Chapter 3: The PHISH-Nets Architecture

An autonomous agent or virtual character in the PHISH-Net architecture is divided into five basic units: a sensor-processor, a set of behaviors, a set of goals and their relative importance, a control unit, and actuators. The sensor-processor converts raw sensory data into high-level boolean state information such as *I'm-at-location <location-arg>*. This state information is updated every time-step. The behaviors in a system are the actions the agent can take. They are domain-dependent programs that perform relatively simple low-level actions on the environment (e.g., *Walk-to <location-arg>*). These behavior primitives are treated as black boxes and can be implemented in hardware, hand coded, or even implemented as another PHISH-Net. While behaviors in ANA take no arguments, PHISH-Net behaviors may take one argument. The control unit is the part that, given the agent's goals and the state of the sensor-predicates, decides in which of the behaviors to engage. Finally, actuators implement the chosen behavior. The PHISH-Net architecture is concerned primarily with the control unit and to some extent with the set of behaviors and goals made available to a character. The sensor-processor and the actuators are assumed, but are not dealt with by the algorithm.

3.1 High-level specification language

A character is written in a high-level behavior specification language that is later converted into a compiled representation. In this high-level specification, each action available to a character is represented by a *behavior module* which specifies when the action can be executed and what its expected results are. The character's possible high-level goals are

also encoded. These behavior modules and goals are then connected into a network at compile time by links that represent causal relations among the behaviors.

3.1.1 Behavior Specification Modules

A behavior module contains the code that executes a behavior's particular action and characterizes the effects of that action. It may take one argument. Four lists of predicates and two other fields specify a behavior:

- A *precondition list* contains predicates that must be true for the behavior to be executable.
- An *add list* contains predicates expected to become true by execution of the behavior. These are only the expected positive effects of an action, used for action selection. This list is not used by the simulated world, and the expected and actual effects of an action may differ.
- A *delete list* contains predicates which are expected to become false by execution of the behavior. These are only the expected negative effects of an action, used for action selection. This list is not used by the simulated world, and the expected and actual effects of an action may differ.
- A *relevant-when list* contains predicates which must be true for the behavior to be relevant. This indicates under what conditions a module will act as expected, as opposed to what conditions must be met for an action to be taken at all. For example, the action *eat-object* will only satisfy hunger when the object is food.[1] All conditions in a relevant-when list must be true for a behavior to be relevant.
- The *argument type* of a module specifies what type of argument the module can take. If the module takes no argument, the dummy variable “blank” is used.

1. This is the equivalent of the *context-condition* in the HAP architecture.

- The *precondition-ordered* field indicates whether it matters what order preconditions are achieved. If this boolean is true, a module will only attempt to satisfy later preconditions when earlier ones are already satisfied (i.e., preconditions will be satisfied in order).[2]

An example specification for a simple version of The Big Bad Wolf character from *The Three Little Pigs*. is shown in Tables 1-4. The complete specification for The Big Bad Wolf is given in Appendix A.

Name	Eat-object
Action	[implementation for Eat-object]
Argument Type	object
Preconditions	carrying-object <object-arg>
Preconditions-ordered?	No
Add-list	hunger-satisfied (interesting-objects-from-blank[a] <object-arg>) hands-free (carried-object-from-blank[b] <object-arg>)
Delete-list	carrying-object <object-arg>
Relevant-when	object-edible <object-arg>

Table 1: The behavior module for *Eat-object*

a.This transformation produces every object in the world, given the argument "blank."
b.This transformation produces the object currently being carried given the argument "blank."

Name	Pickup-object
Action	[implementation for Pickup-object]
Argument Type	object
Preconditions	I'm-at-location (location-of <object-arg>) hands-free (blank-from-object[a] <object-arg>)
Preconditions-ordered?	Yes
Add-list	carrying-object <object-arg>
Delete-list	hands-free (interesting-objects-from-blank <object-arg>)
Relevant-when	

Table 2: The behavior module for *Pickup-object*

a.This transformation produces the argument "blank" given any object.

2. This distinction between ordered and unordered subgoals is the same as the distinction between sequential and parallel plan nodes in the HAP architecture.

Name	Walk-to-location
Action	[implementation for Walk-to-location]
Argument Type	location
Preconditions	[none for the simple version]
Preconditions-ordered?	No
Add-list	I'm-at-location <location-arg>
Delete-list	
Relevant-when	location-on-ground <location-arg>

Table 3: The behavior module for *Walk-to-location*

Name	Drop-object
Action	[implementation for Drop-object]
Argument Type	object
Preconditions	carrying-object <object-arg>
Preconditions-ordered?	No
Add-list	hands-free (carried-object-from-blank <blank-arg>)
Delete-list	carrying-object <object-arg>
Relevant-when	

Table 4: The behavior module for *Drop-object*

A predicate in a behavior module can consist of lone predicate, such as *object-in-hand <object-arg>*, or a predicate plus a transformation, such as *I'm-at-location (location-of <object-arg>)*. A transformation is a domain dependant function that converts an argument binding into one or more different argument bindings. For example, a transformation could convert an object into the locations of that object, convert a location into the set of all objects blocking the path to that location, or convert an object into itself (i.e. no change). Lone predicates and transformations in the precondition-list must take the same argument type as their module. A precondition-predicate with a transformation must take the same input type as the output type of the transformation. Predicates that do not need arguments (such as *hands-are-empty*) take the dummy variable "blank" as an argument. Transformations for predicates in the add or delete lists of a module are the reverse of precondition-predicate transformations: add and delete list transformations must output the same type as

the module and accept as input the same type as the transformations associated predicate. For example, the predicate *hands-free* takes type "blank" as an argument. When it appears in the precondition-list of *Pickup-object*, *hands-free* is paired with a transformation that turns any object into a blank. On the other hand, when it appears in the add-list of *Drop-object*, *hands-free* is paired with a transformation that turns a blank argument into the object currently being carried.

3.1.2 Goals

Goals consist of a strength value and predicate to make true. For example, the goal "hunger" might be represented by the precondition *hunger-is-satisfied* with a strength of 50. Goals always have a base argument type of "blank," though their predicate can have transformations just like preconditions. For example, *carrying-object (blank-to-pogo-stick-transformation <blank-arg>)* would represent the desire to be holding a pogo-stick. The strength of a goal can change over time as an agent's internal state and environment change.[3] At runtime, goals feed activation energy to modules that best satisfy them by a mechanism explained in Section 3.4.

Name	Hunger
Argument Type	blank[a]
Predicate-to-make-true	hunger-satisfied
Goal Value	22

Table 5: The specification for the goal *Hunger*

a.Goals never take arguments, so this field is always of type "blank"

3. For example, a character's hunger might decrease because he has just eaten (a change in the environment) or because he has gotten distracted (a change in the character's internal state). Changes due to the environment would be performed by the world's physics layer; changes due to the character's internal state would be performed by the implementation of individual actions.

3.1.3 Variables vs. indexical functional aspects

ANA does not incorporate classical variables and variable-passing, and Maes [Maes, 1989] argues that many of the algorithm's advantages would disappear if variable-passing was introduced. Maes suggests that *indexical-functional aspects* [Agre, 1987] be used instead of classical variables. The idea is that behaviors are specified relative to the current situation of the agent rather than to absolute identities of objects. For example, instead of specifying a behavior as *Goto-location (x,y),* one would specify the location in terms of the functional aspects desired, as in *Go-to-the-nearest-exit-from-the-room*.

Indexical-functional aspects reduce the problem space by differentiating only those objects, locations, and states that have unique and relevant functions. However, in complex environments the number of unique, relevant functions is usually large. For example, a tool-using agent needs more differentiation than *Get-the-nearest-tool*, because the action *Pound-a-nail* requires a hammer, whereas *Drive-a-screw* requires a screwdriver. Furthermore, even if several real-world objects could be conflated into a single function, extensions to the agent could later require a reworking of these groupings [Rhodes, 1995].

