April 1990 LIDS-P-1961

THE DESIGN OF SIMULATION LANGUAGES FOR SYSTEMS WITH MULTIPLE MODULARITIES[1]

by

S. Rowley
C. Rockland

This research was supported by the Army Research Office under grant ARO-DAAL03-86-K-0171 and the National Science Foundation under grant NSF ECS-8809746.

The Design of Simulation Languages for Systems with Multiple Modularities

Steve Rowley [*] and **Charles Rockland** [†]
Laboratory for Information and Decision Systems, MIT
77 Massachusetts Ave.
Cambridge MA 02139 USA

Abstract

Biological systems exhibit several characteristics that are not shared by human-engineered systems: there are often no clear module boundaries (or there are several module boundaries, depending on the question being asked); individual parts often serve multiple roles, depending on the behavior being studied; and system characteristics often vary from individual to individual.

Conventional simulation languages, however, do not cope well with this non-modularity. We have developed a theory of the design of simulation languages for such systems, and partially verified it in one case study. We separate the notion of "structure" S of a system from the "behavior" B of its parts. We allow multiple versions of both the structure $(S_1, S_2, \cdots)$ and the corresponding behavior $(B_1[S_i], B_2[S_i], \cdots)$ of each part S_i. The different structures or behaviors might be alternative theories, or abstractions of each other, for example.

We also have theories of how to interpret the simulations produced by $(S_i, B_j[S_i])$ pairs. One goal is to extract "design" information, i.e., explain how the system solves problems. Another is to test the effects of alternative models of behavior $B_1[S_i], B_2[S_i], \cdots$ for the same structure, or the effects on behavior due to alterations in structure. A third is to judge the relative degree of consistency between various $(S_i, B_j[S_i])$ pairs.

We exhibit how these ideas apply to the motor nervous system of the nematodes *C. elegans* and *Ascaris suum*. We also provide arguments that this kind of simulation methodology is also applicable to engineered artifacts, such as systems where parts must serve multiple roles.

1 Introduction

Most effort in simulation languages has been directed at the simulation of human-engineered artifacts. For example, when designing complex systems such as oil refineries, computers, or information systems, we would like to have a high degree of confidence that the system will work according to design. Does the oil refinery pollute too much? Is the computer fast enough? Will the information system correctly transaction-lock 125 asynchronous database queries?

Traditionally, the design of simulation languages has been driven by that task: given a set of rules for how the

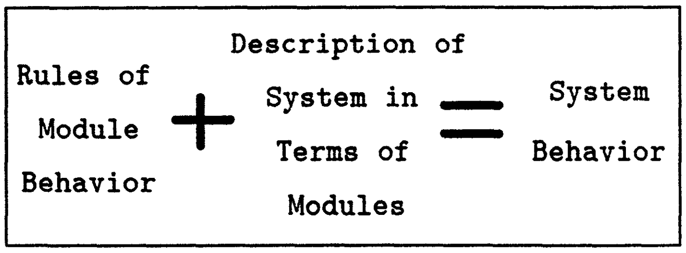

Figure 1: Using component behavior to compare system behavior with design goals.

components of a system work, describe, as in Figure 1, the overall behavior of a collection of such components and describe the degree to which that behavior meets design criteria.

However, such engineered systems share a trait in common: ***abstraction of function into modules.*** That is, the designers of the systems, in order to cope with the complexity of their task, broke the system up into pieces, each of which has some small set of functions and a simple interface to the rest of the world. Oil refineries are made of pipes, pumps, and tanks; computers are made of subsystems for memory, IO, and floating-point arithmetic; databases have files, schemata, and query interfaces. Each of these "parts" has a well-defined and small functionality, and a fairly rigid interface to other parts.

In recent years, the interest in biological systems [Larkin 88; Carruthers 88] has motivated many to study systems that do *not* have this modularity feature.[1] Biological systems use a single part – e.g., a nerve cell – for many different functions, depending on the external conditions, developmental stage of the organism, whether other parts of the organism need to be compensated for, and so on. For example, it is well-known to neurophysiologists that the nervous system can functionally compensate for certain types of damage.

Here we report on a case study of the development of a simulation language for such multi-modular systems, applied to the motor nervous system of the nematodes *Caenorhabditis elegans* and *Ascaris suum* [Kenyon 88;

[*] Also of Center for Theoretical Physics, MIT.

[†] Also of Center for Intelligent Control Systems.

[1] And in some highly-constrained engineered systems, such as the space shuttle. On the space shuttle, a given hull tile has both a heat-conducting function and an aerodynamic function. These functions may come into direct conflict when, for example, we want a particular tile to be thick to be a good insulator, but to be thin to be of the right shape to make the shuttle airbody have the right lift-drag characteristics.

Key Words: Biological simulation, nematodesm, modularity, language design.

Wood 88; Stretton 85]. The "parts list" for *C. elegans*, i.e., all the cells, including their complete developmental history, is quite well-known from biological experiment [Wood 88]. The "wiring diagram" for the nervous system is fully mapped, too. However, any decomposition of this system into "modules" is, at least in part, in the eye of the beholder. For example, while the time-course of assembly of cells into the juvenile organism is known in detail, its logic remains mysterious [Lewin 84]. Some decompositions make some questions easy to answer, but other questions hard. We therefore call such systems *multi-modular.*

We begin in the next section by showing an example of an engineered system that has multiple modularity boundaries, and discuss briefly the problems in simulating it. Then we discuss some language design issues which arise in addressing those problems. We find that object-oriented programming languages and databases offer much power in helping the simulator keep track of the multiple modularity boundaries. Finally, we realize that multi-modular systems might have the property that they produce many simulations, with possibly conflicting results, depending on what modularity decomposition you choose to look at. Our aim is to build a theory of how to weld together those possibly-conflicting simulations into a coherent theory about the multi-modular system in question.

2 What Makes a System Multi-Modular?

A *multi-modular* system is one for which there is no unique, fundamentally motivated decomposition into modules that works acceptably well for all reasonable questions we might want to ask about it. There may be a (hopefully small) *number* of such decompositions, each of which works acceptably well for a class of questions. We want to focus on the impact that multiplicity of viewpoints has on attempts to simulate such systems, and upon our attempts to interpret those simulations in terms of theories of system behavior.

2.1 Pump or Heater? A Multi-Modular Engineered Artifact

To get an understanding of the implications of multi-modularity, let's take a look at a (poorly!) engineered artifact that exhibits these characteristics. Consider an alleged "water pump," shown in Figure 2.

This pump/heater takes in electrical energy E at one port and water at low pressure p at another port. It puts out the water at higher pressure $p+\Delta p$ at a third port (doing work W in the process) and waste heat Q at a fourth. Now, most pumps will be designed in such a way that the waste heat Q is negligible, encouraging the *abstraction* that a pump is a machine for converting electrical energy E into a pressure difference Δp. That is, we can *simplify* our model of the pump for most purposes, ignoring the waste heat.

