Chapter 5: Discussion

5.1 Producing Personalities using PHISH-Nets

As stated in the introduction, one of the goals of this research is to produce an algorithm that can create characters with personality. A complete character needs more than an action selection algorithm; it also needs at least a complex model of emotions, a model of communications and social rules, and at sophisticated animation systems to show the subtleties of mood, focus of attention, and thought processes. However, at the action-selection level much can still be done to endow characters with unique and rich personalities.

In PHISH-Nets, many personality traits can be represented by creating goals that a character generally wants to accomplish. For example, a lazy character might have a goal to minimize the amount of work he has to do. This would bias all his actions towards low-effort. An energetic character, on the other hand, might instead have the goal to be reliable, which would bias his actions towards those with the least chance of failure. Although character traits are usually persistent, goals could selectively change over time, giving the illusion of learning. For example, if a character is burned his fear of fire might increase, biasing him to avoid getting too near fires in the future.

Another way to control a character's personality is through the global parameters of the algorithm. For example, a hedonistic character might have a high valued nearness-parameter. This would cause him to "live for the moment," and take advantage of situations as they arise rather than plan ahead. A fastidious character, on the other hand, would have a high conflictor-multiplier, causing him to hold on to what he has rather than take a chance by letting it go.

5.2 Extensibility

One of the primary advantages of the PHISH-Nets architecture is that creatures can be extended by creating new behaviors, without the need to specify in what way the new behaviors must interact with the old. For example, adding *Pogo-to-location* only required adding the one new behavior module, plus the graphics for the action. The author of the character does not have to specify that in order to grab the new pogo-stick object the character must use the module *Pickup-object*, nor that getting within reach of the pogo-stick would require *Walk-to-location*. Furthermore, any old or new actions will automatically use *Pogo-to-location* as an alternative to walking.

Such extensibility comes about because the compiler automatically links all possible causal links. Adding a new module requires a recompile but no more. Extensibility also comes because variable bindings are always encoded in a common base language. For example, to get to the location of the pogo-stick, *Pickup-object <pogo-stick>* passes the (x,y,z) coordinates of the location of the pogo-stick to *Walk-to-location*. *Walk-to-location* need not know about pogo-sticks at all, so long as it knows about (x,y,z) locations.

5.3 Speed of the algorithm

The speed of the algorithm depends a great deal on the kind of environment a character is in, but the order of the speed the algorithm can be determined with the help of a few assumptions. First, assume that as a network grows the number of goals, modules, and conditionals (predicates computed from sensor-values) in the network also grow but that the length of modules' precondition-lists, add-lists, and delete-lists remain constant.

Assume also that the number of precondition-lists, add-lists, and delete-lists in which a given conditional appears remains roughly constant; that is, when modules are added they tend to have new effects rather than duplicate already existing modules. Together, these two assumptions let us state that the number of links in a network grows in proportion to the number of behavior modules. Assuming that the variables in Table 6 remain constant as the network grows and given the variables shown in Table 5, one can easily show that the speed of each step of the algorithm is shown in Table 8. It follows that the overall speed of the algorithm is $O(S \times M \times B)$, where S is the length of the longest sequence of actions possible, M is the number of behavior modules in the network, and B is the maximum number of bindings (usually objects or locations) in the world. The value of B increases with the size of the world, whereas M and S increase with the complexity of the character.

Although it is dangerous to make claims of parallizability without going through the rigors of implementation, this algorithm certainly lends itself to parallelization. Each link in the network acts independently, and could be implemented on a separate processor. Such an implementation would reduce the speed of the algorithm to $O(S \times B)$. However, as is noted in [Johnson, 1991], to create a parallel version of this algorithm one would need a method by which each link could access the sensor-data and state of the world. Designing a parallel implementation of the PHISH-Nets architecture is outside the scope of this thesis.

