Object-Oriented Design of the Subsumption Architecture

Greg Butler*, Andrea Gantchev, Peter Grogono
Department of Computer Science, Concordia University, 1455 de Maisonneuve Blvd. West, Montréal, Québec, Canada, H3G 1M8

SUMMARY

The subsumption architecture is a layered mediator invented by Rodney Brooks for behaviour-based control of robots. The layers are minimally dependent and use minimal communication. We develop an object-oriented software design for the subsumption architecture, and demonstrate that each layer can be used as a slot for a set of plug-and-play components that implement different micro-strategies for achieving a particular goal. Guidelines for the development of specific layers and components of a subsumption architecture are also presented.

KEY WORDS: software architecture; object-oriented design; mediator; agent

INTRODUCTION

Many intelligent systems such as agents, knowledge-based systems, planners, and adaptive software need to be able to integrate and reuse a variety of decision-making components. Today most of these systems are developed in software, although there is still strong interest in autonomous robots that have such capabilities.

The subsumption architecture is a layered mediator invented by Rodney Brooks for behaviour-based control of robots. Behavior-based systems consist of task-oriented modules implementing domain-specific solutions with representations de-emphasized, and control decentralized. In subsumption, each layer of behavior includes as a subset the behaviour of the lower layers. There is a fixed priority arbitration scheme to handle conflicts between layers, and a layer may suppress or inhibit lower layers. The layers are minimally dependent and use minimal communication. We develop an object-oriented software design for the subsumption architecture, and demonstrate that each layer can be used as a slot for a set of plug-and-play components that implement different micro-strategies for achieving a particular goal. Furthermore, a set of guidelines for the development of specific layers and components of a subsumption architecture is also presented.

The software architecture, and the reuse of micro-strategy components, is validated by developing truck agents within the Truckin' simulation game. The game is played in a simulated country. In this country, there are three kinds of dealer: producers,

*Correspondence to: Greg Butler, Department of Computer Science, Concordia University, 1455 de Maisonneuve Blvd. West, Montréal, Québec, Canada, H3G 1M8 (email: gregb@cs.concordia.ca)
Contract/grant sponsor: Natural Sciences and Engineering Research Council of Canada, Fonds pour la Formation de Chercheurs et l'Aide a la Recherche of Québec, Seagram Innovative Research.

retailers, and consumers who trade in a single commodity. The producers, retailers, and consumers have fixed locations and cannot trade directly. They rely on trucks to ship goods from one place to another. Trucks can buy, sell, move, or make phone calls to gather information. Trucks can move around the country, using gas. The country contains several gas stations at which trucks can buy gas. A truck that does not trade will eventually not have enough money to buy gas. A truck without gas is stuck and can do nothing. The objective of the game is for retailers and trucks to trade in such a way that there is a steady flow of goods from producers to consumers. The winning truck is the one with the most capital at the end of the game. (The original Truckin' game was invented by Mark Stefik and others at Xerox PARC as part of their research into expert systems.)

The development of truck agents for Truckin' demonstrates that the subsumption architecture provides a well-designed structure for strategies, and that micro-strategy components can be reused as plug-and-play components within the subsumption architecture.

BACKGROUND

In the mid-eighties researchers in artificial intelligence began developing a new approach to designing mobile robots. Robots based on traditional AI approach were not able to operate in real time in a real world. They relied on a perfect internal representation of the world which can not be achieved in a dynamic unpredictable environment, and the focus on "depth" search to provide solutions was not timely enough in an environment where quick reaction is critical. Rodney Brooks argued that internal world models that are complete representations of the external world were unnecessary for robots to act in a competent manner. According to Brooks the world is its own best model [3]. Inspired by ethology and biology, he observed that actions of a robot are separable and that coherent intelligence could emerge from the interaction of independent reactive sub components with the environment [4]. Brooks aimed to build robots with intelligence on the scale of insects as the first step towards building robots with higher intelligence [1]. Brooks' approach to building insect-like robots is in many ways orthogonal to the approach taken earlier. Instead of a top-down, centrally controlled system built around an internal world model, Brooks' system is built bottom-up, has distributed control, with the components interfacing directly with the world (see Figure 1). This approach is referred to as behavior-based robotics.