However, suppose that's not true in this case, i.e., that the waste heat Q is significant. After all, the pump/heater's primary loyalty is to the First Law of thermodynamics, $E = W + Q$, not to our notions of what constitutes useful behavior. We are equally justified in looking

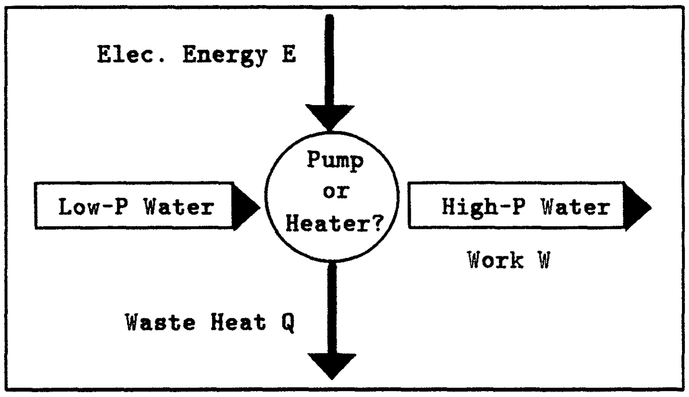

Figure 2: An electric water pump. Or is it an electric heater?

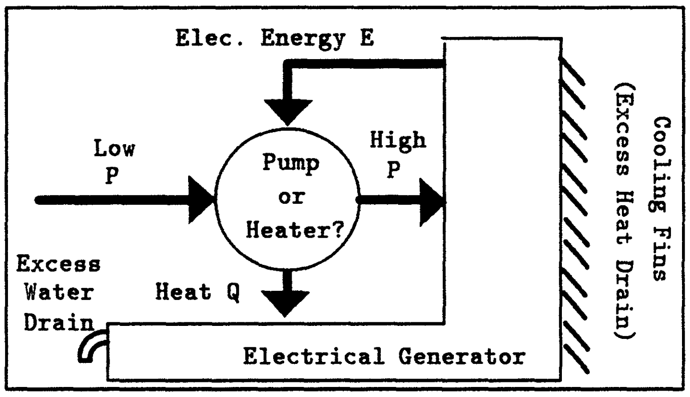

Figure 3: A pump/heater & generator system. Non-modularity propagates!

at the device as an electrical heater instead of as a pump! It has more than one useful way of being put into a surrounding system. For example, consider the new system in Figure 3, where we have incorporated the pump/heater into an electrical generator.

At first blush, we might be tempted to divide this system into two modules, corresponding to the pump/heater and the generator. However, it doesn't take long to discard that idea. The generator might well call upon the pump/heater when it either needs water pumped into it, or when it needs to be warmed up. Either view of the pump/heater will do. However, the functions of heat and pumping are no longer *separable*, i.e., the action is non-modular. If the generator is already too hot but needs water pumped anyway, it must take steps to compensate for the extra heat. It might, for example, have a set of cooling fins that it can open to release heat or close to retain heat. Similarly, if the generator already has enough water but needs heat, it has to have a way to drain off the excess water pumped. The state transition diagram for the way the generator operates the pump is shown in Figure 4.[2]

[2]Note that there are other states possible, e.g., Pump On, Drain Open, Fins Out. For now, this odd state doesn't make sense, since it turns on the pump and then compensates for both of the pump's effects, wasting electricity in the process. An engineer would just turn down the generator instead. How-

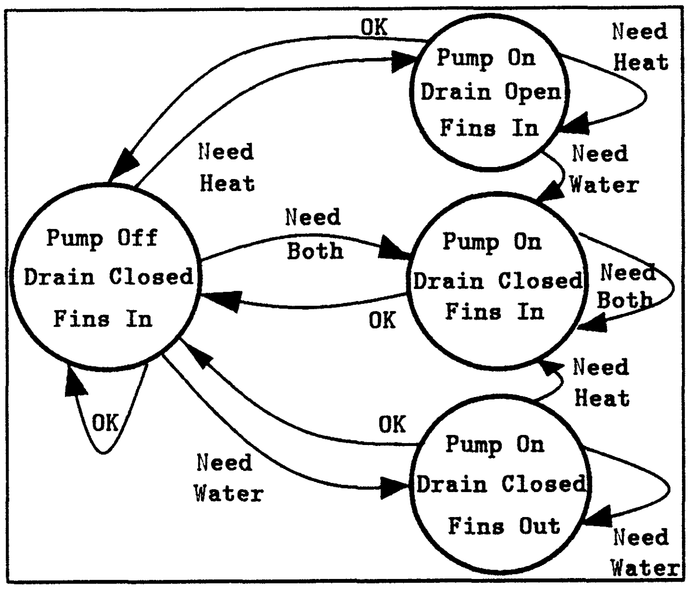
Pump Off Drain Closed Fins In
Pump On Drain Open Fins In
Pump On Drain Closed Fins In
Pump On Drain Closed Fins Out
OK
Need Heat
Need Heat
Need Water
Need Both
OK
Need Both
Need Heat
OK
OK
Need Water
Need Water

Figure 4: State transition diagram for the pump/heater & generator system.

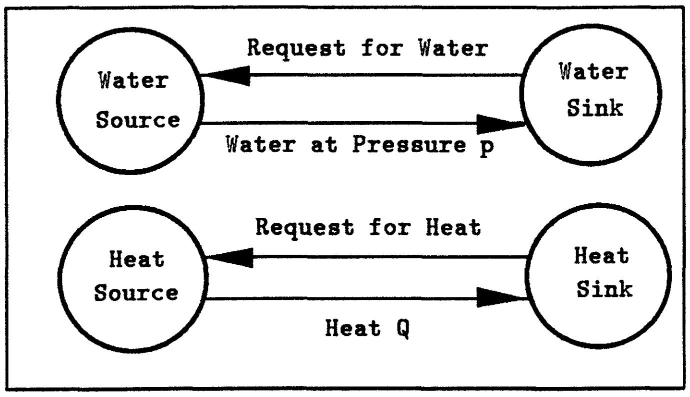

Figure 5: The pump/heater & generator system, viewed as a water circuit and a heat circuit.

Thus we see that the non-modularity of the pump/heater has *propagated* into the systems nearby, so they can compensate, extracting just the parts of the functionality they need. If we insist on a module decomposition between the pump/heater and the generator, we must live with the fact that treatment of the pump waste heat is now *non-local*, i.e., created in the pump "module" and radiated away in the generator "module." Similarly, the pumped water is created in the pump module, but sometimes compensated for by a drain in the generator module. Once several such non-localities crop up, we must begin to suspect that our decision to divide the observed system into pump and generator modules might have been wrong. Perhaps we should be dividing it into water circuits and heat circuits, instead, as in Figure 5.

Initially, this looks pretty good: at least it separates the heat and water flows, treating them as separate "modules." Well, almost: the notion of pump and generator have gone away! Even though the real world contains an artifact for the pump and one for the generator, there is no artifact we can point to in this model and say, e.g., "This is the generator." To recover them requires *coupling* the apparently separate heat and water flows. Once again, non-locality has reared its ugly head: this time, the useful notions of pump and generator have been smeared across module boundaries.

