Agent Architecture as Object Oriented Design

Joanna Bryson and Brendan McGonigle

Laboratory for Cognitive Neuroscience and Intelligent Systems, Edinburgh University
Edinburgh EH8 9JZ, UK

Joanna.Bryson@ed.ac.uk

Abstract. Improving the development of agent intelligence requires improving the mechanisms of that development. This paper explores the application of an established software methodology, object-oriented design, to agent development in two ways. We present a distributed agent architecture, Edmund, and describe first its own object-oriented structure. Then we relate the methodology for developing agent behaviors under Edmund. We explain how this methodology exploits key aspects of object-oriented design, particularly the development of the class hierarchy, as a prototype for agent design.

1 Introduction

The history of computer science, like that of mathematics, has clearly demonstrated that tools and methodologies are not only useful but necessary for advancing through stages of complexity. Human problem solving seems to have finite limits in the level of complexity one mind can encompass. A powerful representation or language can reduce the cognitive load by encapsulating and condensing well understood elements of a problem. Similarly, a methodology is a tool that provides rules for decomposing a problem into manageable sections.

Methodology is particularly important in the design of reactive systems, because the reactive approach to intelligent agent research depends entirely on the hand-coding of the agent's behavior [36]. Our laboratory operates on the hypothesis that the most likely way to replicate the natural phenomena of intelligence is through careful design [22]. This implies that building on and expanding design methodologies is a necessary component of research into artificial intelligence.

This paper describes one of the architectures, Edmund, under development in our laboratory as a platform for cognitive modeling. Edmund was developed under object-oriented design (OOD), and the design methodology developed for using Edmund is similar to OOD. Because the agent community is examining OOD as a design tool, we present both Edmund's control architecture and accompanying methodology as an exemplar.

2 Edmund

Edmund can be thought of as an agent architecture in two senses. First, it is designed to run the intelligence of an autonomous mobile robot, an agent in the strong sense of the term. Second, it serves to structure a library of behaviors, which are themselves implemented as objects. To some extent, any behavior-based architecture could be described as a multi-agent system, but most constituent behaviors, though independent and executing in parallel, are too simple to describe as agents. Edmund's Nomad Robot Library (ENRL), however, is decomposed along lines not of behavior, but of capability. Thirty-two sensing and action primitives for the architecture are based on methods from five classes. Each class is based on a knowledge-set representing attributes of the external world or the robot's mechanism. These might be considered agents in the weaker sense of semi-autonomous software units with state, monitoring aspects of the world and responding to queries.

Edmund is the result of an attempt to scale the behavior-based artificial intelligence (BBAI) approach to building intelligent systems without resorting to hybrid symbolic / reactive architectures such as 3T [3] or PRS [13]. During development, Edmund has evolved commonalities with these architectures which we review briefly Section 6. It addresses three problem areas in BBAI, the first being control structure. Strict parallelism provides insufficient bias for action selection and goal persistence. Tyrrell demonstrates that this is particularly true in non-hierarchical machines such as Maes [20], but suggests it is even a problem with his own hierarchy-based spreading activation approach [32]. Animal intelligence appears to provide sequencing primitives for both representation and action [23, 31]. At the same time, rigidly connected structures as in Brooks' subsumption architecture [4] or even as constructed by Firby's RAPs [10] overly constrain behavioral flexibility. Subsumption architecture fixes goal prioritization in a linear order. RAPs constructs tree structures not suited to the continuous, non-terminating processes of life-like agents; processes including language [11]. Edmund's architecture addresses the selection bias problem by providing structures for both parallel and sequential control components. It addresses the flexibility issue by providing a parallel reactive drive structure, each element of which allows for chaining or looping sub-structures. Edmund's control is discussed in the architecture section below.

The second problem area we address is the need for specialized, modularized learning. In BBAI, learning tends to be either nonexistent, as in strictly reactive systems [4], or fairly homogeneous, with each behavior having a generic learning capability [28, 2]. Often it is monolithic, as in systems that employ reinforcement learning, genetic algorithms, or neural networks to supervise control. Vertebrate brains have specialization both by organs and regions within organs for storing relevant state, whether that state is fairly transient computations of perceptual or motor information or longer term skills and information [6]. BBAI systems with specialized learning or state elements (such as [21, 16]) have been successful, but no principled system for supporting multiple state elements with varying representations has been developed. Edmund addresses this with object-oriented behavior libraries, described in Section 4 below.

Finally, a crucial element to scaling is programmability. Since behavior-based architectures (and most actively used agent architectures) are essentially hand coded, their complexity is highly dependent on the methodological support for the software designers. We consider developing a methodology for programming agents under Edmund as important as developing its architecture. To facilitate development under Edmund we offer a development methodology based on the established methodology of object oriented design (OOD). Methodology is discussed in Section 5.

