The Behavior-Oriented Design of Modular Agent Intelligence

Joanna J. Bryson

University of Bath, Department of Computer Science
Bath BA2 7AY, United Kingdom
jjb@cs.bath.ac.uk [+44] (0)1225 38 6811

Abstract. Behavior-Oriented Design (BOD) is a development methodology for creating complex, complete agents such as virtual-reality characters, autonomous robots, intelligent tutors or intelligent environments. BOD agents are modular, but not multi-agent systems. They use hierarchical reactive plans to perform arbitration between their component modules. BOD provides not only architectural specifications for modules and plans, but a methodology for building them. The BOD methodology is cyclic, consisting of rules for an initial decomposition and heuristics for revising the specification over the process of development.

1 Introduction

This chapter examines how to build complete, complex agents (CCA). A *complete agent* is an agent that can function naturally on its own, rather than being a dependent part of a Multi-Agent System (MAS). A *complex agent* is one that has multiple, conflicting goals, and multiple, mutually-exclusive means of achieving those goals. Examples of complete, complex agents are autonomous robots, virtual reality (VR) characters, personified intelligent tutors or psychologically plausible artificial life (ALife) [2, 8, 25]. Being able to reliably program CCA is of great practical value both commercially, for industrial, educational and entertainment products, and scientifically, for developing AI models for the cognitive and behavioural sciences.

All of the methodologies I describe in this chapter are modular. The advantages of treating a software system as modular rather than monolithic are well understood. Modularity allows for the problem to be decomposed into simpler components which are easier to build, maintain and understand. In particular, the object-oriented approach to software engineering has shown that bundling behavior with the state it depends on simplifies both development and maintenance.

The problem of when to treat a module as an individual actor, an *agent*, is currently less well understood [3]. Unlike most chapters in this volume, this chapter separates the issues of agency from the issues of MAS. This chapter addresses how to develop and maintain a system that pursues a set of goals relatively autonomously, responding to the challenges and opportunities of dynamic environments that are not fully predictable. However, it does this without agent communication languages, negotiation or brokering. The assumption is that the system will be run on a single platform where no module is likely to die without the entire system crashing, and that the system is being

developed by a relatively small team who can share code and interfaces. In other words, the assumption is that this is in many respects a normal software engineering project, except that it is producing an intelligent, proactive system.

2 The Previous State of the Art

The last decade of research has shown impressive convergence on the gross characteristics of software architectures for CCA. The field is now dominated by 'hybrid', three-layer architectures [19, 22]. These hybrids combine the following:

1. *behavior-based AI* (BBAI), the decomposition of intelligence into simple, robust, reliable modules,
2. *reactive planning*, the ordering of expressed actions via carefully specified program structures, and
3. (optionally) *deliberative planning*, which may inform or create new reactive plans, or, in principle, even learn new behaviors.

In this section I will discuss these systems and their history in more detail. The remainder of this chapter presents an improvement to three-layer architectures, called Behavior-Oriented Design.

2.1 Behavior-Based Artificial Intelligence (BBAI)

BBAI was first developed by Brooks [6], at a time when there were several prominent modular theories of natural intelligence being discussed [13, 17, 29]. In BBAI, intelligence is composed of a large number of modular elements that are relatively simple to design. Each element operates only in a particular context, which the module itself recognizes. In Brooks' original proposal, these modules are finite state machines organized into interacting *layers*, which are themselves organized in a linear hierarchy or stack. The behaviors have no access to each other's internal state, but can monitor and/or alter each other's inputs and outputs. Layers are an organizational abstraction: the behaviors of each layer achieve one of the agent's goals, but a higher layer may subsume the goal of a lower layer through the mechanism of affecting inputs and outputs. This is the *subsumption architecture* [6].

BBAI has proved powerful because of the robustness and simplicity of the programs. Each behavior module is straight-forward enough to program reliably. BBAI is also strongly associated with *reactive intelligence*, because the subsumption architecture was both behavior-based and reactive. Reactive intelligence operates with no deliberation or search, thus eliciting the good response times critical for the successful operation of real-time systems such as robots or interactive virtual reality [34].

The cost of reactive intelligence is engineering. The agent's intelligence must be designed by hand since it performs no search (including learning or planning) itself. Although there have been some efforts made to learn or evolve BBAI programs offline [e.g. 23, 33], these efforts themselves take immense amounts of design, and have not proved superior to hand-designed systems [32, 36]. Although behavior modules are themselves definitionally easy to create, engineering the interactions *between* behaviors

has proved difficult. Some authors have taken the approach of limiting behaviors to representations that are easily combined [e.g. 1, 24], but this in turn limits the complexity of the agent that can be created by limiting its most powerful building blocks.