We just can't win: there is no single decomposition of this system into modules that works for all reasonable questions we might ask about it.[3] There is no question that this is bad engineering design: it would have been a *much* better idea to separate the functions of pumping and heating by making the pump deal with its own waste heat, perphaps by giving it its *own* cooling fins. This would have contained the heat problem within a module, saving us the headache of dealing with the heat non-locally. We could then add a separate heater module, if required. The pump/generator modularity would be thereby validated, instead of giving rise to a competing water/heat circuit decomposition.

2.2 Nematode Biology as a System with Multi-Modularity

However, there are many situations in Nature where you are not allowed to redesign the parts you're given. For example, marine plankton are known [Beardsley 89] to emit dimethyl sulfide gas as a by-product. Dimethyl sulfide in turn has a meteorological effect in cloud formation; cloud formation regulates sunlight input to the plankton. Thus the plankton and cloud systems are coupled in a way that invites us to question whether or not the whole of the division into subsystems of algae and atmospheric gasses is useful.

Another example [Carruthers 88] is that of the grasshopper *C. pellucida* and its fungal pathogen *E. gyrilli*. The dyanics of the two organisms tempts us to lump them together into a single entity that responds to environmental stresses through *either or both* grasshopper and fungal populations.

We have begun studying non-modularity in the architecture and behavior of the nematode *C. elegans* [Rockland 89], Figure 6. *C. elegans* is a small (ca. 1mm), free-living (i.e., non-parasitic) soil roundworm, dining on bacteria.

ever, the non-modularity of biological systems means that evolution would equally favor a kludge that used the pump as a place to dump excess electricity, thus using this odd state.

[3]Of course, the same is true of truly modular systems, if we admit "unreasonable" questions, i.e., ones that have nothing to do with the design goals of the system. In order to know what a – presumably modular – electrical generator smells like, we'd need to analyze the chemistry of all its components in the presence of high electromagnetic fields. Fortunately, we rarely care about the smell of a generator. Still, it is exactly questions like this that vex us in many environmental hazards: carpet glues that work admirably as glues later turn out to emit formaldehyde, i.e., "smell" in a dangerous way. Sometimes they cause illnesses, unless compensated for by air-handling equipment (non-locality!). The real behavior of a system in principle depends on *all* the physics involved, not just that piece we dignify as the "purpose" of the system.

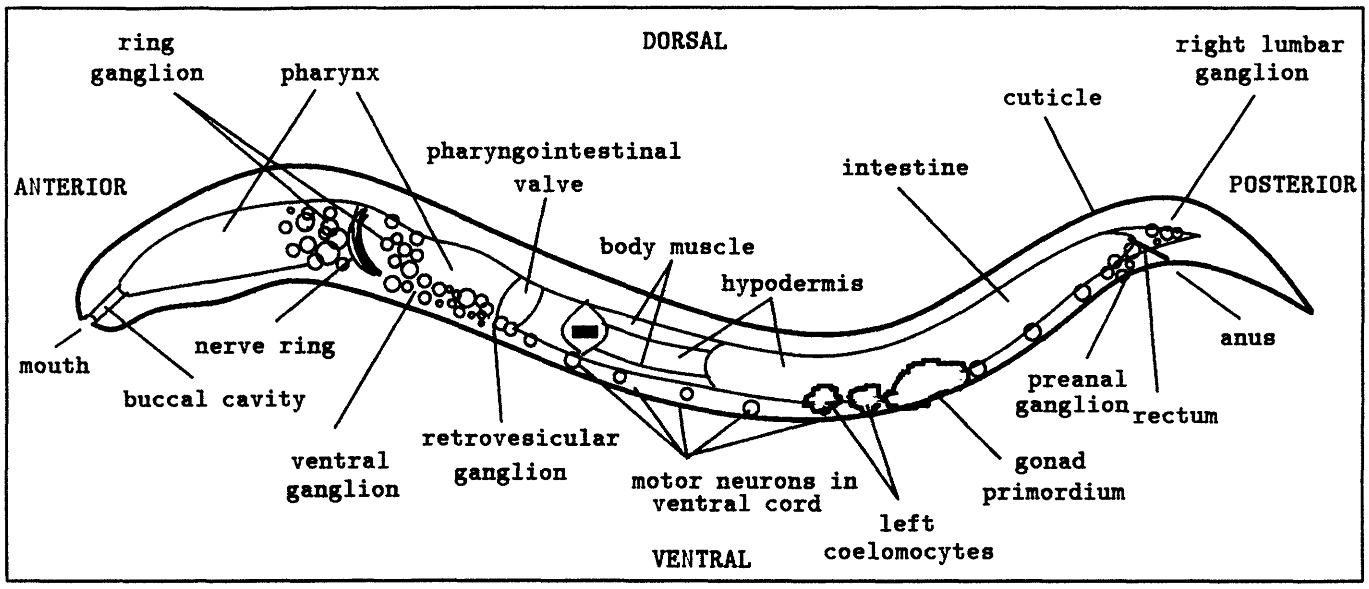

Figure 6: The nematode *Caenorhabditis elegans*, after [Sulston 83].

It is, in various respects, the simplest existing cellularly-differentiated organism. It has, over the past 25 years, been used as a model system for concerted study of development, structure, function, and behavior [Wood 88]. Thus, it is perhaps the best-documented organism on the planet:

- It is has exactly 959 cells, all of which are classified and named. They are invariant from individual to individual.
- There are 302 neurons, for which a complete wiring diagram is known.
- The complete lineage of every cell, all the way back to the fertilized zygote, is known.
- The genetics is exceptionally well-understood, and a large range of mutants have been catalogued.
- Despite the small number of neurons, the animal is capable of a rich variety of behaviors: locomotion, head-waving, pharyngeal pumping, response to touch, response to gradients in temperature or chemical concentration, etc. Some of these give evidence of plasticity, and even rudimentary learning.

We chose to work with this organism because of the large amount of data available, and because the small cell number is likely to enhance the extent of mutual coupling and integration of the various control structures. Our goal is to provide a framework for integrating all this data, in an attempt to help come up with a theory of how the organism works. For example, there is a related pig parasite, *Ascaris suum*. Experiments on *Ascaris* complement those on *C. elegans* [Stretton 85; Walrond 85]. *Ascaris*, about 30cm long, has around 50,000 cells, but a nervous system of only 298 cells, essentially homologous to that of *C. elegans*. Is the control system for *C. elegans* "over-engineered," or does *Ascaris* have some trick for controlling an order of magnitude more muscle cells with an "under-engineered" control system?

Our project has begun to study the motor nervous system of the worm [Rockland 90], i.e., the set of neurons and body-wall muscles that control locomotion by propagating a wave of muscle contraction along the body. The motor nervous system of the worm can, in one modularity, be viewed as 5 repeated "segmental oscillators," as in Figure 7a [Stretton 85; Walrond 85]. We should emphasize that this picture is *already* an abstraction of reality; the "real" worm has 11 neurons in each segmental oscillator, not 4. In addition, the actual neurons are branched structures with a precise geometry of synaptic connections along their fibers. That is to say, the neurons and synaptic relations in Figure 7a are aggregations of real neurons ignoring synaptic geometry, i.e., someone's idea of a module. This is what we call *abstraction of structure*: we may wish to have multiple different views of what constitute the parts of the system under study, ignoring details we think might be irrelevant.