Sense	A perceptual primitive returning a value.
Act	A primitive affecting the external world or internal state, including perception.
Trigger	A sequence of primitives executed atomically, for selecting actions. Should have no impact external to agent.
Lap	An action pattern, a sequence of primitives which may have external impact. The *L* is for "learnable", not yet implemented.
Competence	A prioritized set of Laps or other Competences and their triggers. Includes a goal lap that, if successfully triggered, indicates completion.
Drive	A type of competence (may also contain Drives) that runs its components in parallel. Generally has no goal.

Fig. 1. The classes composing the Edmund architecture.

3 The Architecture

3.1 Elements

The Edmund architecture classes are as specified in Figure 1. The classes may be divided into two sets, primitives and collections. Primitive actions and senses are the interface to the behavior library code, which will be discussed in the next section. Collections provide bias through control structure for determining appropriate and coherent ordering of the primitives. This ordering of primitives into control structures via collective classes is specified in script files. Behavior libraries are specific to a platform, they provide the agent's fundamental behaviors and perceptions. Scripts specify specific tasks or persona. Figure 2 shows an example control structure for navigation on a symmetric mobile robot. Section 4.1 describes the behavior library.

The collective classes are described in detail below, but roughly their tasks are the following. Action sequences provide persistence and reduce the combinatorics of action selection by encoding skills in a canonical way. During execution they maintaining state specifying which element of a skill is currently being executed, and thus which element should be next. Competences are essentially reactive plans, much like triangle tables [25]. Drives are similar to competences but may have several elements in some state of execution at the same time, they provide scheduling and priority between parallel tasks for the robot. Like action sequences, competences and drives maintain attentional state while the program is executing.

Edmund was written under object-oriented design primarily to exploit inheritance and polymorphism: a collective class does not need to know the type of an element it contains, merely when to execute it. Edmund also extends the class/object idiom to the instances of its classes. There may, for example, be multiple occurances of a single primitive occuring in different control contexts, that is, in different LAPs. Within an action sequence, a sense primitive serves as a conditional to check whether the sequence should continue. Consequently, scripts always associate senses with tests, which are in turn stored in the sense instance when it is incorporated into a LAP. Different instances of the same sense may be associated with different tests in different laps.

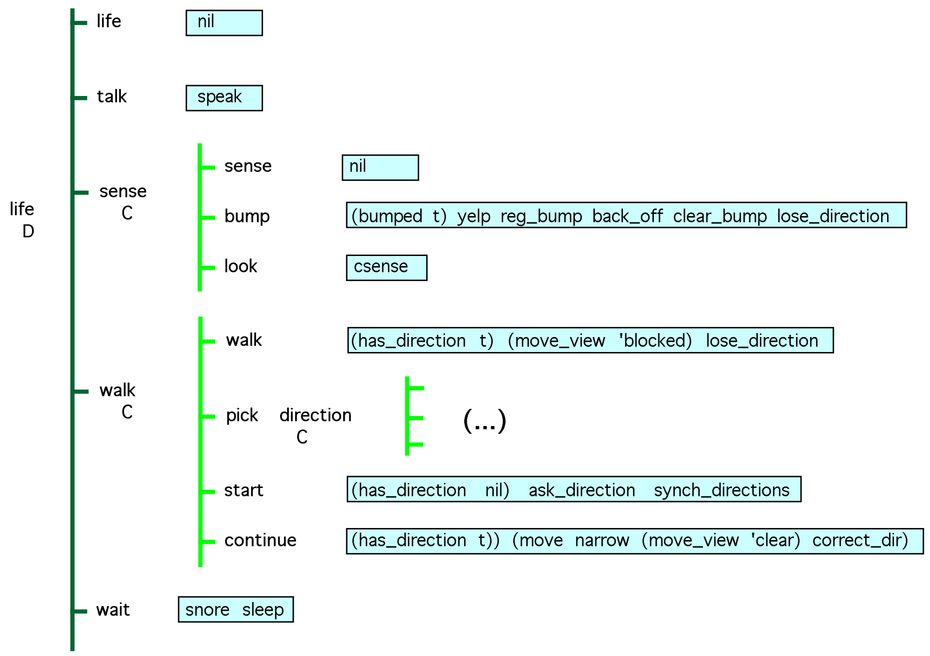

Fig. 2. A simple control heirarchy with no loops. This controls a navigating mobile robot. Priorities are not shown, but initial priority is ordered from top to bottom. Drives are labeled **D**, Competences **C**. Senses are paired with test values, unpaired primitives are Acts.

