAC underpinnings, Tapir shares much with other agent control architectures. For example, ESL (Gat 1997a), DAML-S (Burstein *et al.* 2001) and PDDL (McDermott *et al.* 1998; McDermott 2000) roughly match Tapir's expressive power (see (Blythe 2001) for a comparison). But whereas ESL adds an action language to Common Lisp, Tapir wraps an action language around it. This difference is motivated in part by our desire to craft a simpler language usable by non-programmers but it also tends to make Tapir actions more modular and comprehendible.

PDDL and DAML-S are more concerned with describing services than actually implementing them. They tend towards the declarative whereas Tapir is more procedural in nature. We see these two poles as complementary and hope to weave some of DAML-S's ontologies into Tapir's executable model.

The APEX architecture also attempts to manage multiple

```
(defaction move-in-formation ()
  (:resources ((agent :count :all
                      :group-type group-blob)))
  (:parameters ((formation-shape :required)
                (distance 1.0)
                (target :required)))
  (:locals ((leader nil)
            (followers nil)
            (assignments nil)))
  (:initially
   (:code
    (setf leader (select-leader the-simulation
                                the-action)
          followers (remove leader (resources agent)))))
  (:on-message
   (resources-gone
    (:debug "MIF: ~A died" (name leader))
    (setf leader (select-leader the-simulation
                                the-action)
          followers (remove leader (resources agent)))
    (stop-children)
    :restart))
  (:do
    (:debug "MIF: Moving to ~A with ~A leading"
            target leader)
    (:in-parallel
     (:in-sequence
      (formation-leader-move :agent leader
                             :target target
                             :followers followers)
      (:generate completion)))

     (:in-sequence
      (:code
       (setf assignments
             (assign-followers-to-criteria
              the-simulation leader followers
              (create-criteria formation-shape
                               leader followers))))
      (:foreach (follower . assignment)
                in assignments :in-parallel
                (formation-follower-move
                 :agent follower
                 :target-agent leader
                 :distance (distance assignment)
                 :angle (angle assignment)
                 :final-target target
                 :use-follow-if-further-than
                 (* 2.0 distance))))))
  :send-messages-to-parent)
```

Figure 4: Multi-agent formation movement in Tapir.

tasks in complex, uncertain environments, placing particular emphasis on the problem of resolving resources conflicts (Freed 1998). The current version of Tapir is less concerned with planning and scheduling and more concerned with letting authors tell agents what to do. Future versions of Tapir will add a planning language superset.

Like PRS (Georgeff & Ingrand 1989), Tapir allows for the specification of blocking and non-blocking children (child actions that run in sequence with their parents or in parallel), and like later versions of RAP (Firby 1994), success and failure are treated like any other message, and do not implicitly determine the flow of control between actions.

Tapir and HAC use the same representation for actions at all levels of the hierarchy, and also for sensors. Contrast this with the majority of current agent control architectures, e.g. CYPRESS (Wilkins *et al.* 1995) and RAP (Firby 1996), which distinguish between procedural low-level "skills" or "behaviors" and higher level symbolic reasoning. Different systems are often used to implement each level (CYPRESS combines SIPE-2 and PRS, for example). Tapir does not conceptually differentiate between discrete actions and continuous processes, nor does it limit the the language used to describe them.

Conclusion and Future Work

Tapir has achieved many of its initial design goals: it is a simple, flexible and intelligible agent control language that is easy to use and understand. Even so, Tapir remains a work in progress. We are investigating three extensions: planning, real robot control and better programming environments.

Tapir for planning

Tapir is an agent control language but it is not a planning language. We intend to extend Tapir with constructs for pre- and post-conditions, invariants, temporal reasoning and goal specification by merging it with HAC's GRASP Planner (Atkin, Westbrook, & Cohen 2000b). Doing so will make Tapir significantly more powerful and more useful. The difficulty will be to keep it simple enough for SMEs to use productively.

Tapir for real robots

We have used Tapir to control simulated robots and our goal is to extend it (and the HAC substrate) so that actions can control real Pioneer II robots. The current HAC engine uses a centralized queue and imposes no constraints on the CPU time used by an action. Future engines will be operating in a real-time, decentralized environment, and will need to deal with widely varying time scales, from microseconds to days.

Visual Tapir

Programming is replete with syntax, keywords, options, flags and clauses that are difficult for non-programmers to remember. Like other visual tools, Visual Tapir will add scaffolding to the programming environment so that authors can focus on their goals and not on minutia.