Brooks described the subsumption architecture as an instance of behavior-based robotics used to build robots that operate in the real world [5]. It is a framework from which to build behavior-based robots. Maes [7] argues that the behavior-based approach is appropriate for the class of problems that require a system to autonomously fulfill several goals in a dynamic, unpredictable environment. She gives examples of applications, such as virtual actors, process scheduling, interface agents to name a few, where the behavior-based approach could be applied. While Maes argues for the generality of the behavior-based approach, Bryson [6] suggests that subsumption architecture can serve as a general framework to develop behavior-based systems.

Rodney Brooks points out that traditional AI had difficulty with the integration of multiple sensor devices, achievement of multiple goals, robustness, and extensibility when it came to systems for control of autonomous mobile robots [4]. The subsumption

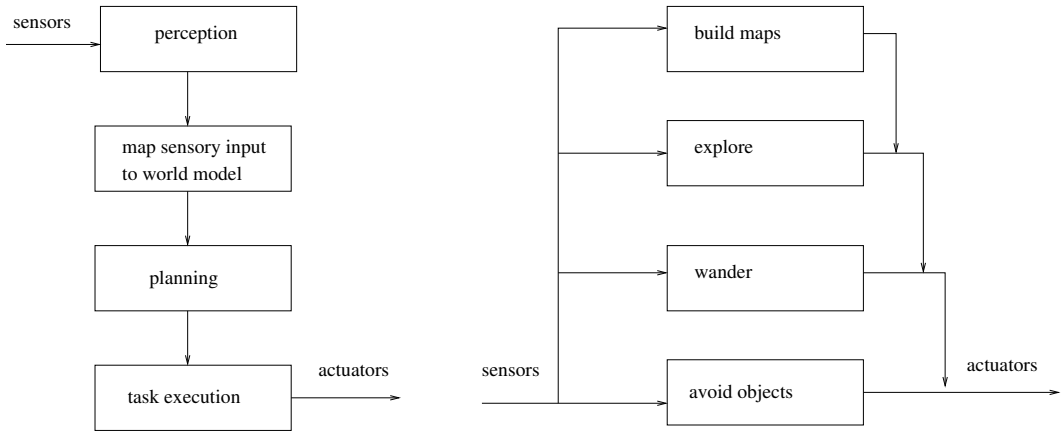

Figure 1. Traditional Approach vs Behaviour-based Approach

architecture was developed to address those difficulties. The fundamental ideas of the subsumption architecture are a decomposition into layers of task-achieving behaviors, followed by an incremental composition through debugging in the real world [3].

Brooks proposed a decomposition of an autonomous intelligent system based on desired external manifestations of the system. The decomposition resulted in a collection of simpler independent behaviors which when composed produced more complex behavior. Each behavior should achieve a task that is in some way observable. A set of behaviors together provide the robot with some level of competence. The behaviors should be designed so that as new behaviors are added to the system the level of competence of the system increases. A set of behaviors that produces a level of competence is referred to as a layer, and the process of increasing the level of competence by adding new behaviors to existing sets of behaviors is called layering [1]. Each layer connects its own sensing to action and is not dependent on any other layer to decide what it should do. The layers operate in parallel with minimal communication.

The overall system is robust and extensible. Multiple distributed layers of behavior mean there is less chance that the system will collapse given some drastic change in the world. Each layer has its own sensors to monitor the world, by sensing the environment often enough, it is able to decide on the appropriate goal to pursue in light of the current environment. The layers are able to make timely adjustments to their goals in response to changes in the world. New layers are added to the system without changing the original system, all that is required is to interface the new layer to the existing system.

Brooks' computational model is organized as an asynchronous network of augmented finite state machines, with a fixed topology of unidirectional connections. Each layer of control is a finite state machine with some instance variables, and input/output lines that can send and receive typed messages. The connections between layers are predefined wires which allow higher layers to suppress and replace input, or inhibit the output of lower layers. The messages sent over connections are small numbers. The meanings of the numbers are determined by the designers of the sender and

receiver, and are dependent on the state of both the sender and receiver. The layers operate asynchronously, each layer outputs actuators in response to its own sensory information. There is a fixed priority arbitration scheme to handle conflicts that occur when more than one layer produces actuators at the same time. Under this scheme only one layer has control of the robot's effectors at a time. All data is distributed over many computational elements, and there is no central locus of control.