2.2 Reactive Plans and Three-Layer Architectures

At roughly the same time as BBAI was emerging, so were reactive plans [16, 20]. Reactive plans are powerful plan representations that provide for robust execution. A single plan will work under many different contingencies given a sufficiently amenable context. An agent can store a number of such plans in a library, then use context-based preconditions to select one plan that should meet its current goals in the current environment. The alternative — constructing a plan on demand — is a form of search and consequently costly [12]. In keeping with the goal of all reactive intelligence, reactive plans provide a way to avoid search during real-time execution.

A hybrid behavior-based system takes advantage of behaviors to give a planning system very powerful primitives. This in turn allows the plan to be relatively high-level and simple, a benefit to conventional planners as well as reactive plans [28]. Most three-layer hybrid architectures (as described above) have a bottom layer of behaviors, which serve as primitives to a second layer of reactive plans. They may then optionally has a third 'deliberative' (searching) layer either to create new plans or to choose between existing ones.

Consider this description of the ontology underlying three-layered architectures:

> The three-layer architecture arises from the empirical observation that effective algorithms for controlling mobile robots tend to fall into three distinct categories:
>
> 1. reactive control algorithms which map sensors directly onto actuators with little or no internal state;
> 2. algorithms for governing routine sequences of activity which rely extensively on internal state but perform no search; and
> 3. time-consuming (relative to the rate of change of the environment) search-based algorithms such as planners.
>
> Gat [19, p. 209]

In this description, behaviors are the simple stateless algorithms and reactive plans serve as state- or context-keeping devices for ordering the activity of the behaviors. In Gat's own architecture, ATLANTIS, [18] the second, reactive-plan layer dominates the agent: it monitors the agent's goals and selects its actions. If the second layer becomes stuck or uncertain, it can reduce the agent's activity while consulting a third-level planner, while still monitoring the environment for indications of newer, more urgent goals.

3 Behavior-Oriented Design (BOD)

Despite the development of three-layer hybrid architectures, programming CCA is still hard. Consequently, more approaches are being tried, including using MAS as architectures for single CCA [e.g. 35, 37]. I believe that the problem with three-layer architectures is that they tend to trivialise the behavior modules. The primary advantage of

the behavior-based approach and modular approaches in general is that they simplify coding the agent by allowing the programming task to be decomposed. Unfortunately, the drive to simplify the planning aspects of module coordination has lead many developers to tightly constrain what a behavior module can look like [24, fip]. Consequently, the engineering problem has become hard again.

To address these issues, I have developed Behavior-Oriented Design (BOD) [8, 11]. BOD has two major engineering advantages over three-layer architectures:

1. Perception, action, learning and memory are *all* encapsulated in behavior modules. They are expressed in standard programming languages, using object-oriented design (OOD) techniques.
2. The reactive plan structures used to arbitrate between these behavior modules are also designed to be easily engineered by programmers familiar with conventional programming languages.

One way to look at BOD is that the behaviors are used to determine *how* an agent acts, while the plans are used to largely determine *when* those actions are expressed.

At first glance, it may appear that BOD differs from the three-layer approach in that it has dropped the top, deliberative layer, but this is not the case. The real difference between my approach and Gat's is the loss of the bottom layer of purely reactive modules. I don't believe that there *are* many elements of intelligence that don't require some state. Perception requires memory: everything from very recent sensory information, which can help disambiguate a sight or sound, to life-long learning, which can establish expectations in the form of semantic knowledge about word meanings or a building's layout.

Under BOD, the behaviors are semi-autonomous modules, optionally with their own processes. A behavior can store sensory experience and apply complex processes of learning or deliberation across them. Or it can simply encapsulate clever algorithms for performing tasks. Nevertheless, a BOD system is reactive. This is both because the individual behaviors can respond immediately to salient sensory information, and because arbitration between modules is controlled by reactive plans. The primitive elements of the reactive plans are *an interface* to the behaviors; they are implemented as methods on the objects that encode the behaviors. These methods should produce immediate (or at least very rapid) results. Reactive plan primitives require a behavior to provide an anytime response [14] on demand, but the behaviors are free to perform the sorts of longer-term computations described by Gat continuously in the background. Thus all of an agent's behaviors (including behavior arbitration) can run continuously in parallel. Only when actions are expressed externally to the agent are they likely to be subject to action selection through behavior arbitration. Action selection is forced by competition for resources, such as the location of visual attention or the position of the agent's body [4].