Conclusion

This paper has introduced Tapir, an agent control language built on top of the supervenient HAC architecture. Tapir highlights include:

- A simple and consistent syntax for the specification of actions, sensors, process monitors, message handlers, slots, and resources.
- A powerful action combination facility that makes it easy to build complex actions out of simpler building blocks.

- A flexible resource ontology, description language, and specification language.
- A set of helpful tools include debugging aids, syntax customization, and documentation facilities.

Tapir has been used to control units in a simulated war game, simulated robots and cellular dynamics. No controlled studies have been run to validate its utility but informal evidence indicates that Tapir users enjoy the language and are significantly more productive.

Acknowledgments

This research is supported by DARPA/USAF under contract numbers F30602-01-1-0589, N66001-00-C-801/34-000TBD, DASG60-99-C-0074, F30602-99-C-0061, and F30602-00-1-0529. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of the Defense Advanced Research Projects Agency/Air Force Materiel Command or the U.S. Government.

References

Atkin, M. S.; Westbrook, D. L.; Cohen, P. R.; and Jorstad, G. D. 1998. AFS and HAC: Domain-general agent simulation and control. In *Working Notes of the Workshop on Software Tools for Developing Agents, AAAI-98*, 89–95.

Atkin, M. S.; King, G.; Westbrook, D.; Heeringa, B.; Hannon, A.; and Cohen, P. R. 2000. SPT: Hierarchical agent control: A framework for defining agent behavior. In *Proceedings of the Fifth International Conference on Autonomous Agents*. Autonomous Agents.

Atkin, M.; Westbrook, D.; and Cohen, P. 2000a. HAC: A unified view of reactive and deliberative activity. In *Working Notes of Fourteenth European Conference on AI Workshop on Balancing Reactivity and Social Deliberation in Multi-Agent Systems*. ECAI.

Atkin, M. S.; Westbrook, D. L.; and Cohen, P. R. 2000b. Domain-general simulation and planning with physical schemas. In *Proceedings of the 2000 Winter Simulation Conference*, 1730–1738.

Blythe, J. 2001. Mappings between SADL and other action languages (available at http://www.isi.edu/expect/rkf/sadl-mapping.html). Technical report, University of Southern California.

Burstein, M.; Hobbs, J.; Lassila, O.; Martin, D.; McIlraith, S.; Narayanan, S.; Paolucci, M.; Payne, T.; Sycara, K.; and Zeng, H. 2001. DAML-S 0.5 draft release (available at http://www.daml.org/services/daml-s/2001/05/). Technical report, DARPA.

Cohen, P. R.; Greenberg, M. L.; Hart, D. M.; and Howe, A. E. 1989. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine* 10(3):32–48. also Technical Report, COINS Dept, University of Massachusetts.

Firby, R. J. 1994. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, 49–54.

Firby, R. J. 1996. Modularity issues in reactive planning. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, 78–85.

Freed, M. 1998. Managing multiple tasks in complex, dynamic environments. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 921–927.

Gat, E. 1997a. ESL: a language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the IEEE Aerospace Conference*, 319–324.

Gat, E. 1997b. On three-layer architectures. In Kortenkamp, D.; Bonnasso, R. P.; and Murphy, R., eds., *Artificial Intelligence and Mobile Robots*. AAAI Press.

Georgeff, M. P., and Ingrand, F. F. 1989. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 972–978. Detroit, Michigan: AAAI Press, Menlo Park, CA.

Georgeff, M. P., and Lansky, A. L. 1987. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 677–682. MIT Press.

Kleene, S. E. 1988. *Object-Oriented Programming in Common Lisp: A Programmer's guide to CLOS*. Addison-Wesley.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. The PDDL planning domain definition language (available at http://www.cs.yale.edu/). Technical report, Yale Computer Science Department.

McDermott, D. 2000. The 1998 AI planning systems competition. *The AI Magazine* 21(2):35–55.

Spector, L., and Hendler, J. 1994. The use of supervenience in dynamic-world planning. In Hammond, K., ed., *Proceedings of The Second International Conference on Artificial Intelligence Planning Systems*, 158–163.

St. Amant, R. 1996. *A Mixed-Initiative Planning Approach to Exploratory Data Analysis*. Ph.D. Dissertation, University of Massachusetts, Amherst. Also available as technical report CMPSCI-96-33.

Steele Jr, G. L. 1990. *Common Lisp: The Language*. Digital Press, second edition.

Westbrook, D. L.; Anderson, S. D.; Hart, D. M.; and Cohen, P. R. 1994. CLIP: Common lisp instrumentation package. Technical Report 94-26, University of Massachusetts at Amherst, Computer Science Department, Amherst, MA 01003.

Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; and Wesley, L. P. 1995. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI* 7(1):197–227.