The first three layers of behavior defined by Brooks for the robot Allen [2]

1. Avoid obstacles.
2. Wander aimlessly around without hitting things.
3. Explore the world by seeing places in the distance that look reachable and heading for them.

Each layer of behavior includes as a subset the earlier layers of behavior. Exploring includes the ability to wander without hitting things, wandering without hitting things includes the ability to avoid contact with objects. This permits layers to be built incrementally beginning with lowest layer on up. The design of a layer can rely on the presence of successful operational earlier layers. The layers do not call on one another explicitly, instead their reliance is implicit. The Wander layer does not have to worry about avoiding obstacles because there is an operational Avoid Obstacles layer that successfully ensures that obstacles are avoided.

The lowest level layer of control (avoid obstacles) was implemented and completely debugged before adding a higher layer. This layer results in a robot that avoids collision with objects. The robot moves away from approaching objects, and halts before colliding with stationary objects. When the next level of control (wander) is added the robot moves in a random direction every few seconds. The Wander layer suppresses the heading produced by the Runaway module of the Avoid Obstacles layer. In fact the Avoid module combines the two headings resulting in a heading that points in the direction specified by the Wander module, but avoids any obstacles. The Wander layer subsumes the Avoid Obstacles layer when it suppresses the output of the Runaway module.

The Explore layer looks for corridors of free space then moves the robot towards the free space. The Whenlook module of the Explore layer looks for a corridor of open space whenever it detects that the robot has been idle for a few seconds. It inhibits the Wander layer so it can take some pictures and process them without wandering away. The Avoid Obstacles layer continues to operate, ensuring that no objects collide with the robot. Once a free corridor is found a heading is sent to the Avoid module suppressing any heading that may have been produced by the Wander module. The Wander layer in turn suppresses the Runaway module of the Avoid Obstacles layer. The Explore layer subsumes the Wander layer whenever it inhibits or suppresses the Wander layer.

The three key ideas [5] introduced in the subsumption approach above are

1. Improvements in performance come about by incrementally adding more situation specific circuitry while leaving old circuitry in place, able to operate when new circuitry fails to operate. Each additional collection of circuitry is referred to as a new layer, and each new layer produces some observable behavior in the system interacting in the environment.
2. Keep each added layer as a short connection between perception and actuation.

3. Minimize the interaction between layers.

SUBSUMPTION SOFTWARE ARCHITECTURE

This section explains the design of the software architecture for subsumption and presents the validation of the reusability of micro-strategies within the Truckin' simulation game. The design follows the OMT notation [9] and was implemented in C++ [10].

Overview

The software version of the subsumption architecture preserves the essence of subsumption:

- a layered architecture,
- a fixed priority scheme between layers,
- the ability of a layer to inhibit lower layers, and
- the use of the "real world" as the source of feedback on the consequences of actions.

The major differences between the software architecture and the hardware architecture for subsumption are the sequential nature of the trucks and the lack of "sensors" to the "real world" since a truck is embedded in an existing Truckin' simulation framework.