4 Building an Agent with Good BOD

Behavior-Oriented Design is not just an architecture, but a design methodology, partially inspired by Object-Oriented Design (OOD) [e.g. 26, 31]. The analogy between

BOD and OOD is not limited to the metaphor of the behavior and the object, nor to the use of methods on the behavior objects as primitives to the reactive plans. The most critical aspect of BOD is its emphasis on the design process itself.

The fundamental problem of using a modular approach is deciding what belongs in a module — how many modules should there be, how powerful should they be, and so on. In BBAI this problem is called *behavior decomposition*; obviously analogous problems exist for OOD and MAS. BOD adopts the current accepted OOD practise for solving object decomposition: it focuses on the agent's adaptive state requirements, then uses iterative design and a set of heuristics to refine the original decomposition.

BOD emphasizes cyclic design with rapid prototyping. The process of developing an agent alternates between developing libraries of behaviors and the reactive plans to control the expression of those behaviors, and the process of clarifying and simplifying the agent by re-examining its behavior decomposition. The following sections explain the BOD guidelines for both the initial decomposition and for recognizing and correcting problems in the decomposition during the development process.

4.1 The Initial Decomposition

The initial decomposition is a set of steps. Executing them correctly is not critical, since the main development strategy includes correcting assumptions from this stage of the process. Nevertheless, good work at this stage greatly facilitates the rest of the process.

1. Specify at a high level what the agent is intended to do.
2. Describe likely activities in terms of sequences of actions. These sequences are the the basis of the initial reactive plans.
3. Identify an initial list of sensory and action primitives from the previous list of actions.
4. Identify the state necessary to enable the described primitives and drives. Cluster related state elements and their dependent primitives into specifications for behaviors. This is the basis of the behavior library.
5. Identify and prioritize goals or drives that the agent may need to attend to. This describes the initial roots for the reactive plan hierarchy (described below).
6. Select a first behavior to implement.

The lists compiled during this process should be kept, since they are an important part of the documentation of the agent. The process of documenting BOD agents is described below in Section 8.

4.2 Iterative Development

The heart of the BOD methodology is an iterative development process:

1. Select a part of the specification to implement next.
2. Extend the agent with that implementation:
   - code behaviors and reactive plans, and
   - test and debug that code.

3. Revise the current specification.

BOD's iterative development cycle can be thought of as sort of a hand-cranked version of the Expectation Maximization (EM) algorithm [15]. The first step is to elaborate the current model, then the second is to revise the model to find the new optimum representation. Of course, regardless of the optimizing process, the agent will continue to grow in complexity. But if that growth is carefully monitored, guided and pruned, then the resulting agent will be more elegant, easier to maintain, and easier to further adapt.

Unlike behaviors, which are simply coded directly in a standard object-oriented language, reactive plans are stored in script files. The plan is normally read when the agent is initialized, or "comes to life," though in theory new plans could be added during execution. The reactive plans for an agent grow in complexity over the course of development. Also, multiple reactive plans may be developed for a single AI platform (and set of behavior modules), each creating agents with different overall characteristics, such as goals or personality.

Even when there are radically different plan scripts for the same platform or domain, there will generally only be one behavior library — one set of code. Each agent will have its own instance or instances of behavior objects when it is running, and may potentially save run-time state in its own persistent object storage. But it is worth making an effort to support all scripts for a single platform or domain in a single library of behavior code.

Testing should be done as frequently as possible. Using languages that do not require compiling or strong typing, such as lisp or perl, significantly speeds the development process, though they may slow program execution time. "Optimize later", one of the modern mantras of software engineering, applies to programming languages too. In my experience, the time spent developing an AI agent generally far outweighs the time spent watching the agent run. Particularly for interactive real-time agents like robots and VR characters, the bottle-necks are much more likely to be caused by motor constraints or speech-recognition than by the intelligent control architecture.

The most interesting part of BOD's iterative design cycle is the set of rules for revising the specifications. However, understanding these rules requires understanding BOD reactive plans. The following section explains the details of BOD action selection. Section 6 returns to the question of knowing exactly how to optimize the agent.

5 BOD Reactive Plans

Reactive plans support action selection. At any given time step, most agents have a number of actions which could potentially be expressed, at least some of which cannot be expressed simultaneously, for example sitting and walking. In architectures without centralized action selection, such as the Subsumption Architecture [6] or the Agent Network Architecture (ANA) [27], the developer must fully characterize *for each action* how to determine when it should be expressed. This task grows in complexity with the number of new behaviors. For engineers, it is generally easier to describe the desired behavior in terms of sequences of events.