3.2 Reactivity

Edmund's reactiveness comes from two sources. First, on execution a competence checks the trigger of each of its elements to find the highest priority element that can run. No assumptions are made about the outcome of an action sequence; the environment is always resampled for selecting the next element. Second, drives allow the execution of multiple competences or actions in parallel. Although the program only attends to a single drive component at a time, each component maintains state while the other components are being attended to. Consequently, their execution proceeds as if in parallel, via coarse-grained time slicing. In the library described below, primitive acts often give commands to parallel hardware systems and do not wait for a result, so true parallelism from the perspective of the robot is achieved.

The continuous sampling of the environment through triggers results in a form of active perception. Competences that are currently active and have priority will be the ones executing their components' triggers, thus extra time and attention is given to seeking information relevant in the current context.

Which drive component receives attention is determined by priority level. The priorities can be "habituated"; scripts may specify a penalty per execution and a recovery rate per second. In drives, this allows for allocation of attention resources via best-effort scheduling. Competence elements may also habituate, though normally with no recovery. This allows for alternative solutions of similar priority to be tried, and for competences to guarantee failure after a certain number of attempts. Competences fail if none of their elements can be triggered.

When the top drive or competence has determined where its attention lies, that element is executed. Since drives and competences can be nested, an architecture like this might become highly unreactive due to a long chain, or worse, a loop of components calling each other. Edmund solves this problem by not fully preserving a stack. Competences don't attend to other competences, but rather shift the attention currently allocated to themselves. Assume a drive $D = (C_a, C_b, L_c, \ldots)$ where competence $C_a = (L_1, L_2, C_3, \ldots)$. If C_a's arbitration selects C_3 as the highest priority triggerable element, then C_a flags itself for replacement. The next time D fires, it contains $(C_3, C_b, L_c, \ldots)$. Thus the the activation tree is never deeper than the deepest drive plus three (a competence, a lap, and a primitive). In practice, we have yet to nest drives, and doubt that they should need to be nested more than two layers.

This chaining of attention guarantees reactivity, but might fail to provide coherent behavior or persistence. A competence that is an element of another's list might reasonably be described as a subgoal. Once a subgoal has completed, shouldn't its parent continue? Edmund addresses this issue in the drives. A drive not only keeps track of the processes to which it is currently attending, but also has a default process for each element. The default starts if that drive element is triggered with no other active process, for example, at start up. Thus if a competence completes, the drive returns attention to the root of the tree that brought it to that competence in the first place. If the context is the same, then attention will return to the parent competence. If the context has changed, some other behavior may now be more appropriate. A competence can terminate by failing to execute any of its elements or by succeeding to trigger its goal element.

3.3 Biological Plausibility

Edmund is designed to be a platform for running psychologically and ethologically based models of intelligence. We have attempted to make it neurologically plausible. Behavior sequences are supported in the natural intelligence literature as described in Section 2. Competences are not directly supported, but could exist as sets skills that are collectively stimulated to varying degrees in a particular context, similar to priming in spreading activation networks [24]. Similar action selection mechanisms have been postulated by [14].

Drives allow the simulation of parallel activity in non-cortical parts of the brain. For example areas of the forebrain such as the amygdala operate on separate channels of sensor input and can interrupt cortical activity by providing independent activation, while elements of the hindbrain maintain metabolic event scheduling independent of cortical sensing and activation [6].

The perpetuation of activation and its chaining through competences as described above is related to the problems of binding and attention, issues which are receiving a great deal of attention in neuroscience at the moment. See for example [33, 29].

Direction	For moving and detecting motion-relevant information, knows current and preferred directions. Instances correspond to faces of the robot.
PMem	Perceptual Memory, a short list of recent combined sensor readings. The main source of sensing. No significant instances.
Bump	Provide information about local "invisible" obstacles. Instances represent collision points.
Wave	Detect people waving at sonar (a source of communication).
DpLand	Decision point landmarks, the basis for navigating (vs. wandering.) Instances represent choice points and the decisions made there.

Fig. 3. The classes composing the Edmund's Nomad Robot Library (ENRL).

4 The Behavior Library

4.1 Structure

The functions and state on which the sensing and action primitives depend are located in platform-dependent behavior libraries. So far two libraries have been constructed, one for using visual routines theory in a blocks world simulating (as in [34, 7]), and the other for a Nomad 200 mobile robot. This paper focuses on Edmund's Nomad Robot Library (ENRL), which is the more recent work.

The Edmund architecture's only interface to its libraries are function pointers that are passed to its primitives on loading. A library routine might consist entirely of calls to underlying hardware or software routines provided by the agent's platform. The Nomad provides primitive actions and senses, but these are at a lower level of abstraction than is actually useful for the control scripts. The behaviors that *are* needed cluster naturally by shared resources and functions. Consequently, ENRL is also of object-oriented design, with the classes based around these clusters. See Figure 3 for examples.