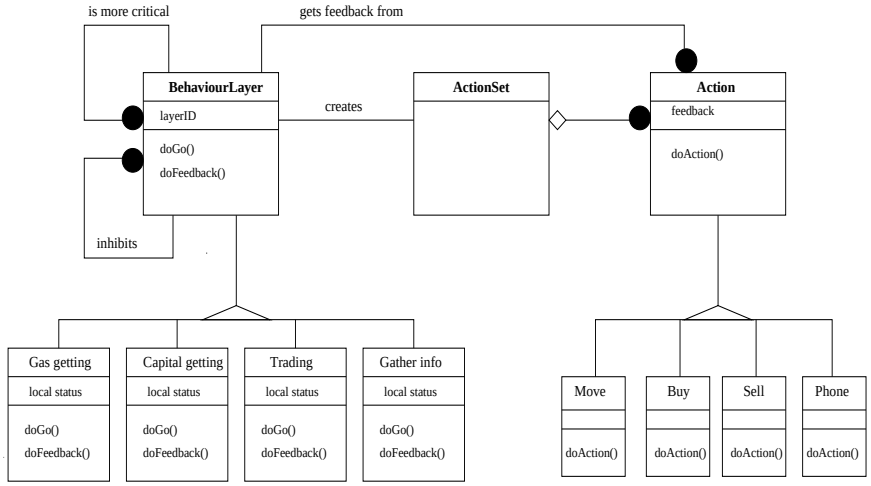

Figure 2. Conceptual View

The major concepts, see Figure 2, are the following:

- A BehaviourLayer is concerned with one particular set of behaviours and subgoals. The BehaviourLayers are linearly ordered in a fixed priority, where a BehaviourLayer subsumes the behaviour of the BehaviourLayer below it, and may inhibit all BehaviourLayers below it.
- An ActionSet is generated by a layer in order to make progress towards its subgoals.

- An Action is one of the primitive steps available to the agent. Each Action is known to the object ActionSet.

In the Truckin simulation, trucks can perform a combination of four actions, namely move, buy, sell, or phone for information. Each action has associated costs in terms of money, time, capital, and/or gas. The layers are associated particular subgoals, namely

1. Gas Getting, in order to retain the ability to move;
2. Capital Maintenance, in order to make a profit from the current trade with the local dealer;
3. Trading, in order to maximize overall profit from trades over the entire game; and
4. Gather information, in order to have knowledge of distant positions in the simulation world.

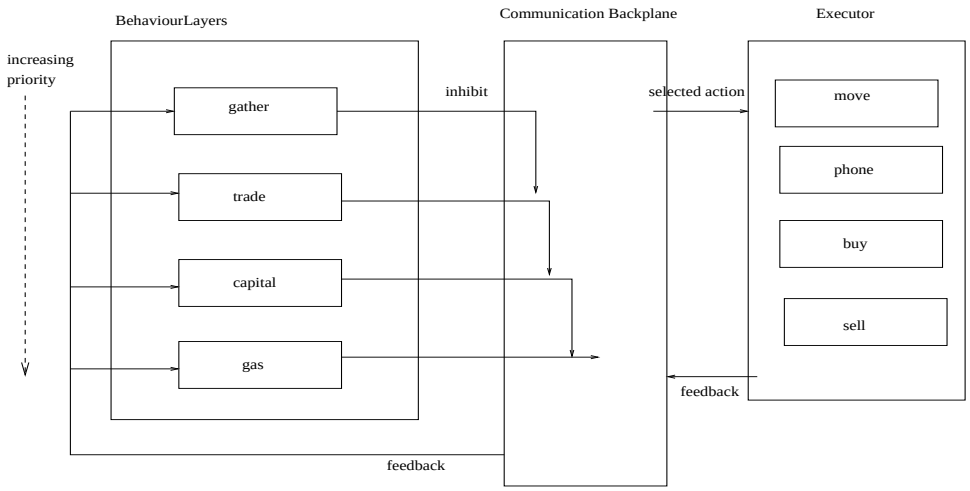

Figure 3. Overview of Architecture

The major architectural design decision is the use of a communication backplane. The backplane acts as a short-term memory, and as a communication channel, to provide the feedback from the "real world" to the layers. As short-term memory, the backplane stores the proposed sets of action and the inhibition information. With the fixed priority scheme, this memory allows the mediated selection of the current course of action. This selection is also recorded by the backplane and allows each layer to acquire the necessary feedback about the state of the world following the execution of the selected actions. Once the feedback has been acquired, the short-term memory is cleared.

The use of a communication backplane leads to a two phase behaviour of the layers: first obtain feedback, then propose an action set (and optionally inhibit lower layers).

Major Classes

Besides the classes representing the main concepts, BehaviourLayer, Action, and ActionSet, there are classes for the control and feedback mechanisms, namely Selector,

Executor, and CommunicationBackplane. These mechanisms utilise the ControlData class.