Of course, action-selection sequences can seldom be specified precisely in advance, due to the non-determinism of environments, including the unreliability of the agent's

own sensing or actuation. Several types of events may interrupt the completion of an intended action sequence. These events fall into two categories:

1. some combination of alarms, requests or opportunities may make pursuing a different plan more relevant, and
2. some combination of opportunities or difficulties may require the current 'sequence' to be reordered.

Thus the problems of action selection can be broken into three categories: things that need to be checked regularly, things that only need to be checked in a particular context, and things that do not strictly need to be checked at all.

BOD uses reactive plans to perform action selection through behavior arbitration. Individual behavior modules should be simple enough to be programmed to only recommend one action at any particular instant. BOD reactive plans provide three types of plan elements corresponding (respectively) to the three categories of action selection above: drive collections, competences, and action patterns.

The rest of this section explains these three types of elements, in reverse of the above order. BOD reactive plans are described more formally elsewhere [7, 10, 11]. This section gives a quick, informal introduction through examples.

5.1 Action Patterns: Some Things Always Follow

The *action pattern* is a simple sequence of primitives. Primitives are either actions or sensory predicates, and supported directly by the behaviors. Including the sequence as an element type is useful for two reasons. First, it allows an agent designer to keep the system as simple as possible, which both makes it more likely to succeed and communicates more clearly to a subsequent designer the expected behavior of that plan segment. Second, it allows for speed optimization of elements that are reliably run in order, which can be particularly useful in sequences of preconditions or in fine motor control. Here's an example that might be useful to an artificial monkey:

$$\langle \text{get a banana} \rightarrow \text{peel a banana} \rightarrow \text{eat a banana} \rangle \tag{1}$$

5.2 Competences: Some Things Depend on Context

A sequence is *not* equivalent to the process of *chaining* a set of productions, where each element's precondition is set to the fire its action as a consequence of the outcome of the prior element. Besides the possibility of optimizing away the step of checking preconditions, a sequence includes an additional piece of control state. Its elements may also occur in different orders in other contexts, and there is no ambiguity if more than one sequence's element fires in a perceptually equivalent context.

The advantage of productions of course is that, within the confines of a particular context, they allow for flexible behavior. A *competence* combines the advantages of both productions and sequences. Here is an example of the above sequence rewritten as a competence:

$$\text{(have hunger)} \Rightarrow \left\uparrow \left\langle \begin{array}{r} \text{(full)} \Rightarrow \textit{goal} \\ \text{(have a peeled banana)} \Rightarrow \text{eat a banana} \\ \text{(have a banana)} \Rightarrow \text{peel a banana} \\ \Rightarrow \text{get a banana} \end{array} \right\rangle \right. \tag{2}$$

Rather than encoding a temporal ordering, a competence encodes a *prioritization*. Priority increases in the direction of the vertical arrow on the left. Under this plan, if the monkey is handed a peeled banana, she'll eat it. If she has a whole banana, she'll peel it. Otherwise, she'll try to get a banana. When the goal has been achieved, or if none of the elements can fire, the competence terminates.

This sort of structure could lead to looping, for example if another, larger monkey kept taking the banana away after our agent had peeled it but before she could eat it. To allow termination in this circumstance, competence elements not only have priorities and preconditions, but also (optional) retry limits. When an element has reached its retry limit, it will no longer fire.

5.3 Drive Collections: Some Things Need to be Checked at All Times

Finally, there must be a way to arbitrate between plan elements or goals. There must be a way to determine the current focus of action-selection attention — to deal with context changes (whether environmental or internal) which require changing between plans, rather than within them. Some hybrid architectures consider this problem the domain of 'deliberation' or 'introspection' — the highest level of a three-layered architecture. But BOD treats this problem as continuous with the general problem of action selection, both in terms of constraints, such as the need for reactiveness, and of solution.

BOD uses a third element type, the *drive collection* for this kind of attention. A drive collection is very similar to a competence. However, it is designed never to terminate — there is no goal condition. The drive collection is the root of the BOD plan hierarchy, and the only element that executes on every program cycle (from hundreds to thousands of times a second, depending on the implementation). The highest priority drive-collection element that triggers passes activation to whatever competence, sequence or action primitive it is currently attending to. Or, if the agent has just been initialized or the element's last attendee has terminated, the element sets its attention to the apex of its plan hierarchy.

$$\textit{life} \Rightarrow \left\langle \left\langle \begin{array}{r} \text{(something looming)} \Rightarrow \text{avoid} \\ \text{(something loud)} \Rightarrow \text{attend to threat} \\ \text{(hungry)} \Rightarrow \text{forage} \\ \Rightarrow \text{lounge around} \end{array} \right\rangle \right\rangle \tag{3}$$

Drive-collection elements have another feature not shown in the above diagram: they also allow for scheduling, so that a high priority element does not necessarily monopolize program cycles. This increases the parallelism and reactivity of BOD agents.