One of the resources shared between behaviors is memory, or state. This can be anything from very short term ($<$ 1 second) perceptual memory, for things like "sanitizing" sonar readings, to long term memory, for tasks like map learning. Often different but related forms of state are stored as class and instance variables for a single class. For example, when an obstacle is encountered via the robot's bumpers, an instance of Bump may record its approximate coordinates, while a Bump class variable refers to the list of all currently remembered Bumps.

The primitives for detecting collisions (using bumper state information provided on the Nomad platform), storing instances (using similarly provided odometric information), recalling and pruning information from memory are all based on methods in the Bump class. The information (except for that provided directly by the Nomad) is not computed automatically, but only on demand when relevant primitives are called from the control structure. Nevertheless, as mentioned in Section 2, behavior library elements are designed to represent different areas of a modular (but not fully encapsulated) memory structure like the brain. This corresponds roughly to Dennett and Kinsbourne's [8] multiple drafts hypothesis of awareness . There is no single continuous stream or record

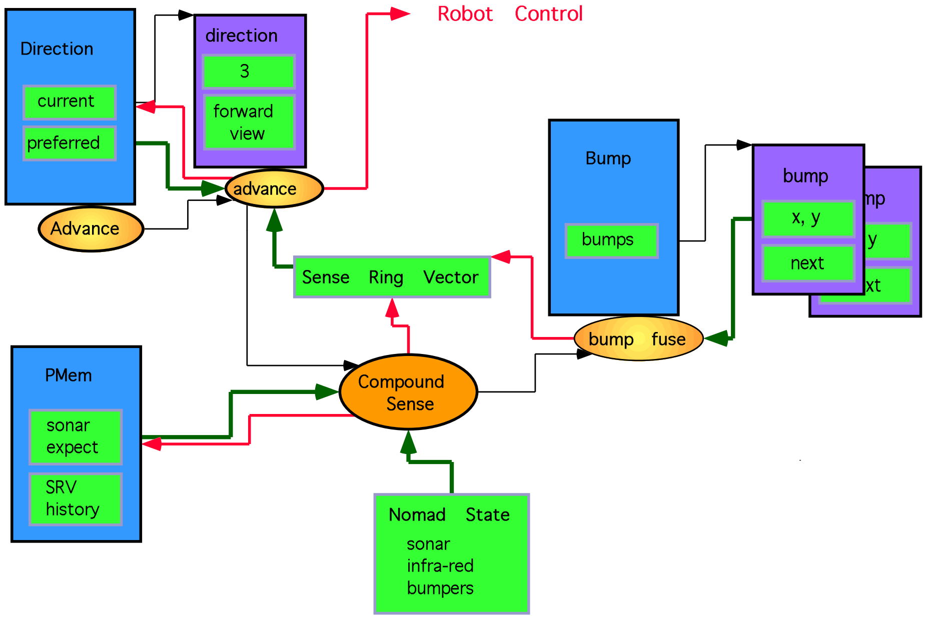

Fig. 4. The state supporting the Direction library function **advance**.

of information; what is perceived is in a continuous state of flux, and only resolves to a snapshot when attended to.

4.2 A Sample Behavior

This section describes one of the elements of ENRL, Direction, as an example of the nature the library. This example will also be referred to in the following section on methodology. Direction is largely responsible for the familiar task of low-level navigation and obstacle avoidance. The behaviors necessary for motion are based on its methods, though these in turn refer to information and methods from the perceptual classes. Figure 4 illustrates the classes involved in the Direction member function *advance*, and their flow of information. The behavior is described below.

There are 16 instances of Direction correlating to the 16 faces of the Nomad, each with a sonar and an infra-red sensor (see Figure 5). An Act primitive, **move**, calls *advance* on whichever instance of Direction the robot is currently pursuing. There were initially two subclasses of Direction to include an additional 16 directions on the axis between two faces, but in experiments these proved unnecessary for smooth trajectories and were removed to simplify the code.

Advance works as follows. The current Direction instance computes a new velocity for translation, based on the room it has to maneuver. It also computes a velocity for rotation, based on the difference between the robot's current steering angle θ_r, and the Direction's own orientation, θ_d. The formulae are based on power curves, making the

Fig. 5. The Nomad 200 robot on which ENRL has been developed. In this picture it is encountering a desk — the surface is too low for the sonar sensors to detect when proximate, and the leg too dark for the infra-red sensors, so the robot must learn about the desk with its bumpers.

robot's trajectories fluid and natural[1]. *Advance* directs the Nomad to move at these velocities, then finishes by choosing a new current direction if obstacle avoidance is called for.