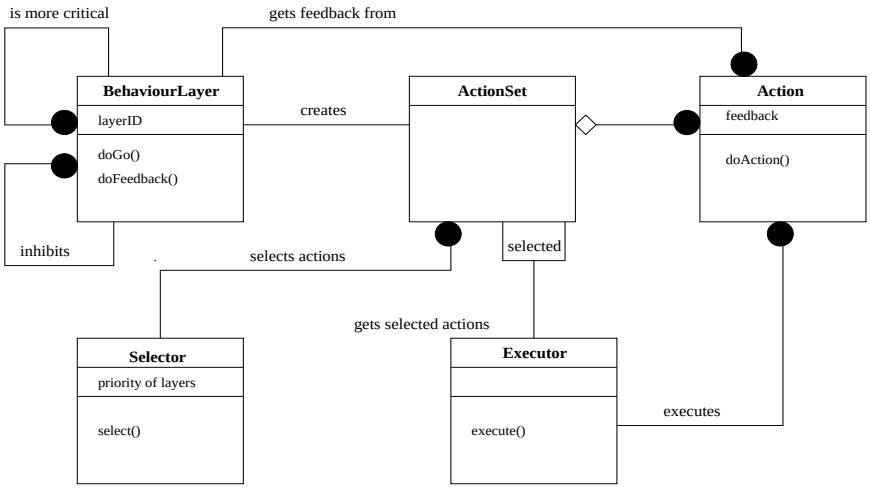

Figure 4. Object Model of the Design

A BehaviourLayer is concerned with one particular set of behaviours and subgoals. Associated with a BehaviourLayer is a *layerID* that determines its rank in the linear order of priorities, and information about a BehaviourLayer is usually indexed by its *layerID*. The behaviour of a BehaviourLayer is done in two phases: first to gather feedback from the "real world" using *doFeedback()*, and second to determine a set of actions using *doGo()*. The method *doGo()* may inhibit all BehaviourLayers below it. (These are the ones with higher priority.) A BehaviourLayer subsumes the behaviour of the BehaviourLayer below it.

An ActionSet is generated by a layer in order to make progress towards its subgoals. In the detailed design, this class does not exist; rather, it is a collection of actions stored within a ControlData object.

An Action is one of the primitive steps available to the agent. In the Truckin' simulation, trucks can perform a combination of four actions, namely move, buy, sell, or phone for information. Each action has associated costs in terms of money, time, capital, and/or gas. An Action is given effect by calling the *doAction()* method.

The CommunicationBackplane implements the associations amongst layers themselves, and the associations amongst layers and the Selector and the Executor. The CommunicationBackplane acts as a short-term memory, and as a communication channel, to provide the feedback from the "real world" to the layers. As short-term memory, the backplane stores the proposed action sets and the inhibition information: for each layer this is stored in a ControlData object. The Selector uses the fixed priority scheme and the inhibition information to select the current course of action, and records this selection in the CommunicationBackplane. This allows each layer to acquire the necessary feedback about the state of the world following the execution of the selected actions. Once the feedback has been acquired, the short-term memory is cleared.

The Selector is responsible for interpreting the fixed priority scheme, as defined

by the *layerID* of the layers, and for interpreting the inhibition information in the CommunicationBackplane in order to select suitable actions. The Selector also controls the deliberations of the subsumption layers. The method *select()* controls the two phase behaviour of layers by (1) calling *doFeedback()* for each layer, (2) clearing the short-term memory of the CommunicationBackplane, (3) calling *doGo()* for each layer, and (4) selecting appropriate actions.

The Executor is responsible for executing the selected actions. Its *execute()* iterates over the actions selected, and calls the appropriate *doAction()* methods.

The ControlData class is responsible for choosing actions whether those actions have been inhibited or not. A ControlData object is associated with each layer and is internal to the CommunicationBackplane.

Behavioural Description

We describe the behaviour of the subsumption architecture in a top-down fashion, first explaining how the behaviour of the Truckin' framework leads to calls to the *play()* method of Truck, and then how *play()* utilises the subsumption architecture. Last we describe the internal behaviour of an individual BehaviourLayer.