For working, non-toy examples of plans for complete BOD agents, see [7–9]; for comparisons to other systems, see [7, 8]. This chapter emphasizes instead the engineering of BOD agents. The next sections return to the question of revising BOD specifications.

6 Revising BOD Specifications

A critical part of the BOD methodology is the set of rules for revising the specifications. The fundamental design principle is *when in doubt, favor simplicity.* A primitive is preferred to an action sequence, a sequence to a competence. Similarly, control state is preferred to learned state, specialized learning to general purpose learning or planning. Given this bias, heuristics are then used to indicate when a simple element should be exchanged for a more complex one.

A guiding principle in all software engineering is to reduce redundancy. If a particular plan or behavior can be reused, it should be. As in OOD, if only part of a plan or a primitive action can be used, then a change in decomposition is called for. In the case of the action primitive, the primitive should be decomposed into two or more primitives, and the original action replaced by a plan element, probably an action pattern. The new plan element should have the same name and functionality as the original action. This allows established plans to continue operating with only minimal change.

If a sequence sometimes needs to contain a cycle, or often does not need some of its elements to fire, then it is really a competence, not an action pattern. If a competence is actually deterministic, if it nearly always actually executes a fixed path through its elements, then it should be simplified into a sequence.

Competences are really the basic level of operation for reactive plans, and learning to write and debug them may take time. Here are two indications provided by competences that the specification of an agent needs to be redesigned:

- *Complex Triggers*: reactive plan elements should not require long or complex triggers. Perception should be handled at the behavior level; it should be a skill. Thus a large number of triggers may indicate the requirement for a new behavior or a new method on an existing behavior to appropriately categorize the context for firing the competence elements. Whether a new behavior or simply a new method is called for is determined by whether or not more state is needed to make that categorization: new state generally implies a new behavior.
- *Too Many Elements*: Competences usually need no more than 5 to 7 elements, they may contain fewer. Sometimes competences get cluttered (and triggers complicated) because they actually contain two different solutions to the same problem. In this case, the competence should be split. If the two paths lead all the way to the goal, then the competence is really two siblings which should be discriminated between at the level of the current competence's parent. If the dual pathway is only for part of the competence, then the original competence should contain two children.

Effectively every step of the competence but the highest priority one is a subgoal. If there is more than one way to achieve that subgoal, trying to express both of them in the same competence can split attention resources and lead to dithering or 'trigger-flipping' (where two plan elements serve only to activate each other's precondition). The purpose of a competence is to focus attention on *one* solution at a time.

The heart of the BOD strategy is rapid prototyping. If one approach is too much trouble or is giving debugging problems, try another. It is important to remember that programmers' experiences are the key selective pressures in BOD for keeping the agent

simple. BOD provides at least two paths to simplicity and clarity: modularity and hierarchical reactive plans. Using cyclic development and some trial and error, the programmer should determine which path is best for a particular problem [5, 30]. This is also why modularity and maintainability are key to BOD: programmers are to be encouraged to change the architecture of an agent when they find a better solution. Such changes should be easy to make. Further, they should be transparent, or at least easy to follow and understand, if another programmer encounters them.

Further heuristics, particularly with respect to prioritization and scheduling in drive collections can be found elsewhere [8]. In the next section, I will instead give an example of the most fundamental revision, trading off complexity in plans for that in behaviors.

7 Trading Off Control and Learning

To demonstrate the differences between representational styles, let's think about an insect robot with two 'feelers' (bump sensors), but no other way of sensing its environment.

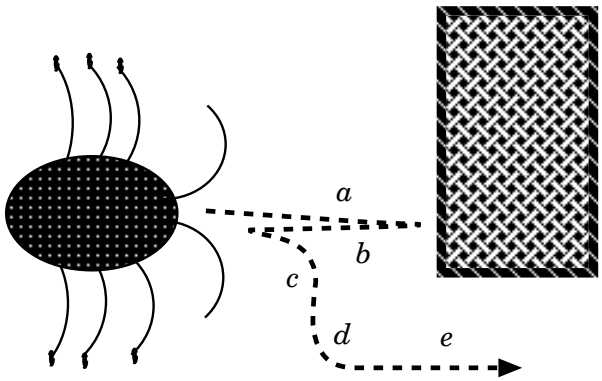


Fig. 1. An insect-like robot with no long-range sensors (e.g. eyes) needs to use its feelers to find its way around a box.