For example:

- [Walrond 85] deals with all the synapses that seem to be present in a segmental oscillator on the basis of structural, anatomical criteria; we call this the "anatomical" segmental oscillator, as in Figure 7b.
- On the other hand, [Stretton 85] identifies just those synapses that seem to be physiologically active; we call this the "physiological" segmental oscillator, as in Figure 7a.

The physiological structure is an abstraction of the anatomical in which the apparently nonfunctioning synapses are ignored. It is this abstraction on which we shall base examples here; the anatomical case, as well as more detail on the physiological case is in [Rockland 90]. Thus the structural abstractions that we have dealt with are, in order of increasing abstraction:

1. The full network of neurons, complete with synaptic geometry.

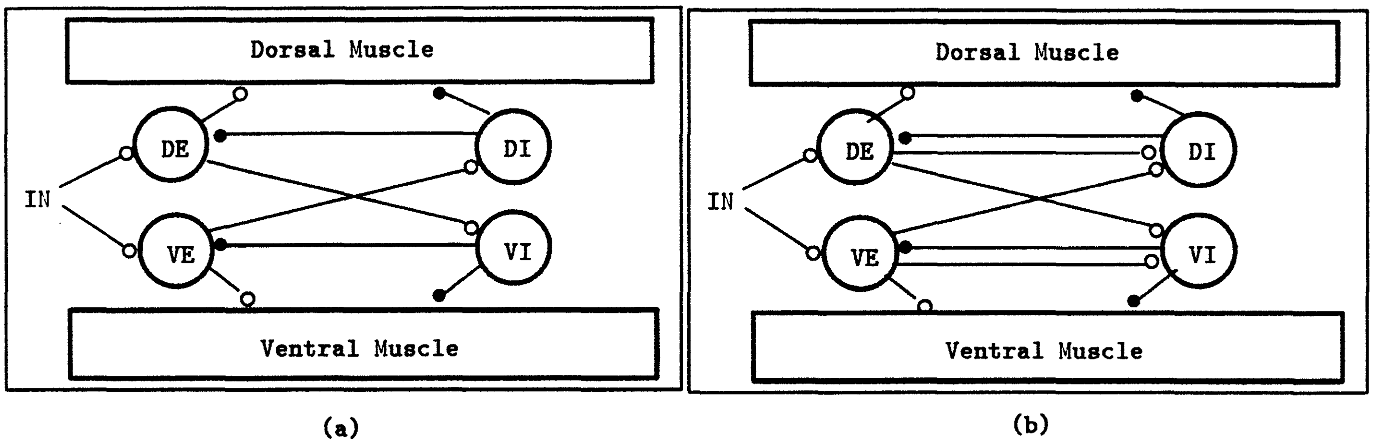

Figure 7: (a) The physiological version of the segmental oscillator. DE = dorsal excitor, VI = ventral inhibitor, etc. IN = interneuron, an excitor from the rest of the nervous system. (b) Anatomical version. Some of these synapses are physiologically inactive.

2. A segmental oscillator structure that lumps the 11 neurons into 4 groups, but retains all synapses – the anatomical case.
3. A segmental oscillator structure that abstracts out (i.e., ignores) all but the physiologically active synapses – the physiological case.

In order to simulate the behavior of such a segmental oscillator, we need models for how muscles and nerves behave. That is, we want to know:

- what *state* do the components carry,
- what *relations* to the state of other components do they have, and
- how does the state *evolve* in time?

For example, one model of nerves & muscles is rather like digital electronics:

- *Binary State*: Each type of cell has associated a static number called a "threshold," and is in one of two states: "firing" or "not firing."
- *Synaptic Relations*: All components evolve in synchrony with a global clock. At each clock tick, cells sum up the weights of synapses from firing cells. (Inhibitory synapses have negative weights.)
- *Evolution*: If that sum is above the threshold, the cell goes into state "firing" on the next cycle, otherwise it goes into state "not firing."

Even this extremely simplistic model of component behavior can give rise to interesting global patterns of worm-like behavior, such as "coiling" and "shrinking," describable in terms of phase coupling of oscillators of period 1, 2 or 4 [Rockland 90]. Moreover, by comparing global behavior patterns resulting from the anatomical and physiological versions, we can determine which criterion for the presence of synapses better reflects reality. On the other hand, there are many more realistic behaviors of neuron and muscle! We might want to consider the effect of asynchrony, or the dispersion of electrical waves as they propagate along the waveguide-like processes, perhaps with a cable equation [Rockland 89; Niebur 86; Niebur 88].

Alternatively, we can model the details left out of the digital behavior above with a probabilistic background: a neuron fires with probability p_1 if it is above threshold, and fails to fire with probability p_2 if it is below. This generates a 2-parameter family of behaviors that approach the digital one as $p_1, p_2 — 1$.

We would very much like to know which details of behaviors are essential for the worm, and which are just accidents of Nature. That is to say, there are many different views of behavior, as well as structure. We call this *abstraction of behavior*: more abstract behaviors ignore some details, and should generate correspondingly less detailed simulations. Using this idea, we can explore the space of cell behaviors and determine which behaviors are essential for explaining observed behaviors of the worm, and which are mere accidents of Nature.[4]

3 Simulation Language Design Issues for Multi-Modular Artifacts

Our notion of simulating multi-modular systems is founded on both abstraction of structure and abstraction of behavior. That is, we have multiple structural decompositions of a system $S_1, S_2, \cdots$. For example, two different decompositions of the systems discussed above were the pump and generator decomposition versus the decomposition into heat-flow and water-flow circuits. Similarly, the 2 forms of segmental oscillators in *C. elegans* are competing structures for the same object. Also, we have multiple theories of the behaviors of the parts with structure S_i: $B_1[S_i], B_2[S_i], \cdots$. For example, the "digital" behavior of neurons above, and a more refined cable-theoretic behavior of the same neurons might be competing behavior hypotheses, or coarse-grainings of each other. We want to find out

[4]Of course, changing selection pressures on a population may select for a previously-inessential trait. What we're really doing is finding the features of structure and behavior that are important for explaining experiments.

which behavior model is right, or we want to find out how much detail is necessary.

Once we make the decision to have structures $S_1, S_2, \cdots$ and behaviors $B_1, B_2, \cdots$, we have to face two problems:

1. We have to keep track of the relationships between the various B_i's, making sure they share code where possible.
2. We have an enormous space of possible simulations, possibly as big as the fiber product[5] $S \bigcirc B$ of structures and behaviors. That is, we may have a simulation for each $(S_i, B_j[S_i])$ pair.

3.1 The Object-Oriented Structure of Vermis

The usual solution to the first problem is the use of object-oriented programming languages.[6] Our system, called Vermis, is implemented in Common Lisp [Steele 84] using the Common Lisp Object System (CLOS) [Keene 88] on the Symbolics Lisp Machine [Symbolics 90]. This is an important insight: by using *protocols*, we can define the functions that can be used on objects in our languages, without having to define their implementation.[7] Furthermore, the use of inheritance permits us to recycle old methods, i.e., build upon and share the code of others.