Behaviors in PHISH-Nets use a mixture of indexical-functional aspects and classical variable-passing. Each behavior may take a single argument of a given type, such as "object" or "location." The set of all possible arguments is kept manageable by only specifying those that are "interesting." For example, in the current implementation of The Big Bad Wolf, the only interesting locations are special locations, the locations of objects, and the locations of other characters. However, after a location is declared interesting it is still encoded by (X, Y, Z) coordinates, rather than by a functional aspect such as "location of screwdriver."

3.2 Compiled PHISH-Nets

3.2.1 Modules and links

The high-level specification for a character is compiled into a PHISH-Net, which is the actual representation used at runtime to determine a character's actions. Compiled PHISH-Nets are made up of the following data types:

- **Goals**: Like goals in the high-level specification, goals in a PHISH-Net indicates the conditions a character is trying to satisfy in the world. For example, the goal to be carrying a pogo-stick encourages the character to act in ways that will make the condition *I'm-carrying-pogo-stick* become true. Each goal has a value indicating how important the goal is. Each goal also has its own list of predecessor-links from that goal, and a list of conflictor-links from that goal. Predecessor links connect a goal to behavior modules that might satisfy that goal. Conflictor links connect a goal to behavior modules that might undo that goal once it is satisfied. Both kinds of links are described in detail below.

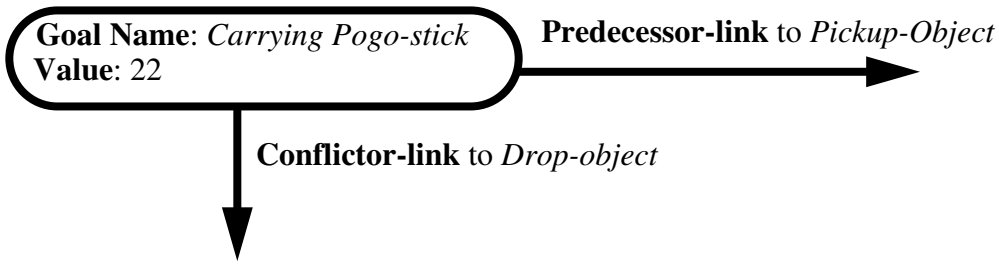

Figure 2: The goal to be carrying a pogo-stick

- **Behavior Modules**: Behavior modules (often referred to simply as "modules") represent an uninstantiated action such as *Pickup-object.* Compiled behavior modules consist of a list of suggestions for and against particular

bindings, a list of predecessor-links and conflictor-links, and the effector for this action (the code that is executed when the action is chosen). Modules also contain three blacklists indicating which bindings are currently considered to be "failures," "given-up on," and "inhibited." Blacklists are simply lists of bindings that should be allowed to pass to a module (in the case of the failure-blacklist and giveup-blacklist) or be allowed to pass from a module (in the case of the inhibited-blacklist). When a module's action is taken and fails to achieve the effect expected, the binding for that action is added to the failure-blacklist for that module, along with a number indicating how many time-steps before the action can be tried again. A similar mechanism for adding bindings to the giveup-blacklist and inhibited-blacklist is described in Section 3.6.

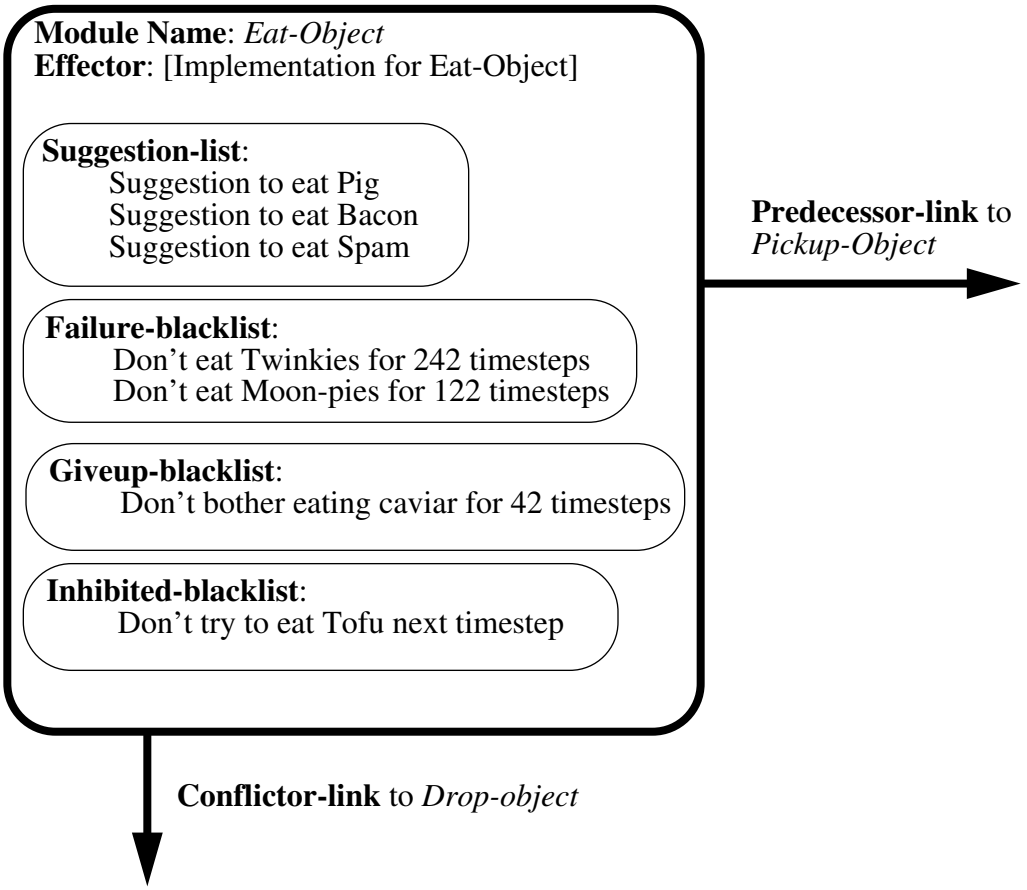

Figure 3: The behavior module *Eat-Object*

- **Bindings**: A binding is an argument with which a behavior module can be instantiated. For example, *pickup-object* can be instantiated with the binding <*pogo-stick*>. A behavior module paired with a binding is called an instantiated behavior, or just a behavior**.** Bindings are tagged by type, but there is no typechecking in the current implementation.

Figure 4: The binding for a pogo-stick

- **Suggestions**: Suggestions consist of a behavior module and binding that is being suggested, the goal that is the original source of this suggestion, a "hop-count" counter indicating the suggestion's distance from the original suggesting goal and a field indicating whether the suggestion is in favor of or against the instantiated behavior being executed. If a suggestion is in favor of the behavior being taken, it is called a "supportive" suggestion. If it is opposed to the behavior, it is called "inhibitory." As a rule, supportive suggestions are passed down predecessor-links and inhibitory suggestions are passed down conflictor-links. During each initialization phase of the algorithm, suggestions are created by goals and passed down links to behavior modules. When a goal creates a suggestion the suggested-module and suggested-binding fields are both labeled "blank," but these are set later by transformations in the links that pass suggestions to behavior modules.