7.1 Control State Only

This plan is in the notation introduced earlier, except that words that reference parts of control state rather than primitives are in bold face. Assume that the 'walk' primitives take some time (say 5 seconds) and move the insect a couple of centimeters on the diagram. Also, assume turning traces an arc rather than happening in place. This is about the simplest program that can be written using entirely control state:

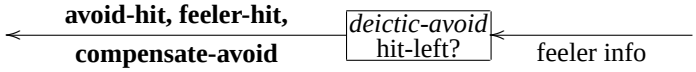
$$\textbf{walk} \Rightarrow \uparrow\left\langle \begin{array}{r} \text{(left-feeler-hit)} \Rightarrow \textbf{avoid-obstacle-left} \\ \text{(right-feeler-hit)} \Rightarrow \textbf{avoid-obstacle-right} \\ \Rightarrow \text{walk-straight} \end{array} \right\rangle \quad (4)$$

$$\textbf{avoid-obstacle-left} \Rightarrow \langle \text{walk backwards} \rightarrow \text{walk right} \rightarrow \text{walk left} \rangle \quad (5)$$

$$\textbf{avoid-obstacle-right} \Rightarrow \langle \text{walk backwards} \rightarrow \text{walk left} \rightarrow \text{walk right} \rangle \quad (6)$$

7.2 Deictic State as Well

If we are willing to include a behavior with just one bit of variable state in it, then we can simplify the control state for the program. In the behavior *deictic-avoid*, the bit hit-left? serves as a deictic variable the-side-I-just-hit-on. **Avoid-hit** and **compensate-avoid** (primitives *deictic-avoid* supports) turn in the appropriate direction by accessing this variable. This allows a reduction in redundancy in the plan, including the elimination of one of the action patterns.

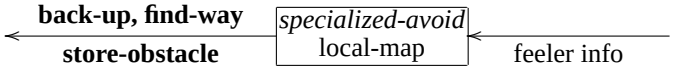


$$\textbf{walk} \Rightarrow \uparrow\left\langle \begin{array}{r} \text{(feeler-hit)} \Rightarrow \textbf{avoid-obstacle} \\ \Rightarrow \text{walk-straight} \end{array} \right\rangle \quad (7)$$

$$\textbf{avoid-obstacle} \Rightarrow \langle \text{walk backwards} \rightarrow \text{avoid hit} \rightarrow \text{compensate avoid} \rangle \quad (8)$$

7.3 Specialized Instead of Deictic State

Instead of using a simple reference, we could also use a more complicated representation, say an allocentric representation of where the obstacle is relative to the bug, that is updated automatically as the bug moves and forgotten as the bug moves away from the location of the impact. Since this strategy requires the state to be updated continuously as the bug moves, walking must be a method (**find-way**) on this behavior.



$$\textbf{walk} \Rightarrow \uparrow\left\langle \begin{array}{r} \text{(feeler-hit)} \Rightarrow \textbf{react-to-bump} \\ \Rightarrow \text{find-way} \end{array} \right\rangle \quad (9)$$

$$\textbf{react-to-bump} \Rightarrow \langle \text{store-obstacle} \rightarrow \text{walk backwards} \rangle \quad (10)$$

If this is really the only navigation ability our bug has, then the vast increase in complexity of the behavior *specialized-avoid* does not justify the savings in control state. On the other hand, if our bug already has some kind of allocentric representation, then it might be sensible to piggy-back the feeler information on top of it. For example, if the bug has a vector created by a multi-faceted eye representing approximate distance to visible obstacles, but has bumped into something hard to see (like a window), it might be parsimonious to store the bump information in the vision vector, providing that updating the information with the bug's own motion isn't too much trouble. Insects actually seem able to do something like this [21], and I've done it with a robot [9].

8 Documenting a BOD Agent

As I said at the end of Section 6, maintainability and clarity of design are key to the BOD development process. This is best acheived through self-documenting code. "Self-documenting" is something of a misnomer, because of course the process takes discipline. The primary argument for incorporating documentation into functioning code is that this is the only way to ensure that the documentation will never get out of synchronization with the rest of the software project. The primary argument against this strategy is that code is never really that easy to read, and will never be concisely summarized. BOD at least somewhat overcomes this problem by having two types of summary built into the agent's software architecture. The reactive plans summarize the aims and objectives of the agent, and the plan-primitive / behavior interface documents at a high level the expressed actions of the various behaviors. Further information, such as documentation on the adaptive state used by the agent, can be found in the code of the behavior modules.