The first problem we attacked was figuring out the classes from which user-defined structures and behaviors would inherit. They are shown in Figure 8.

- The theory of structure in Vermis is one of recursive containers: objects can be inside other objects, for whatever "inside" might mean in your theory. We provide generic functions for mapping functions over an object's contents, finding the container of an object, and so on. In addition to the "inside"-ness relation, we can define other relations between objects, such as synaptic connections, chemical influences, heat circuits, etc. For example, the pump and generator system of Figure 3 would be an object containing 2 parts: the pump/heater and the generator. We create additional relations that describe electrical, water, and heat connections. The pump/heater and generator might then have their own recursive structure, as well.
- The theory of behavior in Vermis is that the objects in structures have state; the behavior defines a way of evolving that state in time by looking at containment and other relations. We provide generic functions for changing the current state and evolving the state of a composite structure to a later state. For example, the pump and generator system of Figure 3 would assign state variables such as temperature, water content, and power output to the generator. The evolution of those variables would depend on their current values, possibly their time history, and the heat, electrical, and water connections to the pump.

[5]That is, at each structure S_i, we erect a fiber of appropriate behaviors $B[S_i]$.

[6]See, e.g., [Carruthers 88; Larkin 88] for a similar application to insect disease dynamics.

[7]See, e.g., [Rowley 87] for another example of the use of protocols in language design.

Those classes that are ancestors of **basic-behavior** are for implementing behaviors; those that are ancestors of **basic-structure** are for implementing structures. Obviously, some are for both. The "division of labor" amongst the classes of Figure 8 is this:

- **named-object-mixin** provides names by which all instances of classes may be referred to. There are various generic functions to ask an object what its name is, make up a new name for a new object, and find an object with a given name.
- **instance-remembering-mixin** provides generic functions for mapping a function over all instances of a class, and for accessing instances built on **named-object-mixin** by their names.
- **hierarchical-object-protocol** defines a protocol for objects that can be nested "inside" each other. It requires its client classes to supply methods for generic functions that map over contents, find the container a given object is in, draw the object and its contents to a graphics stream, and generally to understand the idea of containment. Also, it has a notion of state, so that it can redisplay the object appropriately when it changes.

  It does not, however, supply an implementation of container data structures. It is a *protocol* class: it requires the programmer to mix in an implementation class, and checks that it is complete at compile time.
- **abstract-container-mixin** is a partial implementation of **hierarchical-object-protocol**. It provides a slot for the container of an object and the required methods to alter it. It does not, however, supply a complete implementation of **hierarchical-object--protocol**, as it does not know how to represent an object's contents. Such classes are called "abstract."
- **container-only-mixin** completes the implementation of **hierarchical-object-protocol** started in **abstract-container-mixin**, by providing methods that do nothing (gracefully) when we attempt to refer to an object's contents. That is, it models objects that can be part of other things, but are themselves atomic.
- **abstract-container-&-contents-mixin** builds on **abstract-container-mixin**, supplying a slot for the object's contents. It does not, however, commit to a particular representation for the set data structure of the contents: it could be a list, array, hash table, etc. Hence this class is "abstract," as well.
- **list-contents-mixin** completes the implementation of **hierarchical-object-protocol** started in **abstract-container-&-contents-mixin** by supplying methods to treat the contents of an object as a list.
- **hash-contents-mixin** completes the implementation of **hierarchical-object-protocol** started in **abstract-container-&-contents-mixin** by supplying methods to treat the contents of an object as a hash table. For large numbers of objects contained, this can lead to a speedup over **list-contents-mixin**, but such a table takes more memory. For small numbers of contents, a list is usually faster.

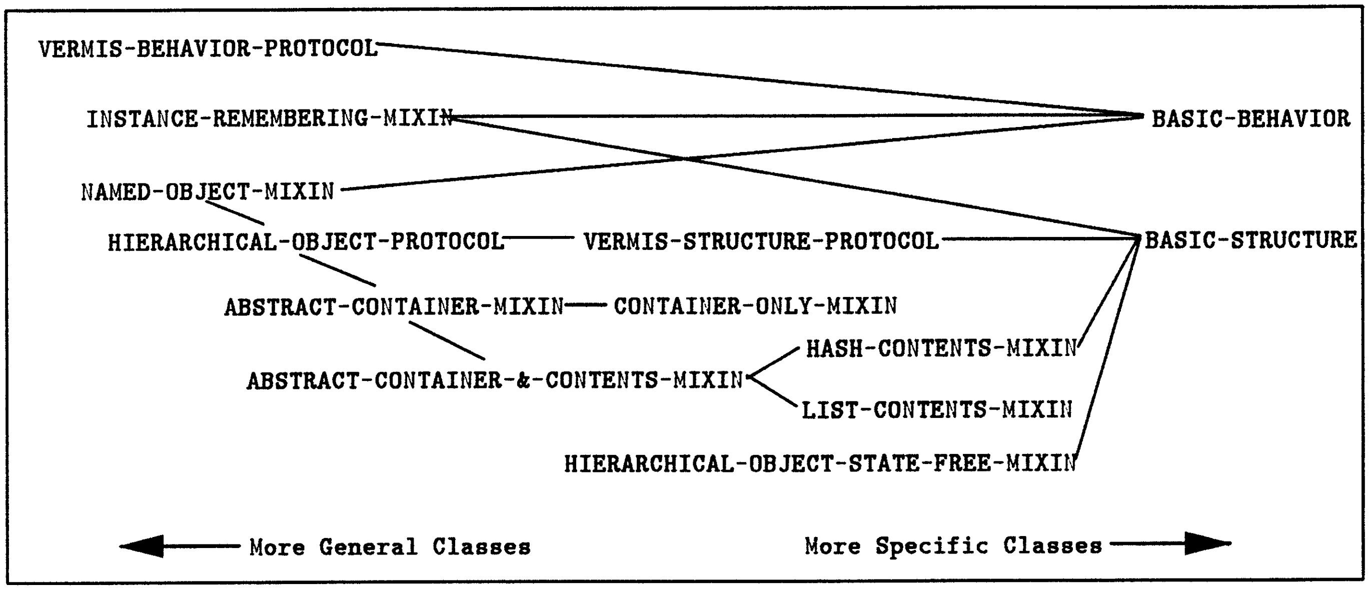

Figure 8: The basic class structure of Vermis

- **hierarchical-object-state-free-mixin** speeds up graphical redisplay of hierarchical objects by noting that this object has no state, i.e., its display can never change as a function of time.
- **vermis-structure-protocol** is a protocol class, like **hierarchical-object-protocol**. This one defines the protocol obeyed by all structures that will be simulated by Vermis. This consists of the protocol of **hierarchical-object-protocol** augmented by a few more generic functions for creating and killing instances.
- **vermis-behavior-protocol** is also a protocol class. It defines the protocol obeyed by all behavior models. It consists of the protocols of **named-object-mixin** and **instance-remembering-mixin**, along with the generic functions:
  - **set-state** writes a new state into an object, presumably having computed it via **note-next--state**.
  - **note-next-state** figures out, from the current state of an object, its relations with other objects, and a behavior, what its next state will be. This state cannot be written into the object until all the objects have been advanced, however. The Vermis software takes care of this by computing all the new states first and then writing them into the objects together.[8]
  - **evolve-organism** takes an object and a behavior model, and finds the total state of the object at some future time.