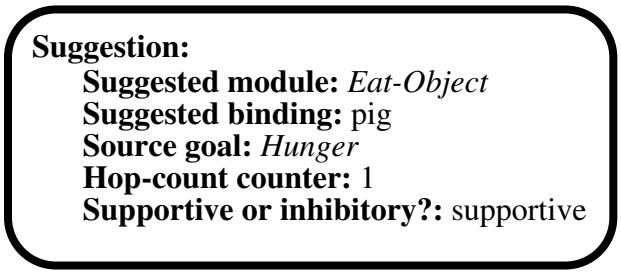

Figure 5: The suggestion to eat the pig

- **Links**: Links are the conduit through which suggestions are passed among goals and behavior modules. Links have two ends. The front end can be either a goal or a behavior module; the back end is a behavior module. Between the two ends is a series of filters and transformations that convert input suggestions to output suggestions. For example, if a character wanted to pick up a pogo-stick, the filter-series for the link from *Pickup-object* to *Walk-to-location* would take the pogo-stick suggestion as input and would output all the locations in which pogo-sticks can be found. The filter-series is explained in more detail later.

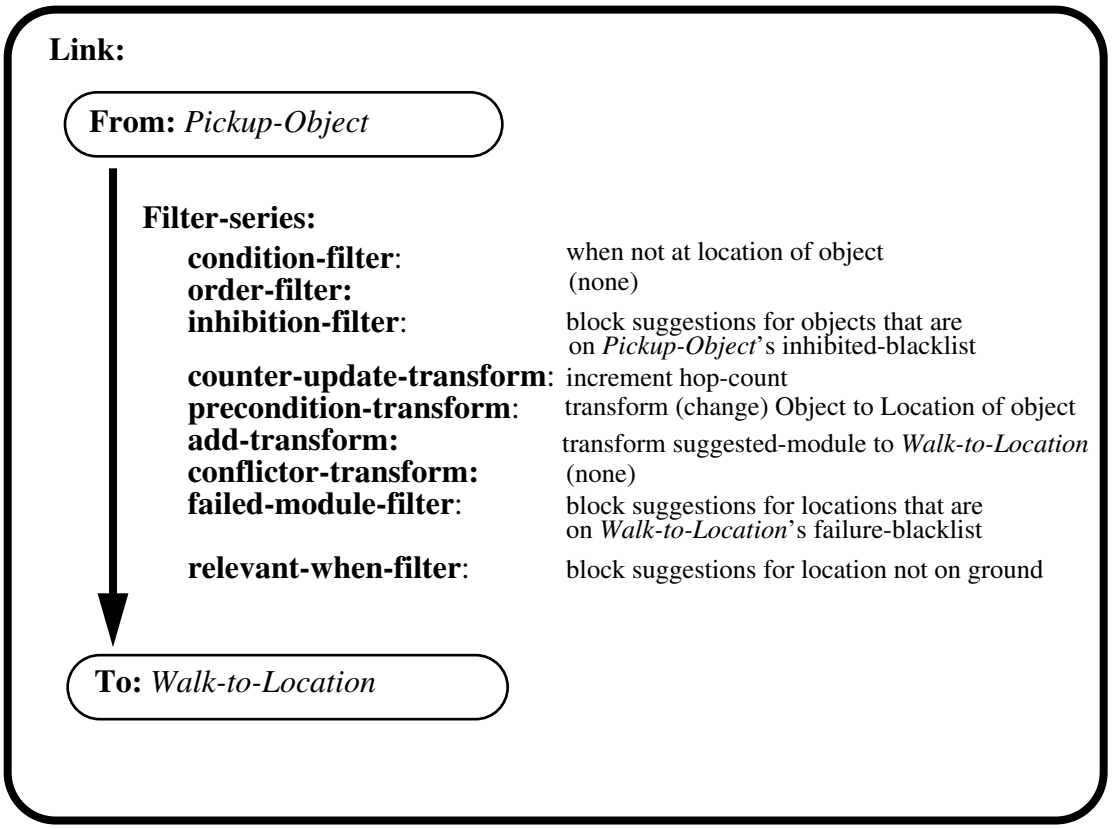

Figure 6: The predecessor link between *Pickup-Object* and *Walk-to-Location*

At compile-time, predecessor links are created from every goal to every behavior module that can satisfy it, and from every behavior module to every behavior module that can make true one of the from-module's preconditions. Conflictor links are created from every goal to every behavior module that would undo that goal once it was satisfied, and from every behavior module to every module that can undo one of the from-module's preconditions. These links represent causal relationships among the behaviors. For example, a predecessor link would exist from *Pickup-object* to *Walk-to-location* because the latter satisfies *I'm-at-location*, which is a precondition of the former. There would also be a link between the goal *Hunger* and *Eat-Object*, because the latter satisfies the former, and a conflictor-link between *Eat-Object* and *Drop-Object*, because dropping an object undoes a precondition necessary for eating that object.

More formally, there is a **predecessor-link** between a goal and a module when the condition associated with the goal is in the behavior module's *add-list*. There is a predecessor-link between two modules every time the from-module contains a predicate in its *precondition-list* which matches a predicate in the to-module's *add-list*. Similarly, there exists a **conflictor-link** between a goal and a module when the condition associated with the goal is in the to-module's *delete-list*, and between two modules every time a predicate in the from-module's *precondition-list* matches one in the to-module's *delete-list*. Conflictor links and predecessor-links differ only in their conflictor-transform and condition-filter, described below.

Suggestions are passed from one module (the "from-module") to another module (the "to-module") through the following series of filters and transformations. When a filter receives a suggestion, it either passes that suggestion on unchanged to the next filter or transformation in the series, or it "blocks" that suggestion and does not let it pass. For example, a filter might block all suggestions to act with bindings that had failed recently.

Transformations, on the other hand, modify suggestions they receive into different suggestions. Transformations can also convert a single suggestion into multiple suggestions. For example, a transformation might convert a suggestion to act with a particular object into several suggestions, one for each location in which that object might be found. Transformations and filters pass their processed suggestions to the next filter or transformation in the series, in a bucket-brigade fashion. The first filter in the series receives all suggestions contained in the from-module's suggestion-list. The last filter in the series passes all the suggestions it doesn't block to the to-module, which adds those suggestions to its suggestion-list. Each filter and transformation in the filter series modifies suggestions such that only suggestions that are relevant in the character's current situation are passed through a link.

A filter series consists of the following filters and transformation, in order:

- **condition-filter**: The condition-filter blocks all suggestions for which the from-module's preconditions are already satisfied. Blocked suggestions are not passed to filters later in the series and do not affect the suggestion-list of the to-module. This filter insures that a module not try to make true a precondition that is already true. In conflictor-links, this filter instead blocks all suggestions that are *not* satisfied, because only satisfied conditions should be able to inhibit.
- **order-filter**: The order-filter blocks all suggestions for which earlier preconditions in the suggesting module's ordered precondition-list are not yet satisfied. This filter prevents a module from attempting to satisfy its preconditions out of order. If order does not matter for this particular module, the order-filter passes all suggestions through unchanged.

- **inhibition-filter**: This blocks removes all suggestions for bindings that were inhibited in the previous time-step. As described later, inhibited bindings are not passed.
- **counter-update-transform**: This transformation increments the hop-counter: the counter in a suggestion that indicates how many steps away from the original goal the suggestion is. As is explained later, the hop-counter is used to slightly bias the algorithm towards shorter action sequences.
- **precondition-transformation**: This transformation changes a suggestion from one type (e.g., "object") to another (e.g., "location"). The transformation to use (if any) is specified in the suggesting module's precondition. For example, the transformation for the first precondition in *pickup-object* takes an object and returns the location of that object. Some transformations produce several outputs for the same input. For example, the *location-of <object-arg>* transformation might return several locations if the object is especially large, or if there are several copies of the same object.
- **add-transformation**: This is the same as the precondition-transformation, except that it is specified in the receiving-module's add-list (delete-list for conflictor-links). The add-transformation also changes the suggested-module field of each suggestion from the "from-module" to the "to-module" for this link.
- **conflictor-transform**: If in a predecessor-link, this filter simply passes a binding on, unchanged. In conflictor-links it changes a suggestion in favor of executing with an argument into a suggestion opposed to executing with that argument. In this way, goals can inhibit actions with which they clash, and modules can inhibit actions that would make their true preconditions become false.