C	number of conditionals (predicates)
M	number of behavior modules
G	number of goals (max = C)
B	max number of bindings (objects, locations, etc.)
S	length of the longest sequence of actions possible

Table 8: Variables

P	maximum number of preconditions for a module
A	maximum number of add-list entries for a module
D	maximum number of delete-list entries for a module
Cp	maximum number of preconditions a conditional appears in
Ca	maximum number of add-lists a conditional appears in
Cd	maximum number of delete-lists a conditional appears in

Table 9: Assumed constants

$Lgm = Ca \times G$	links from goals to modules
$Lmm = Cp \times Ca \times M$	inter-module links
$Lmc = Cp \times Cd \times M$	inter-module conflictor links

Table 10: Intermediary values

Initialization phase	$O(M)$
Spreading from goals to modules	$O(Lgm \times B)$ $= O(Ca \times G \times B)$ $= O(G \times B)$
Spreading between modules	$O(Lmm \times S \times B)$ $= O(Cp \times Ca \times M \times S \times B)$ $= O(M \times B)$
Conflictor-link phase	$O(Lmc \times B)$ $= O(Cp \times Cd \times M \times B)$ $= O(M \times B)$
Marking for inhibition	$O(M \times B)$
Giving extra state boost	$O(M \times B)$
Resolution phase	$O(M \times B)$
Special cases phase	$O(Lmm \times B)$ $= O(Cp \times Ca \times M \times B)$ $= O(M \times B)$

Table 11: Order of speed for each part of the algorithm

5.4 Comparison with the Agent Network Architecture

A few words should be said about the differences between PHISH-Nets and its progenitor, Maes' Agent Network Architecture. First, the PHISH-Nets algorithm addresses several of the criticisms voiced by Tyrrell that were mentioned earlier. Specifically, because it tags energy sources it does not run into problems dividing up energy. The problem of dividing energy becomes especially significant when goals can be solved in multiple ways. The PHISH-Nets architecture also has no equivalent of ANA's successor links, and therefore does not fall into the problem of attending to appetitive behaviors before consumptive ones. Another advantage over ANA is that there are very few parameters that require tweaking, and the PHISH-Nets architecture is thus easier to evaluate. In general, the parameters that do exist can be kept at the default values except for special cases. The architecture can handle much more complex sequences of actions than can Agent Networks. Finally, because behavior modules can take variables PHISH-Nets characters can easily be extended to handle new environments with existing behaviors.

The architecture also maintains many of the advantages of ANA. Because of their distributed architecture, redundancy of methods, and opportunistic behavior both systems are very robust in uncertain conditions. They can also handle multiple goals gracefully. Finally, it is easy to add new behaviors to an agent and have them integrate smoothly with old ones.

5.5 Limitations and future work

5.5.1 Lack of global knowledge

Like all behavior-based systems, the PHISH-Nets algorithm is based on the idea of producing globally reasonable actions through the local interactions of behaviors [Maes, 1993b]. These interactions should be based solely on local knowledge, without reference to a global model of the world. However, in some environments global knowledge is necessary to produce “the right” action. For example, if the wolf needs to decide whether to undo one goal in order to achieve another, it should reasonably take into account how long the goal will remain undone and whether after undoing it there will be an opportunity to redo it later. A PHISH-Nets behavior module does not have access to such information, and this limitation can cause problems such as those seen in variations 3 and 4 of experiment 3, where the wolf refuses to drop the pogo-stick to eat the pig, even though he will be able to pick up the pogo-stick soon after. The situation is analogous to a local-maximum in a hill-climbing system.

A related problem occurs when a goal has two preconditions and the satisfaction of one of these preconditions will, somewhere down the line, undo the other. For example, *pack-tnt-at-location* has two preconditions: that the wolf be carrying the TNT and that he be at the location where it should be packed. It turns out that all possible methods to satisfy *carrying-object <tnt>* will undo the precondition *im-at-location <location-arg>*. This global knowledge, as well as the knowledge of how many methods might satisfy either precondition, is unavailable to *Pack-tnt-at-location.* In simple cases, this type of problem can be avoided by hand-coding an ordering using the "preconditions-ordered?" field of the behavior module. However, one can imagine situations where the order in which two preconditions should be executed depends on the environment. The current version of PHISH-Nets cannot handle such situations, though it might be possible to add a system similar to the giveup-blacklist that finds an appropriate ordering through trial-and-error.