These all combine to form the root classes on which users build their structures and behaviors: **basic-structure** and **basic-behavior**. To date, we have built in Vermis several interesting such structures and behaviors, shown in Figure 9.

We define new structures in Vermis with the **defstructure** defining form. **defstructure** is a recursive specification of the parts of an object, their initialization keywords, and the relations between them. For example, to define the physiological segmental oscillator of Figure 7a, we could do something like Figure 10[9].

The code of Figure 10 constructs an object out of parts from previously-defined structures, namely **nerve-cell** & **muscle-cell**. The relation **synapse-between**, also defined elsewhere, "wires up" the segmental oscillator as shown in Figure 7a.

We have used this recursive definition method to define the following structures for simulation:

- **segmental-oscillator** is just like the above example, but more general. It is a prototype for *both* the anatomical and physiological oscillators; a keyword at *make-instance* time specifies which version to use.
- **segmental-linear-oscillator** is more like what actually occurs in the real worm, i.e., we string together N of the segmental oscillators, using some simple synapsing rules (detailed in [Rockland 90; Stretton 85; Walrond 85]). An example is shown in Figure 11a.
- **segmental-ring-oscillator** is used to determine which of the global behaviors of **segmental-linear--oscillator** are due to edge effects, i.e., absence of left- or right-hand neighbors for the extreme segments.

[8]This does not mean Vermis is commited to a discrete notion of time. One could construct a behavior model that uses a computer algebra package, like MACSYMA [Symbolics 86], to solve continuous models exactly. In that case, **note-next-state** would use the value of that solution at intervals.

[9]Actually, we're supressing a lot of detail here for pedagogical purposes. In fact, we would define one structure for both the anatomical and physiological segmental oscillators. Also, a lot of geometrical information – affine transformations to position the parts graphically – has been suppressed.

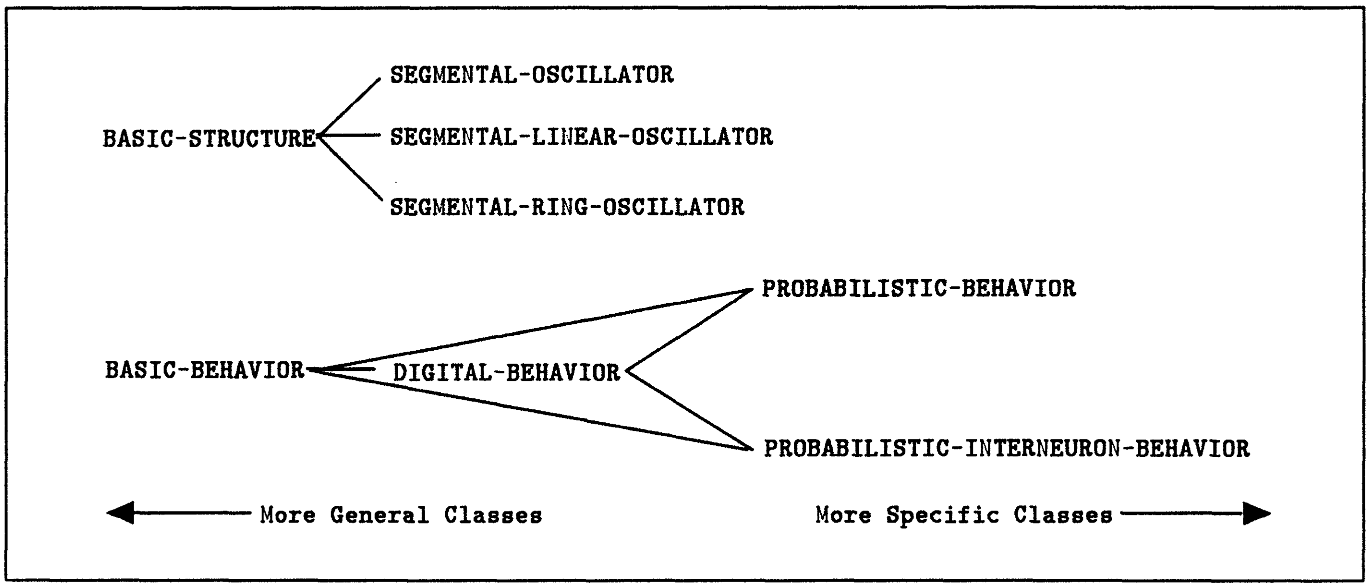

Figure 9: The derived classes built in Vermis.

```
(defstructure physiological-segmental-oscillator ()
  ;; the physiological segmental oscillator takes no parameters.
  :parts '((nerve-cell  :name "DE"             :threshold 1 :firing-p t   :geometry ···)
           (nerve-cell  :name "DI"             :threshold 1 :firing-p nil :geometry ···)
           (nerve-cell  :name "VE"             :threshold 1 :firing-p nil :geometry ···)
           (nerve-cell  :name "VI"             :threshold 1 :firing-p t   :geometry ···)
           (muscle-cell :name "Dorsal Muscle"  :threshold 1 :firing-p t   :geometry ···)
           (muscle-cell :name "Ventral Muscle" :threshold 1 :firing-p nil :geometry ···))
  :relations '((synapse-between "DE" "Dorsal Muscle"  :weight +1 :geometry ···)
               (synapse-between "DE" "VI"             :weight +1 :geometry ···)
               (synapse-between "DI" "Dorsal Muscle"  :weight -1 :geometry ···)
               (synapse-between "DI" "DE"             :weight -1 :geometry ···)
               (synapse-between "VE" "Ventral Muscle" :weight +1 :geometry ···)
               (synapse-between "VE" "DI"             :weight +1 :geometry ···)
               (synapse-between "VI" "Ventral Muscle" :weight -1 :geometry ···)
               (synapse-between "VI" "VE"             :weight -1 :geometry ···))
  ···)
```

Figure 10: Example definition of the structure of a segmental oscillator, with physiological synapses. Compare Figure 7a.

We impose circular boundary conditions, i.e., make an annular worm. An example is shown in Figure 11b.

That is, one can both *parametrize* structures and *compose* them to form larger structures. We use this often to explore variations on a structural theme.

Similarly, the mechanism for making a behavior specification is the defining form **`defbehavior`**. **`defbehavior`** lets you define a new class, just as CLOS **`defclass`**, but also encourages you to supply method definitions for **`note-next-state`** and **`evolve-organism`**. Since the behaviors are implemented as classes, you can use inheritance to share code. For example, the digital model alluded to above can be defined as in Figure 12a. Using inheritance, the probabilistic firing model alluded to above may be built on top of it as in Figure 12b.[10] Note that the probabilistic version need only override the **`note-next-state`** method; it simply inherits the rest.