- **failed-module-filter**: This filter removes all bindings for which the receiving module has previously failed or otherwise given up. Failure of and giving up on modules is discussed in Section 3.6.
- **relevant-when-filter**: This filter removes all bindings for which the receiving module is not applicable. Applicability is determined from the receiving module's relevant-when field. For example, *eat-object* is only relevant when the object in question is a kind of food.

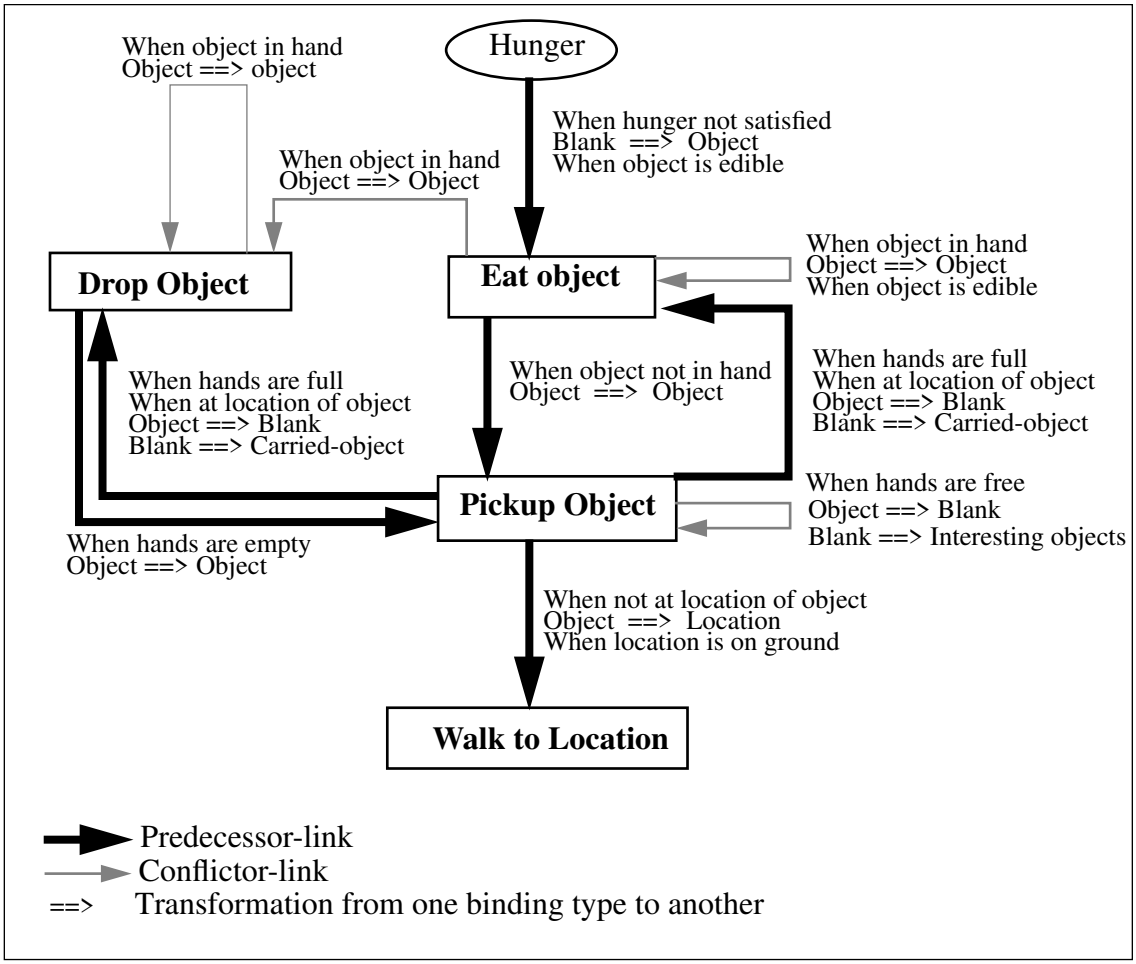

Figure 7: The simple wolf as a compiled PHISH-Net

3.2.2 An example link

Assume *Pickup-object* already has several suggestions of objects to pick up in its suggestion-list. These suggestions would either have come from other behavior modules (via those module's own predecessor links) or directly from goals. *Pickup-object* passes all these suggestions to its own predecessor and conflictor-links, including the link from *Pickup-object* to *Walk-to-location*. In this particular link, the condition-filter blocks suggestions to pick up objects already in the current location of the agent, and passes on the rest of the suggestions. The order in which *Pickup-object*'s preconditions are satisfied matters, but because *I'm-at-location (location-of <object>)* is the first precondition of *Pickup-object*, the order-filter passes all the suggestions through. Then the transformation-filter changes all the suggestion's bindings from objects (the object to be picked up) into the (x,y,z) coordinates of the location that is to be walked to (the location of that object). If an object is in multiple locations (either because there are several copies or because the object is especially large), this transformation will pass on suggestions for several different locations for every suggestion for an object the transformation receives. After this point in the series, all subsequent filters and transforms will receive suggestions for locations that should be walked to, not suggestions for objects that should be picked up. The add-list for *Walk-to-location* contains no transformations, so it passes suggestions unchanged. The link between *Pickup-object* and *Walk-to-location* is not a conflictor-link, so the conflictor-transform also passes suggestions unchanged. The failed-module-filter then blocks all suggestions for locations that *Walk-to-location* has previously failed (that is, those bindings still on *Walk-to-location*'s blacklist), and passes on the rest. Finally, the relevant-when-filter blocks all locations not at ground-level, since *Walk-to-location* does not work above ground. The remaining suggestions are added to *Walk-to-location*'s suggestion-list. As

will be discussed in Section 3.4, *Walk-to-location* will then repeat the process by passing these new suggestions down its own predecessor and conflictor-links.

3.2.3 Tagging suggestions with source information

Suggestions need more information than just argument bindings: they also need to communicate the importance of a suggestion. Maes' Agent Networks Architecture communicates the importance of an action by passing activation energy from goals and state variables to modules and from module to module. However, as Tyrrell points out in [Tyrrell, 1994], there is not enough information maintained by ANA to manipulate energy properly.

Briefly stated, the problem is whether to divide activation energy evenly among all modules that can satisfy a precondition, or to give the full amount to each. If activation energy is divided evenly among all modules that satisfy a precondition, modules with preconditions that many other modules can satisfy are unfairly disadvantaged. For example, if a character was very hungry and slightly thirsty but had 50 different behaviors that could satisfy that hunger, each of those behaviors would get only 1/50th of the energy from hunger. This would cause the character to try to get a drink, even though thirst is only a secondary goal. If, on the other hand, energy is given to each module without dividing by the number of modules receiving that energy, modules that are unique in satisfying a goal are unfairly advantaged. Take, for example, a character that is more hungry than thirsty and has the choice of 50 different drinks from a vending machine. With this method for dividing energy, each behavior representing buying a different drink would get the full amount supplied by thirst. However, if using the vending machine to buy any of these drinks required exact change then all 50 modules would give the full amount of energy to "get change for vending machine." The act of getting change for the vending machine would

become an all-consuming passion for the character, even though thirst is still only a secondary goal to hunger.