A special case of the latter problem occurs when a module undoes its own precondition as a side-effect of taking action. For example, *Eat-object* requires that the object be in hand, and one of the effects of this action is that the object is no longer in-hand. However, if the wolf were currently carrying the TNT and wanted to have his hands free, it would make no sense to pass energy to *Eat-object <pig>* because there is no way for him to accomplish the feat without first dropping the TNT first. In this particular implementation of the wolf the problem is solved by the transformation on the *Hands-free* element of the add-list, which transforms "blank" suggestions into the object currently being carried. This transformation limits the add-list to take effect only when the current object being carried is edible.

5.5.2 Limitations on types of actions

The most obvious limitation of the high-level behavior-specification language is that modules can only take one variable argument. For example, *plant-tnt-at-location* takes a location as an argument, but has to implicitly specify "TNT" as the object to place. It would be nice to be able to specify a module such as *plant-object-at-location <object-arg> <location-arg>*. However, such a syntax would greatly complicate the passing of arguments between modules because the order in which arguments are passed would have to be determined.

Another limitation is that the effects of an action must be context independent. For example, it is not possible to write the action *Pickup-object* such that picking up a heavy object makes the wolf tired whereas picking up a light object does not. The only context that can be encoded is in the relevant-when clause, which only states under what conditions the action will perform as expected. It would be possible to extend the algorithm to allow a relevant-when clause for each add-list and delete-list entry, or with the existing system one can create multiple actions such, as *Pickup-light-object* and *Pickup-heavy-object*, each with its own add-lists.

Finally, the "preconditions-ordered?" field only allows an author to specify whether precondition order matters or not. It does not allow more complex situations, such as having the first two preconditions unordered while the rest ordered. Currently, to achieve this kind of ordering a behavior must be broken into two different behaviors, one with each ordering. Finer control of the ordering of preconditions would be useful, for example, in the *Pogo-to-location* module. Currently *Walk-to-location* specifies that both a free path exist to a desired location and that there not exist a blocker to that location. These two preconditions are different ways of phrasing the same condition — one can never be true without the other. However, *Blow-down-object* satisfies eliminating a blocker, while *Climb-*

down-chimney satisfies being in a location where there is a free path to an object. Both conditions should be included so both kinds of solutions are available. *Pogo-to-location* only has the precondition that a blocker to the destination location does not exist, but does not have the precondition that there be a free path. This is because *Pogo-to-location* has ordered preconditions, and there is no way to specify that either action works equally well to accomplish the end goal being able to get to a location without running into a wall. Whichever precondition was listed first would always be the one the agent tried to satisfy, and the second one listed would always be spontaneously satisfied whenever the first was satisfied. Allowing more complex kinds of orderings would allow an author to supply both preconditions to *Pogo-to-location*. Complex orderings would also be an easy feature to add, since the underlying architecture of the compiled PHISH-Nets can already handle such cases.

5.5.3 Deadlines and other hidden constraints on actions

In general, PHISH-Nets do not take time-related issues into account. For example, it is impossible to encode a goal that something be done “before noon” or “at 5:30pm.” It is also not currently possible to encode how long an action might take to execute, except indirectly through the bias towards shorter action sequences and the extra energy boost for an action near completion. Similarly, the cost and reliability of an action can only be coded indirectly through goals that give energy to behaviors that use little energy, or that are known to be reliable.

5.5.4 Performing multiple actions simultaneously

While the PHISH-Nets algorithm takes multiple goals into account when selecting an action to take, it does not currently support taking multiple actions simultaneously. For example, a character cannot walk and chew gum at the same time. A first approximation of this feature would be easy to implement — a character could simply perform all actions that were not mutually exclusive, giving priority to those actions with the highest energy. However, as is pointed out in both [Loyall, 1993] and [Blumberg, 1995], it is also important to blend actions together. For example, a character walks differently when juggling than when walking alone. Furthermore, emotional states should affect how a character acts, for example by changing a characters walking gait from a slouching shuffle to a bound depending on their mood. Such extensions to the behavior algorithm will need to be integrated with the lower-level motor system. Blumberg's Hamsterdam motor system (the motor system currently used by the PHISH-Nets implementation) already supports many kinds of action-blending, so such extensions should be able to leverage off this existing work.