Our project involved a lot of experimentation with alternative versions of both structure and behavior. That is, we didn't start with specifications for structural and behavioral models; rather, we viewed Vermis as a system for prototyping and exploring alternatives. This sort of prototyping view meant that we depended strongly upon:

- object-oriented programming (provided by CLOS),
- a flexible language with functions as first-class data objects (provided by Common Lisp),
- and a very good program development and debugging environment (provided by the Symbolics system).

We found the fast compile-test-debug-edit loop, single uniform address space, user-interface management system, and abstraction power to be essential in tackling tough simulation problems.

3.2 Theory Formation: Managing the Mayhem of Multiple Modularities

Once we accept the theory of multiple modularity, we can generate a large number of simulations, one for each element $S \odot B$, the fiber product of the structures and behaviors. There are a couple of facts that come to our rescue, so we don't actually have to look at that many simulations:

- Most behavior models are written with a particular structure, or family of structures in mind. It doesn't necessarily make sense to apply a behavior model to *all* available structures.
- People don't propose families of structures and behaviors that are unrelated; they almost always have some theory of how they are related.

  For example, the physiological segmental oscillator of Figure 7a is an ***abstraction*** of the anatomical segmental oscillator of Figure 7b, which is in turn an abstraction of the real thing. Similarly, the ring oscillator of Figure 11b should approach in its simulations the linear oscillator of Figure 11a as $N \to \infty$, for any behavior model applicable to both.[11]

$$\begin{tikzcd}
S_2 \odot B_2[S_2] \arrow[r, "\sigma"] & \Sigma_2 \\
S_1 \odot B_1[S_1] \arrow[r, "\sigma"'] \arrow[u, "\mathcal{A}_S \odot \mathcal{A}_B"] & \Sigma_1 \arrow[u, "\mathcal{A}_\Sigma"']
\end{tikzcd}$$

Figure 13: Algebraically commutative diagram showing the relationships between elements of fiber product of structures and behaviors on the one hand, and the simulations they generate on the other.

The existence of this homomorphism, i.e., abstraction relation, makes us think there should be a similar homomorphism, holding at least statistically, between simulations. If structure S_2 is an image of S_1, then there should be a corresponding map, such as averaging or coarse-graining, between simulations involving S_1 and S_2.

Similarly, behavior models can be related. For example, the two behaviors of Figure 12 are related in that the second becomes the first as the probabilities $p_1, p_2 \to 1$. We can test the simulations involving them to see if the distributions of behaviors become statistically indistinguishable in this limit.

The second of these points is particularly important: given a set of mappings relating structures $S_1, S_2, \cdots$ and behaviors $B_1[S_i], B_2[S_i], \cdots$, a multi-modular simulation engine such as Vermis becomes a testing ground for performing numerical experiments. The theories being tested are what we think those mappings preserve, i.e., the statements about structures and behaviors for which they are homomorphisms. Statistical measures of the relations between the simulations should verify or refute the theories allegedly relating $S_1, S_2, \cdots$ and $B_1[S_i], B_2[S_i], \cdots$. This is illustrated in Figure 13. Let $\mathcal{A}_S$ and $\mathcal{A}_B$ be homomorphisms (e.g., abstraction operators) that are claimed to relate the structures and behaviors, respectively. These constitute the "theory" the person inventing the simulation is trying to test. Let σ be the map from a $(S_i, B_j[S_i])$ pair to a simulation Σ_i they generate. Then the algebraic diagram should commute; that is, there should exist another homomorphism $\mathcal{A}_\Sigma$ that relates the simulations. Vermis may be viewed as a system for generating the homomorphism $\mathcal{A}_\Sigma$

[10] Once again, we have suppressed some detail, such as the mechanism whereby new states get recorded and then written back into the cells.

[11] Provided, of course, that measurements are taken in a region not immediately dominated by edge effects. That means it must be distant from the edges of the linear oscillator, and only for times such that the region cannot either have communicated with the edge of the linear oscillator, or propagated information around the ring oscillator.

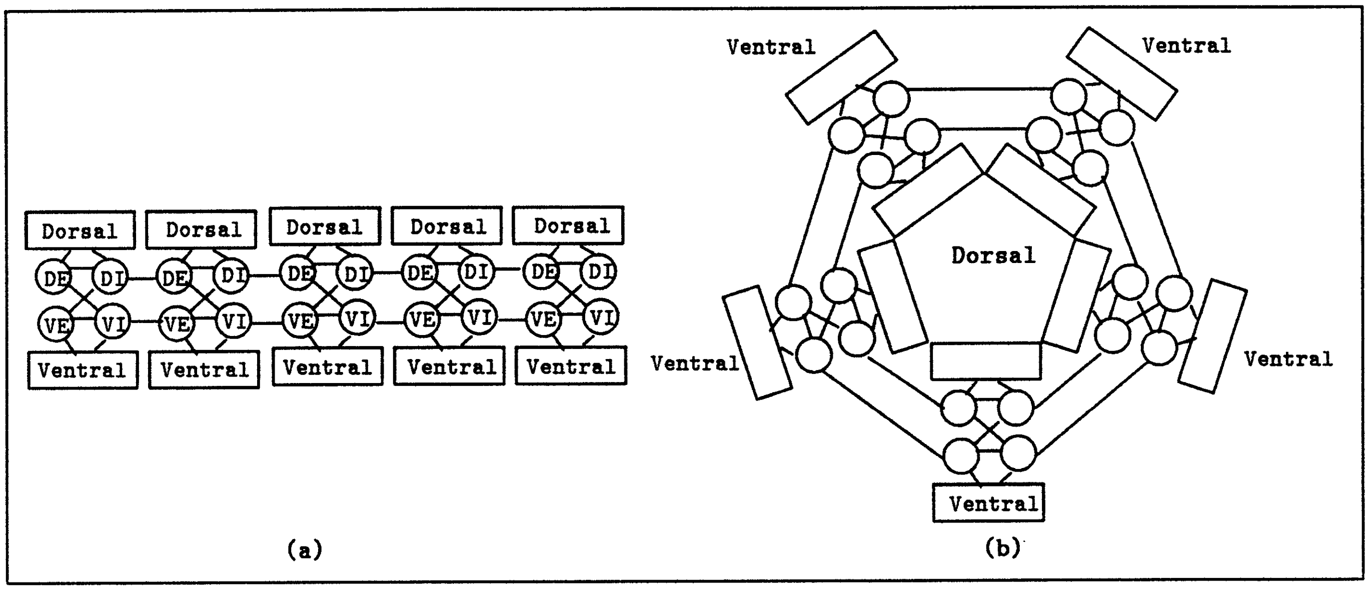

Figure 11: (a) The segmental linear oscillator, physiological synapsing rules, for $N = 5$. (b) The segmental ring oscillator, physiological synapsing rules, for $N = 5$.

```
(defbehavior digital-behavior (···) ()
  :evolve-organism ((organism) ···)
  :write-back-states (() ···)
  :note-next-state ((cell)
                    (let ((new (above-threhold-p self cell)))
                      ··· save new state ···)))

                      (a)


(defbehavior probabilistic-behavior ((p1 0.8)     ;prob firing above threshold
                                     (p2 0.1))    ;prob firing below threshold
             (digital-behavior)
  :note-next-state ((cell)
                    (let ((new (true-with-prob
                                 (if (above-threhold-p self cell) p1 p2))))
                      ··· save new state ···)))

                      (b)
```

Figure 12: A digital cell behavior, and a probabilistic generalization of it. **above-threshold-p** is true if the incoming synapses are above the cell's threshold. **true-with-prob** takes a probability and returns true with that probability.

to see to what degree it is consistent[12] with the product homomorphism $\mathcal{A}_S \odot \mathcal{A}_B$. Alternatively, one might notice among a family of simulations that there is a simple homomorphism $\mathcal{A}_\Sigma$ and seek to find an explaining theory, i.e., construct the $\mathcal{A}_S \odot \mathcal{A}_B$ that underlies it.