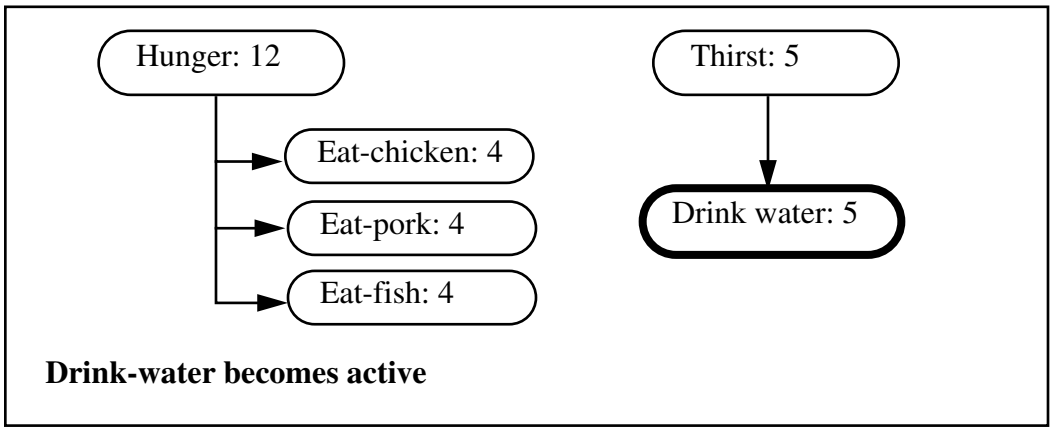

Figure 8: ANA when activation-energy is divided evenly among recipients

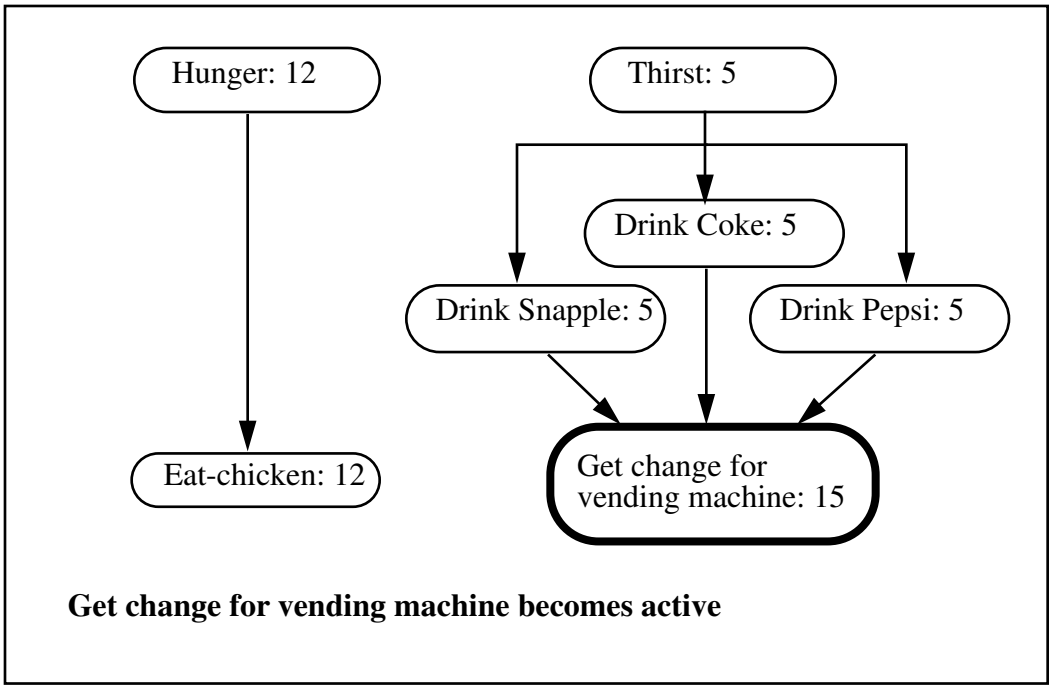

Figure 9: ANA when full activation-energy is given to each recipient

The desirable result is for energy to be divided such that actions are favored relative to the number and strength of the goals they help satisfy, regardless of the paths that connect

actions to goals. For these reasons, in PHISH-Nets every suggestion includes a tag indicating the goal that originally spawned it, rather than an explicit energy value. Energy is only computed when necessary based on these tags, rather than stored as a number. The *base energy* of a suggestion is given by:

$$\text{base energy} = \begin{pmatrix} G \times S^H, \text{ when suggestion is supportive} \\ -G \times C \times S^H, \text{ when suggestion is inhibitory} \end{pmatrix}$$

where G is the source goal's strength value, H is the number of hops between the goal and this suggestion, S is the *sequence-length* parameter (default equals 0.99), and C is the *conflictor-multiplier* parameter (default equals 0.90). These last two parameters are described in Section 3.3.

When S is significantly less than 1.0, the multiplication by H^S causes shorter sequences of actions to be favored. When the sequence-length parameter is close to 1.0 the bias caused by this multiplication is completely overpowered whenever competing suggestions have different goal strengths (G), but still has an effect when two suggestions both have the same source goal, or when their goal sources have the same strength. This bias helps avoid situations where the character chooses a much longer sequence of actions to achieve a goal than necessary.

3.3 Adjustable Parameters

The following parameters are available to adjust the functioning of the algorithm. The use of these parameters will be further described in Section 3.4.

- **Sequence-length-multiplier** (default = 0.99): A suggestion's base energy is multiplied by the sequence-length-multiplier raised to the power of the

number of hops away that suggestion is from its source goal. The closer to 0.0, the more bias there is towards shorter action sequences.

- **Nearness-multiplier** (default = 10): When a behavior module receives an extra boost of energy this nearness-energy ranges between 0.0 and 1.0, times this parameter. A high value of the nearness-multiplier makes the algorithm favor actions that are closer to being accomplished; e.g., going to closer locations.
- **Maximum-number-of-hops** (default = 20): This is the maximum number of steps a suggestion can be away from its goal source before being deleted. This limit puts an upper-bound on the runtime of the algorithm. The default is 20, a value far longer than the length of the longest action sequence in any of the experiments performed.
- **Conflictor-multiplier** (default = 0.9): This parameter indicates the amount by which inhibitory suggestions are multiplied. If it is set to 1.0, one action can keep another of equal energy from acting through inhibition. Lower values give a character more freedom to undo a satisfied goal or precondition if it might mean achieving another. In environments where undoing a satisfied precondition may mean never satisfying it again, this value should be high; lower values allow more dithering and experimentation. However, a value less than 1.0 also opens the possibility of dithering between two near-equal goals. Examples of both possibilities are shown in Section 4.3 and Section 4.4.
- **Failure-penalty-time** (default = 400): The number of timesteps a binding stays on a behavior module's failure blacklist before it can be tried again. If the world is especially dynamic, this parameter should be lower, in order that the character will try again more quickly. The appropriate value for this parameter also depends on how long a typical action takes in the domain world.

- **Giveup-penalty-time** (default = 80): The number of timesteps a binding stays on a behavior module's giveup blacklist before that binding can be tried again. This parameter should be a high enough value that the character will change his environment before retrying, to avoid getting into action loops.
- **Inhibition-penalty-time** (default = 1): The number of timesteps a binding stays on a behavior module's inhibition blacklist before it can be tried again. This should always be set to 1, and is included here simply for completeness.

Parameter	Default Value
Sequence-length-parameter	0.99
Nearness-multiplier	10
Maximum-number-of-hops	20
Conflictor-multiplier	0.90
Failure-penalty-time	400
Giveup-penalty-time	80
Inhibition-penalty-time	1

Figure 10: Default values for adjustable parameters

3.4 The action-selection loop

Every time-step the following actions are taken to determine what a character should do:

3.4.1 Initialization phase

1. All suggestions from the previous time-step are erased.[4]

4. The only state carried over from one time-step to the next is the state of each module's failure-blacklist, giveup-blacklist, and inhibited-blacklist. Every timestep, a character will re-evaluate what action to take based entirely on the values in these blacklists and the state of the world without any other memory of previous history.