Advance is an example of how an apparently continuous action can be sliced appropriately for an architecture based on discrete steps. Operating on the Nomad robot, ENRL results in smooth, continuous motion. In fact, high level drive attention is being switched between motion, sensing the external world, and checking and reporting battery level. Motion is also expressed as a competence, and that competence routinely

[1] The steering rate is $k_2 e^{-k_1 \frac{1}{\sqrt{\theta_c}}}$ where the k_1 (26) adjusts the slope of the exponential, k_2 (450) adjusts the scale of the result, and θ_c is the difference between θ_r and θ_d, adjusted for the nearest direction of rotation. The translation velocity is $k_2 e^{-k_1 \frac{1}{\sqrt{dist}}}$ where k_1 (5) is as above, k_2 (100) is the maximum velocity in tenths of an inch per second, and $dist$ is the nearest obstacle in the direction of motion also in tenths of an inch.

move	(A)	`Direction::current->advance()`
narrow	(A)	`if (Direction::current - Direction:: pref > 4) {lose-dir && fail}`
move-view	(S)	`Direction::current->state()`
correct-dir	(A)	`Direction::correct-to-pref()`

Fig. 6. The interface between the library and the control script. Left are the motion primitives in the action sequence **move** in Figure 2. Right is the corresponding code using member functions from Figure 4.

checks the triggers for starting and terminating motion before choosing the case of continuing motion, executing **move**. In Nomad's commercial simulation, the calls to *advance* do appear as discrete events, because the commands cannot be performed by separate hardware systems as they are on the real Nomad.

4.3 The Use of State

Advance requires a great deal of state and sensing capabilities which are reused in other primitive behaviors. To begin with, choice of speed and direction are determined by the perceived space surrounding the robot. This is represented by an array of 16 values representing the distance in inches from the robots faces to the nearest obstacle. This perceptual array is a construction regularly updated with information from sonar, infra-red and bump sensors. Information from the sonars is not taken directly, because sonars often give false readings. Sonar values are based on a perceptual memory of previous values and rates of change – sudden changes may not be believed for nearly half a second, then they just as suddenly replace the old value in the perceptual memory. This approach is consistent with human perception [8]. Bumps require a quite different, more persistent form of perceptual memory since, in order to be useful, they need to influence behavior by being avoided for some time, possibly even a minute, after impact. Consequently PMem sonar memory is stored in a fixed size array, while Bumps are stored in instances indexed off a class variable and trimmed as a special process.

In addition to these perceptual classes, Direction itself stores state as class variables recording which direction is currently being pursued, and which direction was actually intended or preferred. Another Act primitive, **correct-dir**, tries to move the current direction closer to the preferred one. This is typically called in scripts where the robot is to display persistence in direction, as part of an action sequence with **move** after a Sense primitive shows the robot is not avoiding any obstacles. Other state in Direction includes state in the instances that specifies a distorted window on the perceptual array mentioned above, for which elements are relevant (and how relevant) to forward motion. The Nomad can pass through doorways only a few inches wider than itself, but will stop four inches in front of most obstacles.

5 Methodology

Much of the variety of approach in autonomous agent architectures reflects variety in the underlying theories of intelligence. Ideally, our agents should provide us with platforms for testing and developing such theories. However, our only perspective for viewing and comparing the behavior of our agents superimposes our ability to program on top of our underlying theories. As a result, methodology is an inextricable part of intelligent agent research.

The thesis of this paper is that a good architecture should encourage and facilitate a particular methodological approach. In the previous section we described the implementation of Edmund, which exploited some aspects of object-oriented design to organize state and behavior into logical units. The implementation was explicitly in an object-oriented language, C++. This section describes the development of behavior under Edmund. This requires the co-evolution of the control scripts that specify *when* a behavior is expressed, and the behavior libraries, which describe *how* that behavior expresses itself. We now use object-oriented design (OOD) as an *analogy* — as a basis for Edmund's own methodology.

Behavior-based systems and OOD both exploit modularity and encapsulation to manage complexity. The following subsection discusses how that modularity is established. Other issues of methodology appear at the end of the section.

5.1 Class Hierarchies and Behavior Decomposition

One of the key problems of behavior-based design is *behavior decomposition*, the problem of determining what the fundamental elements of behavior are. Because Edmund separates the flow of control state from that of perception, the decomposition problem is also redescribed. The designer needs to determine whether a behavior should be encoded as a competence, a sequence, or as a primitive supported in the behavior library.