The outcome of a Truckin' simulation is determined by a truck's performance in a *competition* with other trucks. Each competition consists of a series of *games*. Games vary in the starting position of trucks, and in the position (and kind) of dealers in the simulated country, but all games have the same participating trucks. A game is played for a certain number of *rounds* where first the dealers play and then the trucks play. The order amongst dealers and amongst trucks is random from one round to the next, however all dealers play before any truck plays. When it is a truck's turn to play, the Truckin' framework calls the *play()* method of the truck's Controller, which in turn calls the *play()* method of the truck.

All interactions between a truck and the Truckin' framework go through the truck's Controller, which guarantees that the truck obeys the rules of the game, and correctly calculates usage of resources.

A truck plays a turn within a given time slot. A truck can complete several actions within that time. Using the subsumption architecture, a truck repeatedly calls the *select()* method of the Selector followed by the *execute()* method of the Executor until the time slot expires. Figure 5 shows a sequence diagram of this behaviour and Figure 6 an extended object diagram with pseudocode.

The Selector requests each layer to perform its feedback phase by calling *doFeedback()*. The Selector then clears the short-term memory of the CommunicationBackplane. Then the Selector requests each layer to determine its possible actions by calling *doGo()*. Finally, the Selector uses the priority scheme and the available inhibition information in the CommunicationBackplane to flag one of the action sets.

During *doFeedback()*, a layer requests from the CommunicationBackplane the action set that was selected previously. During *doGo()*, a layer creates a new action set and communicates that set to the CommunicationBackplane. Optionally, a layer may also communicate inhibit signals to the CommunicationBackplane.

The Executor fetches the selected action set from the CommunicationBackplane, and requests that each action be executed by calling *doAction()*, which in turn calls the corresponding method of the truck's Controller.

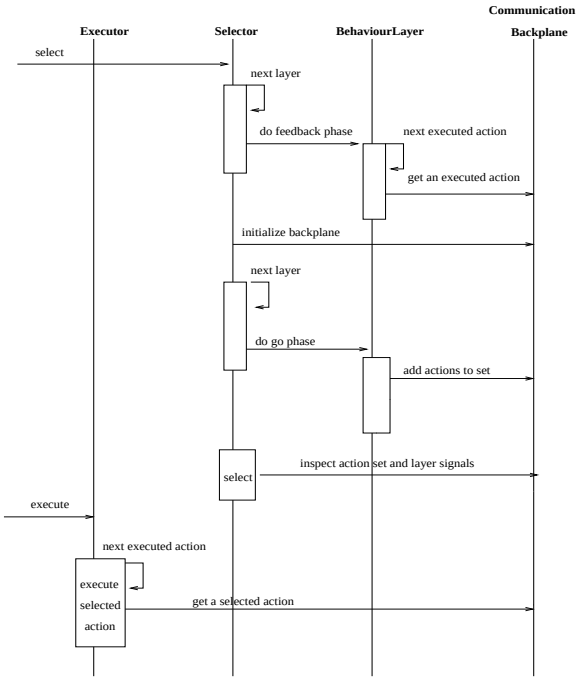

Figure 5. Dynamic Model of Architecture

Validation of Design using Truckin'

In the Truckin' game, as described in the Introduction, trucks compete with one another to transfer goods from producers to retailers and from retailers to consumers. If the trucks were identical, the outcome of the game would depend on random factors such as the starting positions of the trucks and would not be very interesting. In fact, the trucks differ from one another in various ways, and the simulation is written in such a way that an optimal set of features is found by a process analogous to Darwinian natural selection. (This idea is the basis of genetic algorithms and evolutionary programming, but a detailed discussion of these ideas is beyond the scope of this paper.)

In our simulation, each layer corresponds to a gene and the set of all layers corresponds to the genome of the truck. Each layer can be implemented in several different ways, corresponding to the alleles (specific manifestations) of a gene. The "fittest" truck is the truck that has a combination of layers that maximize the truck's profitability.

For each layer we develop five components as instances of strategies. The strategies range from very basic to moderate in complexity. While most strategies are designed as state machines, they are not implemented as such. These are not strategies for the overall goal of winning the game, but strategies for achieving the subgoal(s) of one behaviour layer, so we call them *micro-strategies*.