2. The failure-blacklist, giveup-blacklist, and inhibited-blacklist for every module is checked, and bindings that have been on the blacklist for their allotted "penalty time" are removed from the list so they can be tried again.
3. Goals generate blank suggestions that are passed down the predecessor-links that connecting the goal to behavior modules.[5] The suggestions generated by goals do not specify a suggested-binding or suggested-module; these fields in the suggestion are set to "blank," and will be set later by the precondition-transformation and add-transformation filter of the link from the goal. For example, the goal *Hunger* would create a blank suggestion to pass down its predecessor link to *Eat-Object*. The add-transformation of this link puts *Eat-object* into the module-suggested field of the suggestion. It also passes the suggestion through the transformation "object-from-blank," which is the transformation specified in the add-list for *Eat-Object*. This transformation converts the suggestion for a blank binding into several suggestions, one for each interesting object currently in the environment. At the end of the filter series for this link, the relevant-when-filter will block all but suggestions for edible objects. In this way *Eat-object*'s suggestion-list becomes populated with suggestions to eat every edible object in the environment.
4. At the end of the initialization phase, modules that received new suggestions are marked as "dirty."

3.4.2 Module-passing phase

The module-passing phase is as a loop in which modules pass suggestions among themselves down their predecessor-links:

5. Until all modules in the network are marked "clean":

5. Suggestions are not passed down conflictor-links until the conflictor-link phase, described below.

- For each dirty module pass all suggestions in the module's suggestion-list down all the module's predecessor links. The filter-series in each link will insure that only suggestions for actions that are appropriate to the current situation are passed on to the to-module of a link. Specifically, the condition-filter of a link blocks suggestions for bindings for which the from-module is already executable. The order-filter keeps a link from passing suggestions if preconditions before this link are not yet satisfied and order matters. The inhibition-filter blocks bindings for which the from-module is being inhibited. The precondition and add-transforms convert a suggestion into a type the to-module can understand— e.g., from type "object" to type "location." Finally, the failed-module-filter and relevant-when-filter insure that only suggestions for relevant bindings that haven't failed recently are passed to the to-module.
- Modules that have received suggestions from links add these suggestions to their suggestion-list, with the following exceptions. If a module receives a suggestion that is a duplicate to one already in the module's suggestion-list[6], and if the new copy has a lower hop-count than the old, then the new copy is added to the suggestion-list and the old copy is deleted from the list.[7] Otherwise, the new copy is not added to the suggestion-list. For efficiency, suggestions with a base energy between 0.1 and -0.1 are also not added, nor are suggestions that have a hop-count greater than the value of the parameter *maximum-number-of-hops* (default equals 20 hops).
- When a receiving module adds a suggestion to its suggestion-list, the module is marked "dirty."

6. Two suggestions are defined as identical if they both have the same goal, suggested-module, and suggested-binding. Their hop-counts and whether the two suggestion are supportive or inhibitory are ignored.
7. Note that an inhibitory suggestion can replace a non-inhibitory if the former has a lower hop-count than the latter. However, an instantiated behavior cannot support or inhibit itself because any self-supporting or self-inhibiting suggestion will be a copy with a larger hop-count, and thus will be removed.

- Once a module passes all its suggestions down its predecessor-links, the module is marked "clean."

Dirty modules continue to be serviced until all modules are clean.

3.4.3 Conflictor link phase

In the conflictor link phase, all modules and goals pass their suggestions down their conflictor links. Unlike the module-passing phase, there is no loop because conflictor-links do not propagate more than one hop. The following events happen in the conflictor link phase:

6. Each goal creates a new blank suggestion and passes it down all the goal's conflictor-links. Note that all suggestions passed down conflictor-links are set to be inhibitory suggestions by the conflictor-transform of the link. These suggestions therefore have a negative base energy.
7. Each module passes all supportive suggestions in the module's suggestion-list down all the module's conflictor links. Like the suggestions passed down conflictor-links from goals, these suggestions are turned into inhibitory suggestions and as a result have a negative base energy.
8. The base energies of all suggestions for and against each particular instantiated behavior are summed. This sum is the base energy of the entire instantiated behavior. Two subtotals are also maintained: the sum of the base energy of all supportive suggestions for an instantiated behavior is called the supportive base energy of the behavior. Similarly, the sum of the base energy of all inhibitory suggestions for an instantiated behavior is called the inhibitory base energy of the behavior.
9. Any instantiated behavior whose base energy is negative[8] is declared *inhibited* and

is added to the inhibited-blacklist for the module associated with the behavior. Instantiated behaviors added to an inhibited-blacklist will remain on the list for the number of timesteps indicated in the *inhibition-penalty-time* parameter, which defaults to one timestep. Any instantiated-behavior on the inhibited-blacklist is marked as not executable this timestep. Furthermore, the inhibited-filter in every link's filter-series will block any suggestion that contains a binding on the from-module's inhibited-blacklist, and therefore suggestions for inhibited bindings cannot be passed to other modules.[9]

3.4.4 Resolution phase

In the resolution phase the action the character should take this timestep is determined. The following steps occur in this phase:

10. All instantiated behaviors that are not executable are removed from consideration. By definition, an instantiated behavior is executable if all its preconditions are true for the binding with which it is instantiated, and if that behavior's binding is also not on the module's inhibited-blacklist.
11. Each executable instantiated behavior is given an extra boost of energy associated with how close to completion the action is. For example, each executable binding in *Walk-to-location* would be given extra energy proportional to how close the location is. Actions that are always close to completion, such as *Drop-object*, always receive the maximum boost when executable. This extra energy is called the nearness-

8. I.e., more important goals or a larger number of goals are inhibiting the instantiated behavior than are supporting it

9. Note the difference in the handling of the inhibited blacklist from the failure-blacklist and the giveup-blacklist. If an instantiated behavior is on the inhibited-blacklist of a module, suggestions for that behavior cannot be passed from that module. On the other hand, if an instantiated behavior is on the failure or giveup-blacklists of a module, suggestions for that behavior may not be *received* by that module. These limitations are handled automatically by the inhibition-filter and the failed-module-filter, respectively.

energy of an instantiated behavior.

12. The total energy of each executable instantiated behavior is computed by summing the behavior's base energy and nearness-energy.
13. The instantiated behavior with the highest total energy is chosen as the action to take, and the effector for the behavior module is executed with the binding indicated in the instantiated behavior.

3.4.5 Special cases phase

There are several kinds of situations a character can get into that require special handling. The steps taken to handle these situations are described in detail in Section 3.6 and Section 3.7, but briefly these steps are as follows:

14. Executable instantiated behaviors are called "passed-over" if their supportive base energy plus nearness energy is higher than the supportive base energy plus nearness energy of the instantiated behavior that was chosen for execution. Passed-over behaviors would have been chosen as the action to execute, were it not for inhibition bringing their total base energy down. Of all the passed-over executable instantiated behaviors, the behavior with the largest amount of inhibition is placed on the inhibited-blacklist for its associated module. Because only one behavior per time-step added to the inhibited-blacklist in this fashion, this step circumvents the situation where two high-energy behaviors mutually inhibit each other, thereby allowing a lower energy behavior to be chosen. An example of this situation is given in Section 4.5.[10]

10. A more subtle effect of this step is to insure that a character will give up on taking an action whose preconditions cannot be satisfied due to inhibition. Passed-over behaviors are likely to never get executed, and therefore non-executable behaviors need to know to stop relying on the passed over behaviors for making preconditions become true. This is handled by the give up step, described below and in Section 3.6.