The methodology we recommend is iterative, similar to that of OOD. One of the main principles of OOD is rapid prototyping. The design process is first boot-strapped by a combination of analysis and intuition. The process described in [27] for multi-agent systems would be adaptable for both object-oriented systems and to some extent Edmund. Once an initial decomposition has been reached, the class hierarchy is evolved as the classes of objects are developed and tested.

In OOD, the importance of the hierarchy is to simplify development and reduce replication of code by having basic classes defined whose instances would have the standard requirements of functionality. For objects that require more specialization, subclasses are created that add functionality or attributes. There are standard situations that indicate when a hierarchy needs to be adjusted. Two classes that replicate code indicate the need for the creation of a common super class. If some object needs half the functionality of a class then this indicates a need for dividing a class into a parent and child, so that the object may inherit only from the parent. A parent class with a single child is an opportunity to collapse structure unless instances of the parent actively use methods overwritten in the child. Too much multiple inheritance, the frequent need to inherit from several classes from different parts of the hierarchy, may indicate bad fundamental hierarchy decomposition and a need to start over.

We take as given that rapid prototyping and the design-test cycles are the only means for developing finished robot behavior. In creating Edmund, we have tried to design a platform for developing complex behavior. Ideally, the behavior library should provide reliable, robust elements of behavior, while the control scripts provide flexible ways for expressing that behavior at times suited to the situation and task. To bootstrap, we organize the libraries along the lines of function and adaptive state requirements[2].

We have also been developing heuristics for determining how to develop the decomposition while developing the behavior. To begin with, we favor more simple elements. A primitive is preferred to an action sequence, a sequence to a competence. However, if for some script only part of a primitive behavior is required, then that primitive is broken down into elements, which are typically expressed in a sequence to replicate the former behavior. Similarly, 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.

A competence may be thought of, and designed, as a sequence of behaviors that might need to be executed in a worst-case scenario. The ultimate (last) element is the highest priority element of the competence, the penultimate the second highest and so on. Triggers on each element determine whether that element actually needs to fire at this instant. If a competence is actually deterministic, if it nearly always actually executes a fixed path through its elements, then it should be changed into a sequence.

For complex tasks such as those involving manipulation, one of the most difficult part of programming Edmund is guaranteeing correct behavior from the competences. (See the next section for a discussion of leveraging formal languages for Edmund scripts.) One way a competence can flag a need for incremental change in decomposition is by relying on large numbers of triggers. Perception should be handled at the behavior library level, it should be a skill. A large number of triggers should be converted into a single perceptual test. A perception so central to a competence may well be useful for other elements of behavior as well.

A simple example of these techniques can be drawn from the development of the motion primitives **move** and **correct-dir** mentioned in Section 4.2. **Move** initially behaved as it does now, but as part of developing a behavior for reliable segments for a simple map-learning routine it was extended to correct the direction of travel. This invalidated an old script "wander," which changed directions without concern and was useful for herding the robot into desired areas. The new **move** was therefore broken into elements which included a cleaner version of its previous self, and **correct-dir**. These elements are now run as a Lap in the map-building scripts. (See Figure 2.)

5.2 Other Issues of Methodology

Another related methodological problem and technique is developing the scheduling for Edmund in dynamic situations. Scheduling in Edmund is inexact. Too many things may be scheduled per second with no direct indication they are failing to execute. This is to some extent an artifact of the reactive nature of the system — we expect some events and behaviors to arrest the attention of the agent.

[2] Bryson [5] has suggested earlier that adaptivity requirements serve as good indicators for behavior decomposition.

An initial schedule for Edmund can be computed with the help of simple benchmarking, facilities for which are built into the control program. These help determine the constraints on the number of elements run per second, which allows estimates for rate at which varying numbers of drives can be executed. On the Nomad, for example, sensing is fairly expensive and limits the number of cycles for the architecture to about 340Hz. However, if the robot stops moving and reduces its sense sample rate, the cycle rate increases by an order of magnitude. This suggests that the robot should engage in phased periods of activity, for example switching between sense-intensive work like exploration and compute-intensive work like map learning. This strategy is found in most mammals (e.g. [35]), and is facilitated by the Edmund drive system. We are developing a navigation system based on this for the robot have modeled such behavior in simulation.

Correcting scheduling problems can also be done incrementally; switching elements between competences and drives can also occur at the control script level. Elements in drives can be scheduled the most reliably. However, an element which is only required in a particular context may waste cycles if it is scheduled as a drive. A Lap for handling bumps was initially added into the competence for motion, “walk”, but this overlooked the fact some Laps in walk were long and the bumpers would not be checked between their elements once triggered. Consequently, “bump” was moved to the top level drive. On the other hand, a perceptual routine for watching for landmarks was successfully moved into “walk” since it was only needed in the context of motion, and could rely on perceptual memory if it was called less frequently.