We are at present working on mechanisms to store simulations and the structures & behaviors that generate them in an object-oriented database [Symbolics 88]. We anticipate drawing upon various AI techniques, in particular truth maintenance [deKleer 86] and qualitative physics [Forbus 84], to extract information about theories from the rich set of relationships between simulations, structures, and behaviors.

3.3 Comparison with Diagnosis & Design Systems

We might compare this work in some respects to the work in AI on diagnosis and design expert systems [Davis 84; Hamscher 88; Williams 87]. The general idea in those systems is to use a model for what a system is *supposed* to do (based on engineering documentation) to attempt to diagnose it when it fails, from first principles. "Diagnose" in this context essentially means to come up with a minimal theory of what device is broken that explains observed behavior.

Our project is the reverse: given observed behavior of worms, we want to be able to derive the "design theories" behind them. These design theories are both structural (what part decompositions are important?) and behavioral (what details of cell chemistry/geometry/etc. are important?).

However, both kinds of systems have to explore simulation at varying levels of detail. [Hamscher 88], in particular, discusses the use of qualitative physics [Forbus 84] in diagnosing circuit boards. People who diagnose circuit boards don't do detailed simulations in their head, but ask qualitative questions like, "Is the voltage at the clock input changning in time, or not?" Thus any simulation done by a program to imitate this person should be done at a similarly qualitative level. While we have not yet begun to exploit the techniques of qualitative simulations, we expect these ideas to be very useful to writers of Vermis behavior models.

4 Conclusions

We have surveyed systems that have multiple decompositions into modules, building a theory of how to simulate them and relate the resulting simulations to each other. The resulting system, Vermis, was implemented on a Symbolics Lisp machine in Common Lisp and the Common Lisp Object System. We have applied this system to the motor nervous system of the nematodes *C. Elegans* and *Ascaris suum*, the results of which are reported in [Rockland 90].

[12] In a fashion to be defined by each such theory. That is, each theory must define not only a relation between structures and behaviors, but a hypothesis about how to observe this in the "numerical experiments" of simulation. For example, a coarse-graining of a behavior might lead one to suspect it will generate simulations that are time-averages of the more detailed behavior.

The resulting system is flexible, extensible, and modular[13], reflecting the same qualities of the underlying language and object-oriented programming extensions. It is an engine for generating intuition about theories of structure and behavior, which can then be used to drive more formal mathematical analysis.

5 Acknowledgements

This research was supported by the National Science Foundation under grant NSF FCS-8809746 and the Army Research Office under grant ARO-DAAL03-86-K-0171 (Center for Intelligent Control Systems).

The authors would like to thank Drs. Peter Doerschuk, Carl Johnson, Sanjoy Mitter, and Antony Stretton for discussions and for critical reading of the manuscript.

References

[Beardsley 89]	Beardsley, T. "Gaia: The smile remains, but the lady vanishes," *Scientific American*, December 1989, pp. 35-36.
[Carruthers 88]	Carruthers, R., *et al.* "Simulation of insect disease dynamics: an application of SERB to a rangeland ecosystem," *Simulation*, September 1988, pp. 101-109.
[Davis 84]	Davis, R. "Diagnostic Reasoning Based on Structure and Behavior," *Artificial Intelligence* 24(1):347-410, 1984.
[deKleer 86]	deKleer, J. "An Assumption-Based TMS," *Artificial Intelligence* 28(2), pp. 127 - 162, 1986.
[Forbus 84]	Forbus, K. *Qualitative Process Theory*, MIT AI Laboratory TR-789, 1984.
[Hamscher 88]	Hamscher, W. *Model-Based Troubleshooting of Digital Systems*, MIT AI Laboratory TR-1074, 1988.
[Keene 88]	Keene, S. *Object-Oriented Programming in Common LISP*, Symbolics Press, 1988.
[Kenyon 88]	Kenyon, C. "The nematode *C. elegans*," Genetics 77, pp. 71-94.
[Larkin 88]	Larkin, T., *et al.* "Simulation and object-oriented programming: the development of SERB," *Simulation*, September 1988, pp. 93-100.
[Lewin 84]	Lewin, R. Interview with Sidney Brenner on *C. Elegans*, *Science* 224:1327-1329.
[Niebur 86]	"Computer simulation of networks of electronic neurons," *Conference on Computer Simulation in Brain Science*, Copenhagen, 1986.
[Niebur 88]	Ph.D. Thesis, Institute of Theoretical Physics, University of Lausanne, Switzerland, 1988.
[Rockland 89]	Rockland, C. *The Nematode as a Model Complex System: A Program of Research*, MIT Laboratory for Information and Decision Systems WP-1865, 1989.

[13] Unlike the systems for which it was designed!

[Rockland 90]	Rockland, C. and Rowley, S. "Simulation and Analysis of Segmental Oscillator Models for Nematode Locomotion," submitted to *Journal of Experimental Biology*, 1990.
[Rowley 87]	Rowley, S., *et al.* "Joshua: Uniform Access to Heterogeneous Knowledge Structures," *Proceedings of AAAI-87*, pp. 48-52.
[Stretton 85]	Stretton, A., *et al.*, "Neural Control of behaviour in *Ascaris*", *Trends in Neuroscience 8*, pp. 294-300, 1985.
[Steele 84]	Steele, G., *et al. Common Lisp: The Language*, Digital Press, 1984.
[Sulston 83]	Sulston, J. *et al.* "The embryonic lineage of *C. elegans*," *Devel. Biol.* 100, pp. 64-119, 1983.
[Symbolics 86]	*Macsyma Reference Manual*, Symbolics, Inc., 1986.
[Symbolics 88]	*Statice Reference Manual*, Symbolics, Inc., 1988.
[Symbolics 90]	*Genera 8.0 Reference Documentation*, Symbolics, Inc., 1990.
[Walrond 85]	Walrond, J. and A. Stretton, "Excitatory and inhibitory activity in the dorsal musculature of the nematode *Ascaris*: evoked by single dorsal excitatory motoneurons," *J. Neurosci.* 5, pp. 1-8, 1985.
[Williams 87]	Williams, B. "Diagnosing Multiple Faults," *Artificial Intelligence* 32(1), pp. 97 - 130, April 1987.
[Wood 88]	Wood, W. B. (ed.), *The Nematode Caenorhabditis Elegans*, Cold Spring Harbor Laboratory, 1988.