8.1 Guidelines for Maintaining Self-Documentation

The following are guidelines to the discipline of making sure BOD agents are self-documented:

Document the plan / behavior interface in one program file. As explained earlier, the primitives of the POSH reactive plans must be defined in terms of methods on the behavior objects. For each behavior library, there should be one code file that creates this interface. In my implementations of POSH action selection, each primitive must be wrapped in an object which is either an *act* or a *sense*. The code executed when that object is triggered is usually only one or two lines long, typically a method call on some behavior object. I cluster the primitives by the behaviors that support them, and use program comments to make the divisions between behaviors clear.

This is the main documentation for the specification — it is the only file likely to have both current *and intended* specifications listed. This is where I list the names of behaviors and primitives determined during decomposition, even before they have been implemented. Intended reactive plans are usually written as scripts (see below.)

Each behavior should have its own program file. Every behavior will be well commented automatically if it is really implemented as an object. One can easily see the state and representations in the class definition. Even in languages that don't require methods to be defined in the class declaration, it is only good style to include all the methods of a class in the same source file with the class definition.

Keep and comment reactive plan scripts. This is the suggestion that requires the most discipline, but having a documented history of the development of an agent can be critical to understanding some of its nuances. Documenting plan scripts effectively documents the history of the agent's development — it is easy to determine when behaviors were added or modified by seeing what primitives are present in a script. Those who can't remember history are doomed to repeat old mistakes. Keeping a complete set of working scripts documenting stages of the agents development also provides a test suite, useful when major changes are made to behavior libraries.

Script comments should contain:

– Its name, to flag if it has been copied and changed without updating the comment.
– What script(s) it was derived from. Most scripts are improvements of older working scripts, though some are shortened versions of a script that needs to be debugged.
– The date it was created.
– The date it started working, if significantly different. Since writing scripts is part of the specification process, some scripts will be ambitious plans for the future rather than working code.
– The date and reasons it was abandoned, if it was abandoned.
– Possibly, dates and explanations of any changes. Normally, changes shouldn't happen in a script once it works (or is abandoned) — they should be made in new scripts, and the old ones kept for a record.

8.2 BOD and Code Reuse

One aspect of OOD that I have not yet explicitely touched on is the issue of code reuse. Code reuse is of course one of the main purposes of both modularity and good documentation. We can think of CCA code as breaking into three categories:

1. *Code useful to all agents.* For BOD, this code is primarily embedded in the POSH planning system. POSH reactive plans are the one set of patterns I believe to be generally useful to modular agents, but not to modular programs which do not have agency.
2. *Code useful to most agents on a particular platform or domain.* By the nature of the BOD methodology, most of this code will wind up in the library of behaviors. However, because this is not the main criteria for determining whether code is in a plan or a behavior, it is possible that some aspects of the reactive plans will also be shared between all agents inhabiting a single platform. Nevertheless, behavior modules can and should be treated exactly as a code library.
3. *Code useful only to a single agent.* Because much of what determines one individual from another is their goals and priorities, the reactive plans will generally succinctly specify individual differences. Though again, it is possible that some behaviors in the behavior library will also only be expressed / referenced by a single agent. Much of what is unique to an agent will not be its program code, but the state within its behavior modules while it is running — this is what encodes its experience and memories.

9 Conclusions

Excellent software engineering is key to developing complete, complex agents. The advances of the last fifteen years in CCA due to the reactive and behavior-based movements come primarily from two engineering-related sources:

1. the trade-off of slow or unreliable on-line processes of search and learning for the one-time cost of development, and
2. the use of modularity.

Of course, the techniques of learning and planning cannot and should not be abandoned: some things can only be determined by an agent at run time. However, constraining learning with specialized representations and constraining planning searches to likely solution spaces greatly increases the probability that an agent can reliably perform successfully. Providing these representations and solution spaces is the job of software engineers, and as such should exploit the progress made in the art of software engineering.

In this chapter I have described the recent state-of-the-art in CCA design, and then described an improvement, the Behavior-Oriented Design methodology. This methodology brings CCA development closer to conventional software engineering, particularly OOD, but with the addition of a collection of organizational idioms and development heuristics that are uniquely important to developing AI.

References

[fip] FIPA-OS: A component-based toolkit enabling rapid development of fipa compliant agents. http://fipa-os.sourceforge.net.

[1] Arkin, R. C. (1998). *Behavior-Based Robotics*. MIT Press, Cambridge, MA.

[2] Ballin, D. (2000). Special issue: Intelligent virtual agents. *Virtual Reality*, 5(2).

[3] Bertolini, D., Busetta, P., Molani, A., Nori, M., and Perini, A. (2002). Designing peer-to-peer applications: an agent-oriented approach. In this volume.