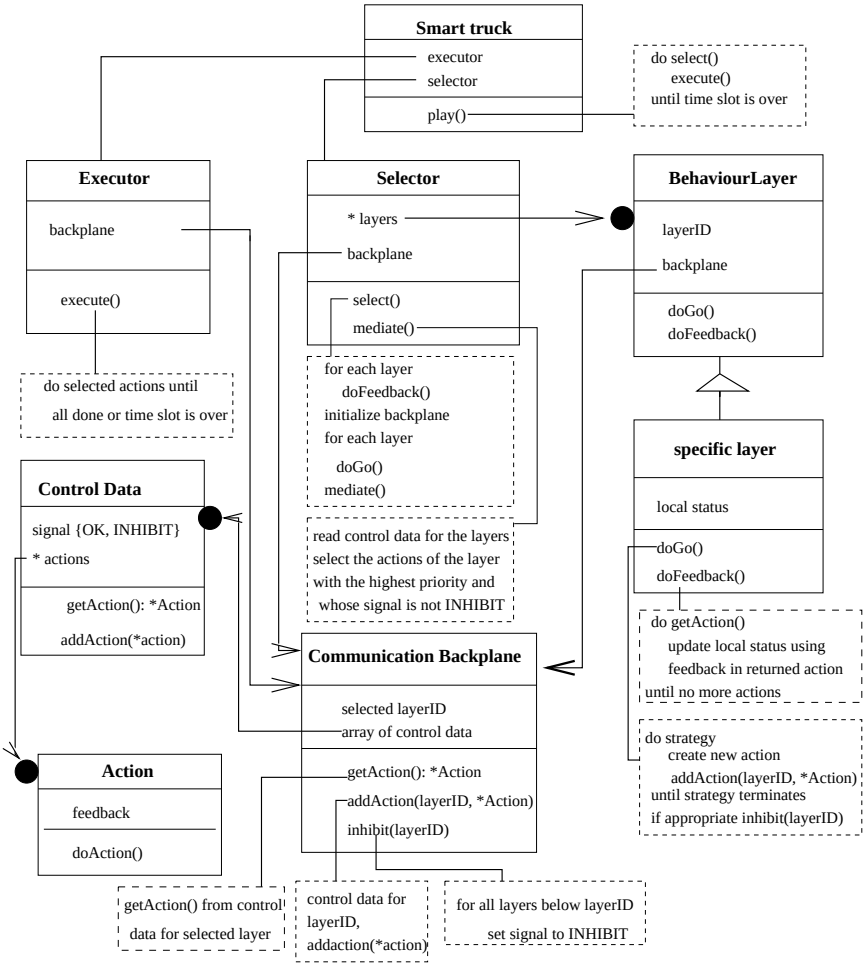

Figure 6. Behavior of Subsumption

GUIDELINES FOR DEVELOPING SUBSUMPTION SYSTEMS

Subsumption systems model the interaction dynamics of the components of the system and the environment in order to produce the desired results. As a consequence, the designer must determine the reflex modules and how they should be combined for each task and environment. It's often not possible to transfer a solution for one class of problems to another, instead, the architecture provides a set of principles and a set of examples that might be useful to the designer.

Brooks provided a methodology to develop subsumption robots that manages the complexity of achieving emergent behaviors, and addresses the complexity associated with distributed control. A bottom-up decomposition, incremental design, testing and debugging in the real world, offers a controlled way to achieve the desired behaviors. Distributed control of many modules is simplified by the minimization of inter-module dependence and communication. The guidelines that follow, derived from [1, 2, 8], keep the development process within the boundaries of the subsumption methodology.

Guideline 1: Decompose the system into task-achieving behaviors beginning with the basic set of reflexes that provide survival in a dynamic unstructured world. A task-achieving behavior is observable in the world. Proceeding bottom-up, define task-specific modules which when combined with existing modules increase the capabilities of the system. See [8] for heuristics describing the process of using task-specific constraints to generate behaviors.

This guideline provides a qualitative way to decompose the system, resulting in layering of behaviors that can be incrementally designed and implemented. The overall behavior of the system provides top-down constraints on the bottom-up decomposition, as a consequence every behavior is geared towards the purpose of the whole system.