15. If the instantiated behavior that just executed is judged to be a "failure," the binding of that behavior is added to the failed module's failed-blacklist. The binding will stay on the failure-blacklist for the number of timesteps specified in the *failure-time-penalty*, which defaults to 400 timesteps. This step of the algorithm forces a creature to avoid behaviors known to fail. Currently failures are detected on an ad-hoc bases by the physics layer of the implementation rather than by the creature himself. Extensions that would give a character greater flexibility in detecting failures are discussed in Section 5.5.6.
16. If a non-executable instantiated behavior cannot satisfy any of its remaining false preconditions, it is called an unsatisfiable behavior. A behavior might not be able to satisfy its preconditions because the modules that would satisfy them are unavailable, have failed, or are being inhibited to the extent that they will never be chosen for execution. By definition, an instantiated behavior cannot satisfy a precondition if a suggestion for the behavior's associated binding cannot pass through the filter-series of any predecessor-link from the associated module. Bindings associated with an unsatisfiable instantiated behavior are added to the associated module's giveup-blacklist. This step of the algorithm forces a creature to avoid trying to satisfy preconditions for behaviors that are hopeless. This kind of situation is discussed further in Section 3.6.
17. In some cases the resolution phase will result in no executable instantiated behavior being available. This condition may exist simply because no behaviors are currently executable at all, or it may be indicative of a failure in the algorithm. To avoid the latter condition, if no action is selected by the resolution phase then the highest energy instantiated behavior that is actively inhibiting another instantiated behavior is added to the giveup-blacklist of the inhibiting behavior's associated module. If no

such instantiated behavior exists (i.e., no instantiated behaviors are currently inhibiting) then the giveup-blacklists for all modules are cleared. This step of the algorithm is designed to get a character out of a few special degenerate cases described in Section 3.7.

3.5 A simple example

As a simple example consider the network implementing The Big Bad Wolf, as shown in Figure 7 on page 40. The wolf is created from the four behavior modules: *Eat-object*, *Pickup-object*, *Walk-to-location*, and *Drop-object*; and from the one goal *Hunger* which has a value of 22. In this example, the world contains the wolf's favorite food, the little pigs Moe, Larry, and Curly, and also contains a pogo-stick. For this example, Moe is at the (x,y,z) location [10, 10, 0], Larry is at [20, 20, 0], and Curly is at [30, 30, 0]. The wolf is lurking nearest to Curly at [35, 35, 0], and he's hungry.[11]

The main algorithm runs as a continuous loop, producing the "best choice action" every time-step. Following the steps outlined in Section 3.4, we have the following phases and steps of the algorithm:

3.5.1 Example initialization phase

1. All suggestions from previous time steps are erased.
2. All blacklists are checked for bindings that should be removed. Since in this example no bindings are currently on blacklists, no change is made.
3. The goal *Hunger* creates a suggestion with the suggested-module and suggested-binding fields both set to "blank," the source-goal field set to "*Hunger*," the hop-count counter set to zero, and the supportive-or-inhibitory tag set to "supportive."

11. Suspenseful, isn't it?

This suggestion gets passed down *Hunger*'s only predecessor-link, to *Eat-object*. Hunger is not currently satisfied, so the link's condition-filter lets the suggestion through. The order-filter and inhibition-filter don't matter for links from goals, so these filters also let the suggestion through unchanged. Then the counter-update-transform increments the suggestion's hop-count counter from zero to one. Because the specification for *Huger* contains no transformation, the precondition-transformation of the filter-series passes the suggestion through unchanged. However, the add-transformation makes two changes to the suggestion. First, it changes the suggestion's suggested-module field from blank to *Eat-Object*. Second, it passes the suggestion through the transformation described in the add-list for the high-level specification of *Eat-Object*. In this case, that transformation is *Interesting-objects-from-blank*. Given the one suggestion with the blank suggested-binding field, this transformation produces a suggestion for each object in the world: one suggestion with the suggested binding field set to "Moe," one set to "Larry," one set to "Curly," and one set to "pogo-stick." All four new suggestions are passed on to the next filter in the filter-series, the failed-module-filter. Because no actions have failed yet in this example, the failed-module-filter passes all four suggestions on unchanged. Finally, the relevant-when-filter blocks the suggestion to eat the pogo-stick, because pogo-sticks aren't edible. The other three suggestions are allowed to pass out of the filter-series, and all three are added to the suggestion-list of *Eat-object.*

4. New suggestions have been added to *Eat-object*, so the module is marked "dirty."

3.5.2 Example module-passing phase

5. The following actions occur in the module-passing phase loop:

- Since *Eat-object* is dirty, it passes the three suggestions in its suggestion-list down its one predecessor-link to *Pickup-object*. The filter-series increment the three suggestions' hop-count counter and changes their suggested-module to *Pickup-object*, but otherwise allow the suggestions to pass through unchanged.
- Since *Pickup-object* currently has no suggestions in its suggestion-list and none of the suggestions' hop-counter is greater than 20, all three suggestions are added to *Pickup-object*'s suggestion-list.
- *Pickup-object* is marked "dirty."
- *Eat-object* is marked "clean."
- The loop repeats, and this time *Pickup-object* is dirty. All three suggestions are passed down all three predecessor-links leading from *Pickup-object*: the link to *Walk-to-location*, the link to *Drop-object*, and the link to *Eat-object*. In the links to *Drop-object* and to *Eat-object*, all three suggestions are blocked by the condition-filter, because the wolf's hands are already empty.[12] However, none of the suggestions are blocked in the link from *Pick-up-object* to *Walk-to-location*. In this link, the filter-series converts the three suggestions with the object bindings "Moe," "Larry," and "Curly" into suggestions with the location bindings "[10, 10, 0]," "[20, 20, 0]," and "[30, 30, 0]," the locations of the three pigs. It also changes their suggested-module field from *Eat-object* to *Pickup-object*., and increments their hop-counters from one to two.
- *Walk-to-location* is marked "dirty."
- *Pickup-object* is marked "clean."

12. Were the wolf's hands full, the suggestions would still be blocked by the order-filter, because the wolf is not yet at the location of any of the objects, and the order in which *Pickup-object*'s preconditions are satisfied matters.

- Again the loop repeats, and this time *Walk-to-location* is dirty. However, in this simple example *Walk-to-location* has no predecessor-links. The module is therefore is automatically marked "clean." Since no module is still dirty, the loop terminates and the module-passing phase is complete.

3.5.3 Example conflictor-link phase

6. *Hunger* creates a new blank suggestion, but because it has no conflictor-links the suggestion does not get passed.
7. *Eat-object*, *Pickup-object*, and *Walk-to-location* all have supportive suggestions in their suggestion-lists, and pass these suggestions down their conflictor-links. *Walk-to-location* has no conflictor-links, and therefore does nothing. *Eat-object* has two conflictor-links, one to *Drop-object* and one back to *Eat-object*. Because none of the three pigs are currently in-hand, all three suggestions are blocked by the condition-filter of both these links. *Pickup-object* passes the three suggestions in its suggestion-list down its single conflictor-link, which leads back to Pickup-object.[13] The wolf's hands are free, so this links conflictor-link passes each suggestion unhindered, and the counter-update-transform increments their hop-count from two to three. The precondition-transformation for this link changes the suggested-binding of each of the three suggestions from a pig into "blank." The add-transformation then converts each of the three "blank" suggestions into four separate suggestions, one for each "interesting" object in the world[14]. At this point, there are

13. Note that no module can ever pass a suggestion to itself without changing the suggestion's suggested-binding field, because the suggestion will automatically be a duplicate with a longer hop-count, and will therefore be deleted. For example, no suggestion will ever make it down the conflictor link from *Eat-object* to itself, because the suggested-binding is not transformed. It is possible, however, for a suggestion supporting picking up one object in *Pickup-object* to lead to an inhibitory suggestion to not pick up any other objects.
14. The add filter also converts the suggested-module field from the from-module to the to-module, but in this case these are both the same module.