5.5.5 Detecting when to give up only at the last minute

One of the design decisions was to give up on an action only when all other preconditions have been satisfied. For example, in experiment number 8 where the TNT was available but the detonator was not, the wolf first placed the TNT next to the house and only then gave up on exploding the house. The underlying principle is that the world may change while other preconditions are being satisfied, making it possible to satisfy everything. More importantly, the act of satisfying one of the preconditions may make another precondition doable. However, as is seen in the above example, this heuristic can sometimes make a character look stupid if in fact the action remains impossible to accomplish: it looks as if the character only noticed that there was no detonator at the last minute. One way around this problem may be to allow the author of a character to determine whether to give up immediately or to wait until all other preconditions are satisfied, just as an author can currently mark whether the order of preconditions matter. Characters that give up immediately would seem more foresighted, while characters that only give up when there is no hope left would seem more foolish, or at least more optimistic.

5.5.6 Detecting failure and breakdowns

Although the PHISH-Nets algorithm tries to handle many situations, it cannot handle every contingency. Beyond a certain point adding more special cases to an architecture to handle bizarre situations correctly makes the architecture more brittle rather than less. The future focus of this project will be the detection and handling of failures and breakdowns, rather than always trying to get things right the first time.

Currently, failure detection is done on an ad-hoc basis: the physics layer specifically tells the character when an action has failed. It would be desirable for a character to determine for itself whether a failure has occurred. Furthermore, a character should be able to detect subtle kinds of breakdowns, such as when it is caught in an action loop or when an action is performing as expected but is not getting the character any closer to its goal. It would also be useful to have more information about a failure or breakdown in reasoning than simply “it didn’t work.”

Failures and breakdowns can be detected in at least the following ways:

- **Time-out**: Stop doing an action if it has been performed too many times in the recent past. This procedure at least guarantees that the character will try new things rather than get caught in a tight action-loop. It is the simplest of failure-handling heuristics, and therefore good when no other method can work.
- **Looping**: Stop doing an action if it always takes you back to the same state of the world. This procedure provides a better way to prevent action-loops, but it requires that the character be able to distinguish between different states of the world.

- **Known effector failure**: Stop doing an action if the low-level physics layer of your character triggers an error while performing it.
- **Unknown effector failure**: An unknown effector failure occurs when the physics layer does not report an error, but an action still does not perform as expected. The action might not have any effect at all, or it could be that no forward progress is being made. An example of the latter would be when the character walks towards a location, but does not getting any closer even though his legs are moving.
- **Undesirable, unexpected side-effects:** An action might yield the expected result, but at the same time produce unexpected side effects in addition to the expected behavior. One problem with detecting side-effects is that it is often difficult to determine which recent action caused an effect, if indeed any action was the cause.
- **Preconditions get removed out from under an action:** An action may simply never be able to make all its preconditions true at once. For example, if the TNT always rolls away from the house before the wolf can get to the detonator, perhaps he should give up on exploding the house at all, because it relies on too many uncertainties.

The scopes of a failure might also be determinable with enough domain knowledge. For example, the failure could be a known temporary problem (e.g., the character is in free-fall, so no action can work at this point in time), or a more permanent problem, such as his leg just fell off. It could also be a problem only with a particular binding (it is impossible to get to a certain location because the ground is too uneven), or with the entire behavior module (it is impossible for the character to pick anything up because his hands are too slippery).

Given any of these failure-modes, there are several different kinds of repairs a character might perform. If the failure is temporary, he might simply try again immediately, whereas with a long-term failure he might wait longer, as he does in the current system. If enough domain knowledge is present, the character may be able to set a condition that must be met before he will try again. For example, the wolf may wait until he feels stronger before he tries to blow down the twig house again. Finally, there may be a remedy that can be made to the network itself. For example, if a new side effect or precondition is found for a behavior module, that effect can be added and that segment of the network recompiled.

5.5.7 Limitations of the current implementation

The current implementation of the PHISH-Nets algorithm was designed for ease-of-modification and speed of implementation, rather than for runtime speed. It is therefore far slower than one would want in an actual interactive story system. Furthermore, the world currently must be modified with scripts rather than with a GUI interface. Both of these problems should be fixed in the next version.