Guideline 2: A layer is a collection of task-achieving behavior modules that together produce a level of competence. The activity of a layer is stimulated by events in the environment, not instructions from another layer. Most information is obtained directly by the layer itself through sensing the environment. Perception should be tightly coupled to action within a layer. A layer is designed to take small incremental steps towards its subgoals relying on frequent sensing of the world for dynamic error correction. At each step a layer must react quickly enough to able to sense changes in the environment as they occur.

The effect of this guideline is autonomous reactive layers whose integration and communication medium is the environment.

Guideline 3: Layers may interact with lower layers via their input and output. A higher layer may suppress the input or inhibit the output of a lower layer. Suppression includes the ability to replace the input to a lower layer, whereas inhibition causes the output of the layer to be ignored.

This guideline contributes to extensibility. Direct inter-layer communication is minimized resulting in simple layer interfaces. Simple interfaces coupled with a hierarchical ordering of layers facilitates the process of layering; incrementally adding higher levels of competence to the system.

Guideline 4: There should be no simplified test environments. Subsumption applications must cope with unpredictability in the environment, and with imperfect sensory information.

The layers are designed to take advantage of the dynamics of their actions on and in some environment. Testing in a simplified environment could lead to a design of a layer that depends on some simplified property, which is not true in the real environment. This dependency will then propagate to the higher layers which rely on the lower layers.

Guideline 5: Each layer should be tested, and debugged extensively in the real world before adding another layer to the system.

This guideline helps to simplify the debugging process. When a new layer is added to the existing system any bugs are likely to be in the new layer, or in the interface of the new layer with the system, and not in the existing system. Any bug fixing will be confined to the new layer.

CONCLUSION

The subsumption architecture is a layered mediator for behaviour-based control of robots. In subsumption, each layer of behavior includes as a subset the behaviour of the lower layers, there is a fixed priority scheme to handle conflicts between layers, and a layer may suppress or inhibit lower layers.

We have demonstrated an object-oriented software design for the subsumption architecture, and demonstrate that each layer can be used as a slot for a set of plug-and-play components that implement different micro-strategies for achieving a particular goal. The software architecture uses of a communication backplane that acts as a short-term memory, and as a communication channel, to provide the feedback from the "real world" to the layers. Furthermore, a set of guidelines for the development of specific layers and components of a subsumption architecture is presented.

The software architecture, and the reuse of micro-strategy components, is validated by developing truck agents within the Truckin' simulation game.

ACKNOWLEDGEMENTS

Helpful discussions were had with other members of the Truckin' project, especially with Debbie Papoulis and Jeff Edelstein, and with Jeff Allen from University of Maine.

REFERENCES

1. R.A. Brooks, *Achieving Artificial Intelligence Through Building Robots*, MIT A.I. Memo 899, May 1986.
2. R.A. Brooks, *A robust layered control system for a mobile robot*, IEEE Journal of Robotics and Automation, RA-2, March 1986, 14-23.
3. R.A. Brooks, *Intelligence without representation*, Artificial Intelligence Journal (47), 1991, pp. 139–159.
4. R.A. Brooks, *Intelligence without reason*, Proceedings of 12th Int. Joint Conf. on Artificial Intelligence, Sydney, Australia, August 1991, pp. 569–595.
5. R.A. Brooks, *From earwigs to humans*, Robotics and Autonomous Systems, Vol. 20, Nos. 2–4, June 1997, pp. 291–304.
6. J. Bryson, *The reactive accompanist: Adaptation and behavior decomposition in a music system*, The Biology and Technology of Intelligent Autonomous Agents, Springer-Verlag, 1995.
7. P. Maes, *Modeling adaptive autonomous agents*, Artificial Life 1(2), 1994, 135-162.
8. M. Mataric, *Behavior-based control: Main properties and implications*, in Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems, Nice, France, May 1992, 46-54.
9. J. Rumbaugh, M. Blaha, W. Premerlane, F. Eddy, and W. Lorenson, *Object-Oriented Modelling and Design*, Prentice Hall, 1991.
10. Stroustrup B., *The C++ Programming Language*, 3rd edition, Addison-Wesley, Reading, Mass., 1997.