12 suggestions being fed into the conflictor-transformation: three supporting the picking up of "Moe," three supporting "Larry," three supporting "Curly," and three supporting picking up the pogo-stick. The conflictor-transformation turns all 12 of these supportive suggestions into inhibitory suggestions, and the failed-module-filter and relevant-when-filter of this link allow these 12 inhibitory suggestions to pass unhindered. When the suggestions are received by *Pickup-object*, all nine suggestions inhibiting the picking up of Moe, Larry, and Curly are ignored, because the suggestions have identical suggested-modules and suggested-bindings as suggestions already on *Pickup-object*'s suggestion-list, and these inhibitory suggestions have a hop-count one higher than the hop-count of the old, non-inhibitory suggestions. Of the remaining three inhibitory suggestions that inhibit the picking up of the pogo-stick, the first is added to *Pickup-object*'s suggestion-list. The other two are ignored, because once the first is added, the remaining two are duplicates.

8. Using the equation: base energy $= \begin{pmatrix} G \times S^H, \text{ when suggestion is supportive} \\ -G \times C \times S^H, \text{ when suggestion is inhibitory} \end{pmatrix}$

and the values $G = 22$, $C = 0.90$, and $S = 0.99$, the energies for each instantiated

behavior are calculated and are found to have the following values:

Instantiated behavior	supportive base energy	inhibitory base energy	total base energy
Eat-object **<Moe>**	**22**	**0**	**22**
Eat-object **<Larry>**	**22**	**0**	**22**
Eat-object **<Curly>**	**22**	**0**	**22**
Pickup-object **<Moe>**	**21.56**	**0**	**21.56**
Pickup-object **<Larry>**	**21.56**	**0**	**21.56**
Pickup-object **<Curly>**	**21.56**	**0**	**21.56**
Pickup-object **<pogo-stick>**	**0**	**-19.21**	**-19.21**
Walk-to-location **<[10, 10, 0]>**	**21.34**	**0**	**21.34**
Walk-to-location **<[20, 20, 0]>**	**21.34**	**0**	**21.34**
Walk-to-location **<[30, 30, 0]>**	**21.34**	**0**	**21.34**

Table 6: Example energies of instantiated behaviors

9. *Pickup-object* <pogo-stick> has a negative total base energy, so it is declared to be inhibited. The binding "pogo-stick" is added to the inhibited-blacklist for *Pickup-object*, with a penalty time of one timestep.

3.5.4 Example resolution phase

10. The only instantiated behaviors that are executable are *Walk-to-location* [10, 10, 0], *Walk-to-location* [20, 20, 0], and *Walk-to-location* [30, 30, 0]. All other instantiated behaviors are removed from consideration.
11. The nearness energy for each executable behavior is calculated and added to the base energy. In the wolf's current implementation the nearness energy for a location a given distance from the wolf is given by the following equation[15]:

$$\text{nearness-energy} = \begin{pmatrix} \text{nearness-multiplier} \times (100 - \text{distance}) / 100, & \text{distance} < 100 \\ 0, & \text{otherwise} \end{pmatrix}$$

With the wolf at [35, 35, 0], the nearness energy calculates to the following values

15. In general, the nearness equation is domain dependent. In this implementation, if a behavior represents an action that is always close to completion, such as *Drop-object*, the nearness energy is simply equal to the nearness-multiplier times 1.0.

for the three suggestions:

Instantiated behavior	total base energy	nearness energy	total energy
Walk-to-location **<[10, 10, 0]>**	**21.34**	**6.46**	**27.80**
Walk-to-location **<[20, 20, 0]>**	**21.34**	**7.87**	**29.21**
Walk-to-location **<[30, 30, 0]>**	**21.34**	**8.58**	**29.92**

Table 7: Example energies of executable instantiated behaviors

12. The total energy of all three instantiated behaviors is computed by adding each behavior's total base energy to its nearness energy. These add up to be 27.80, 29.21, and 29.92, respectively.
13. The instantiated behavior with the highest total energy is chosen as the action to execute. In this case, the effector of *Walk-to-location* is called with the argument [30, 30, 0]. This causes the wolf to take a single step towards Curly, slowly but surely sealing his doom.

3.5.5 Example special cases phase

14. Of the two executable instantiated behaviors that did not get chosen for execution, neither was passed over only because of inhibition, so no behavior is added to an inhibited-blacklist in this step.
15. If the physics layer reports an error in executing the action *Walk-to-location* [30, 30, 0], the binding [30, 30, 0] will be added to *Walk-to-location*'s failure-list. Otherwise, it won't.
16. There are no instantiated behaviors that are currently unsatisfiable, so no bindings are added to any module's giveup-blacklist this timestep.
17. An action was chosen this timestep, so no corrective measures need to be taken.

3.6 Avoiding thrashing caused by failed action

ANA will occasionally get into loops if an action fires but fails to accomplish what is expected. This problem can cause a character to thrash, blindly performing the same failed action over and over again without any benefit. For example, a character try forever to pickup an object that is too heavy to lift. Maes argues that in a rapidly changing world these action loops will soon be broken because the environment will change [Maes, 1989], but loops become more of a problem in less dynamic environments. In the PHISH-Nets architecture, the failure-blacklist in each module prevents this simple form of thrashing.

A more subtle form of thrashing occurs when an action has several preconditions, one or more of which are unsolvable. For example, take the simple Big Bad Wolf specification, but assume *Drop-object* is unavailable (either because it failed previously, is being inhibited, or simply is not implemented). If the wolf is carrying a non-food item, he will first walk over to the food (i.e., a pig). Since his hands are full the wolf cannot pick up the pig, but will still religiously stay next to the pig to satisfy the other precondition for *Pickup-object*, even if staying near the pig means not satisfying less important but satisfiable goals. This situation is an example of a "dog in the manger" problem, where a module keeps less important modules from acting even though it cannot become active itself.

The PHISH-Nets architecture detects dog-in-the-manger problems by noting instantiated behaviors that are not executable but cannot create output from any of their predecessor-links. These behaviors are added to the giveup-blacklist for its associated module, and this blacklist is used the same as the failure-blacklists to prevent thrashing. Because a module cannot accept a binding on a giveup-blacklist, it is possible to have trickle-up effects where the failure of a single instantiated-behavior causes another behavior to be added to a giveup-blacklist, which in turn causes a third behavior to be added to a giveup-

blacklist. For example, in the above case where *Drop-object* is not available the pig would be added to *Pickup-object*'s giveup-blacklist. On the following timestep, the pig would be added to *Eat-object*'s giveup-blacklist, because giving up on picking up the pig made *Eat-object*'s preconditions unsatisfiable.

There are a few degenerate cases where a behavior is impossible to execute (i.e., not all its preconditions can be satisfied), but where this condition is not detected and therefor the instantiated behavior is not put into the giveup-blacklist. For example, looping preconditions (e.g., where a character needs to fix the hole in the bucket to fetch the water needed to fix the hole in the bucket) are not detected in the above procedure. This particular form of the dog-in-the-manger problem is discussed in Section 5.5.1.

3.7 What to do when things still go wrong

Rather than trying to design the algorithm to handle all situations a character might ever encounter, the PHISH-Nets architecture is instead designed to handle as many common situations as is reasonably possible and to detect and remedy a breakdown should it occur. The simplest form of breakdown occurs when a creature has not come up with an action by the end of the action-selection cycle. One possible cause is that there is no executable action that helps a goal. In this case taking no action is appropriate. However, a second possible cause is that an impossible-to-satisfy (but not given up on) high-energy goal is inhibiting lower-energy but satisfiable goals. Yet another possibility is that a behavior could not be executed earlier and was added to a giveup-blacklist, but since that time the environment has changed such that the behavior can now become executable. If no action is taken, the following actions are taken to remedy the situation. First, the highest energy module/binding pair that is actively inhibiting another module/binding is placed in

the giveup-blacklist. If after this action the creature still can take no action, all giveup-blacklists are cleared.

The methods detailed above handle most common failure-modes, but the PHISH-Nets algorithm can still get into loops in certain circumstances. These situations, and how future extensions might handle them, are discussed in Section 5.5.6