6 Discussion: Related and Future Work

In describing his subsumption architecture, Brooks [4] states that development must begin from the bottom up, with each layer being fully tested before the next is added, so that at every level one is only debugging that level and its interface to the working sublayers. Although we consider this a good working principle, in our experience it is not easy to determine that layers are either fully or correctly implemented before other layers that need them have been added. Certainly evolution does not leave primitive brain organs completely untouched, but reuses them creatively [18].

Unlike Brooks, we do not expect layers will necessarily be fully or correctly implemented before moving to the next one. If as the project progresses problems are discovered in early levels, earlier scripts exercising only those levels can be used to test that redevelopment does not damage earlier behavior. The scripting facility also allows for easily switching elements between competences and drives. Our work has more in common with the multi-layered architectures that are currently dominating mobile robotics [1], such as [12, 15, 3]. However, we place more intelligence and learning in what corresponds to the lower, reactive level in these systems. We take the lesson from biology that learning and memory are a part of perception [6, 23], and from Brooks that perception and action must be tightly linked.

We are currently reviewing better established architectures. If we can model the essential elements of our approach in a system that provides better tools for development and analysis, we feel we should exploit these programming benefits. One obvious candidate is PRS, discussed extensively in this volume (for example [9, 17]). However, there would be a problem with implementation at the level of perception and belief. Edmund's beliefs consist of what is currently reported by its senses, combined with its perceptual memory and built-in biases. Moreover, for a real embodied system the entire sensory world cannot be attended to at once. PRS with its uniform sense cycle and rule-based belief libraries would not seem particularly suited to active or selective perception.

On the other hand, there could be a fairly direct mapping of competences to plan library and drives to goals, (though note the loss of scheduling, which relates to the perception points above.) Edmund could be said to keep track of its intentions through its system of control attention. Certainly Edmund's scripts can be described in formal languages such as Robot Schemes (RS) [19] or Structured Circuit Semantics (SCS) [17]. To borrow the formalism of SCS, an action sequence is simply a list of atomic actions (where Senses combined with their test are considered to be primitive actions, as are Acts) and a competence could be expressed as a `do first`. (Competences are also very similar to Nilsson's Teleo-Reactive programs [26].) Although the primitive actions change behavior slightly over time because of their learning, all of these systems have to be robust to unreliable behavior in actions, so these languages might still be used to assist in checking competences for internal cycles or deadlocks. Edmund might then also more easily "plug in" to multi-agent coordination systems such described in [17, 30]. The perception of having a flexible or inevitable action currently blessed by a coordinator [30] could be worked into the action selection routines.

There is a similar imprecise mapping of PRS back into Edmund, which may be useful if only insofar as the development of PRS agents might also exploit this sort of iterative design. So far, nearly all PRS-type architectures use hand-built plans for their plan libraries. Presumably these plans as well need to take into account their appropriate level of implementation.

Another possible direction for Edmund, however, is toward a less rigorously categorized system. In ENRL the object metaphor, particularly encapsulation, breaks down in relation to sensing. This is not surprising given that external to the robot, information is flowing in streams. Senses filter and, in the case of digital systems, segment this information. Many areas of animal brains, both cortical and subcortical, fuse information from multiple senses. In fact, very few animal behaviors are independent of sensory context. Pitch detection, time sense, and intentional learning and recall all show strong context dependency [6]. Encapsulation violations in the object library amount to methods or even functions calling for information from multiple classes. OOD methodologies address the problems of interdependencies by limiting and documenting the points of interface.

These OOD methodologies have been directly transferable to ENRL. It is possible, however, that the control structure itself may need to be fused with the perception and action units. We are currently exploring the use of episodic memory for tasks involving search of multiple contingencies, such as principled (directed) navigation. This may involve treating learned patterns like drives or competences, with associated patterns of habituation or inhibition.

7 Conclusion

This paper has presented an agent architecture, Edmund, as a demonstration of an interaction between object-oriented and agent design. We suggest that an agent architecture can serve as a guideline to application development. We also suggest object-oriented design methodologies such as encapsulation, specified interfaces, and evolving hierarchies are useful in creating a reusable library of behaviors for a particular application domain. If intelligence is to be achieved by design, it will necessarily be a learning experience for the designer. The programming of intelligent agents requires not only principled structure and approach, but also methodological support.

Acknowledgments

Edmund has benefited from discussions with many people, particularly Ian Horswill, Leslie Kaelbling, Brendan McGonigle, and Will Lowe. This paper has benefited from extensive comments by Will Lowe, John Hallam and Rob Ringrose.