[4] Blumberg, B. M. (1996). *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, MIT. Media Laboratory, Learning and Common Sense Section.

[5] Boehm, B. W. (1986). A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):22–32.

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

[7] Bryson, J. J. (2000). Hierarchy and sequence vs. full parallelism in reactive action selection architectures. In *From Animals to Animats 6 (SAB00)*, pages 147–156, Cambridge, MA. MIT Press.

[8] Bryson, J. J. (2001). *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, Department of EECS, Cambridge, MA. AI Technical Report 2001-003.

[9] Bryson, J. J. and McGonigle, B. (1998). Agent architecture as object oriented design. In Singh, M. P., Rao, A. S., and Wooldridge, M. J., editors, *The Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL97)*, pages 15–30. Springer-Verlag.

[10] Bryson, J. J. and Stein, L. A. (2001a). Architectures and idioms: Making progress in agent design. In Castelfranchi, C. and Lespérance, Y., editors, *The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL2000)*. Springer.

[11] Bryson, J. J. and Stein, L. A. (2001b). Modularity and design in reactive intelligence. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 1115–1120, Seattle. Morgan Kaufmann.

[12] Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378.

[13] Chomsky, N. (1980). Rules and representations. *Brain and Behavioral Sciences*, 3:1–61.

[14] Dean, T. and Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 49–54, Saint Paul, Minnesota, USA. AAAI Press/MIT Press.

[15] Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society series B*, 39:1–38.

[16] Firby, J. (1987). An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 202–207.

[17] Fodor, J. A. (1983). *The Modularity of Mind*. Bradford Books. MIT Press, Cambridge, MA.

[18] Gat, E. (1991). *Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots*. PhD thesis, Virginia Polytechnic Institute and State University.

[19] Gat, E. (1998). Three-layer architectures. In Kortenkamp, D., Bonasso, R. P., and Murphy, R., editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 195–210. MIT Press, Cambridge, MA.

[20] Georgeff, M. P. and Lansky, A. L. (1987). Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA.

[21] Hartmann, G. and Wehner, R. (1995). The ant's path integration system: A neural architecture. *Bilogical Cybernetics*, 73:483–497.

[22] Hexmoor, H., Horswill, I., and Kortenkamp, D. (1997). Special issue: Software architectures for hardware agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3).

[23] Humphrys, M. (1997). *Action Selection methods using Reinforcement Learning*. PhD thesis, University of Cambridge.

[24] Konolige, K. and Myers, K. (1998). The Saphira architecture for autonomous mobile robots. In Kortenkamp, D., Bonasso, R. P., and Murphy, R., editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter 9, pages 211–242. MIT Press, Cambridge, MA.

[25] Kortenkamp, D., Bonasso, R. P., and Murphy, R., editors (1998). *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. MIT Press, Cambridge, MA.

[26] Larman, C. (2001). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, 2^{nd} edition.

[27] Maes, P. (1990). Situated agents can have goals. In Maes, P., editor, *Designing Autonomous Agents : Theory and Practice from Biology to Engineering and back*, pages 49–70. MIT Press, Cambridge, MA.

[28] Malcolm, C. and Smithers, T. (1990). Symbol grounding via a hybrid architecture in an autonomous assembly system. In Maes, P., editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 123–144. MIT Press, Cambridge, MA.

[29] Minsky, M. (1985). *The Society of Mind*. Simon and Schuster Inc., New York, NY.

[30] Parnas, D. L. and Clements, P. C. (1986). A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–7.

[31] Parnas, D. L., Clements, P. C., and Weiss, D. M. (1985). The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266.

[32] Pauls, J. (2001). Pigs and people. *in preperation*.

[33] Perkins, S. (1998). *Incremental Acquisition of Complex Visual Behaviour using Genetic Programming and Shaping*. PhD thesis, University of Edinburgh. Department of Artificial Intelligence.

[34] Sengers, P. (1998). Do the thing right: An architecture for action expression. In Sycara, K. P. and Wooldridge, M., editors, *Proceedings of the Second International Conference on Autonomous Agents*, pages 24–31. ACM Press.

[35] Sierra, C., de Màntaras, R. L., and Busquets, D. (2001). Multiagent bidding mechanisms for robot qualitative navigation. In Castelfranchi, C. and Lespérance, Y., editors, *The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL2000)*. Springer.

[36] Tyrrell, T. (1993). *Computational Mechanisms for Action Selection*. PhD thesis, University of Edinburgh. Centre for Cognitive Science.

[37] van Breemen, A. (2002). Integrating agents in software applications. In this volume.