References

1. Special issue: Software architectures for hardware agents, 1997.
2. J. S. Albus. The NIST real-time control system (RCS): an approach to intelligent systems research. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3), 1997.
3. R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):237–256, 1997.
4. Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, April 1986.
5. Joanna Bryson. The reactive accompanist: Adaptation and behavior decomposition in a music system. In Luc Steels, editor, *The Biology and Technology of Intelligent Autonomous Agents*. Springer-Verlag, 1995.
6. Niel R. Carlson. *Physiology of Behavior*. Allyn and Bacon, Boston, 5 edition, 1994.
7. David Chapman. Penguins can make cake. *AI Magazine*, 10(4):51–60, 1989.
8. Daniel C. Dennett and Marcel Kinsbourne. Time and the observer: The where and when of consciousness in the brain. *Brain and Behavioral Sciences*, 15:183–247, 1992.
9. Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In this volume.
10. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 202–207, 1987.
11. James Firby. Personal communication, 1995.
12. Erann Gat. *Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots*. PhD thesis, Virginia Polytechnic Institute and State University, 1991.
13. M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, 1987.
14. David W. Glasspool. Competitive queuing and the articulatory loop. In J. Levy, D. Bairaktaris, J. Bullinaria, and P. Cairns, editors, *Connectionist Models of Memory and Language*. UCL Press, 1995.

15. Henry H. Hexmoor. *Representing and Learning Routine Activities*. PhD thesis, State University of New York at Buffalo, December 1995.
16. Ian Horswill. Visual architecture and cognitive architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):277–293, 1997.
17. Jaeho Lee and Edmund Durfee. On explicit plan languages for coordinating multiagent plan execution. In this volume.
18. Peter J. Livesey. *Evolutionary Processes*, volume 1 of *Learning and Emotion: A Biological Synthesis*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.
19. Damion M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, 9(3), June 1993.
20. Pattie Maes. How to do the right thing. A.I. Memo 1180, MIT, Cambridge, MA, 1989.
21. Maja J. Mataric. Integration of representation into goal-driven behavior-based robots. *IEEE Journal of Robotics and Automation*, 8(3):304–312, June 1992.
22. Brendan McGonigle. Incrementing intelligent systems by design. In Jean-Arcady Meyer and Stuart Wilson, editors, *From Animals to Animats*, pages 478–485, Cambridge, MA, 1991. MIT Press.
23. Brendan McGonigle and Margaret Chalmers. The ontology of order. In Les Smith, editor, *Piaget: A Critical Assessment*. Routledge, 1996.
24. J. H. Neely. Semantic priming effects in visual word recognition: A selective review of current findings and theories. In D. Besner and G. W. Humphreys, editors, *Basic Processes in Reading: Visual Word Recognition*, chapter 9. Lawrence Erlbaum Associates, 1991.
25. Nils Nilsson. Shakey the robot. Technical note 323, SRI International, Menlo Park, California, April 1984.
26. Nils Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.
27. Van Parunak, John Sauter, and Steve Clark. Specification and design of industrial synthetic ecosystems. In this volume.
28. Miles Pebody. Learning and adaptivity: Enhancing reactive behaviour architectures in real-world interaction systems. In F. Moran, A. Moreno, J.J. Merelo, and P. Chacon, editors, *Advances in Artificial Life (Third European Conference on Artificial Life)*, pages 679–690, Berlin, Germany, 1995. Springer-Verlag.
29. W. A. Phillips and W. Singer. In search of common cortical foundations. *Brain and Behavioral Sciences*, forthcoming.
30. Munindar Singh. A customizable coordination service for autonomous agents. In this volume.
31. Jun Tanji. Involvement of motor areas in the medial frontal cortex of primates in temporal sequencing of multiple movements. In R. Caminiti, K-P Hoffmann, F. Lacquaniti, and J. Altman, editors, *Vision and Movement: Mechanisms in the Cerebral Cortex*, volume 2, pages 126–133. Human Frontier Science Program, Strasbourg, 1996.
32. Toby Tyrrell. *Computational Mechanisms for Action Selection*. PhD thesis, University of Edinburgh, 1993. Centre for Cognitive Science.
33. Christoph von der Malsburg. Binding in models of perception and brain function. *Current Opinion in Neurobiology*, 5:520–526, 1995.
34. Steven D. Whitehead. Reinforcement learning for the adaptive control of perception and action. Technical Report 406, University of Rochester Computer Science, Feb 1992.
35. Matthew Wilson and Bruce McNaughton. Reactivation of hippocampal ensemble memories during sleep. *Science*, 261:1227–1232, 29 July 1994.
36. Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.

This article was processed using the LaTeX macro package with LLNCS style