FevahrCassie: A Description and Notes for Building FevahrCassie-Like Agents SNeRG Technical Note 35

Stuart C. Shapiro
Department of Computer Science and Engineering
and Center for Cognitive Science
University at Buffalo, The State University of New York
201 Bell Hall
Buffalo, NY 14260-2000
shapiro@cse.buffalo.edu

September 26, 2003

Contents

1 Introduction

The purpose of this document is to describe the FevahrCassie cognitive agent, and to provide instructions for building SNePS/GLAIR cognitive agents based on FevahrCassie in the ASCII world.

Throughout this document "entity" will mean a mental entity denoted by a SNePS term (node), and "object" will mean an object in the real or simulated world in which the agent operates.

2 The Fevahr Environment

The Fevahr operates in a $17' \times 17'$ room containing:

- Cassie;
- Stu, a human supervisor;
- Bill, another human;
- a green robot;
- three indistinguishable red robots.

FevahrCassie is always talking to either Stu or Bill (initially to Stu). That person addresses Cassie when he talks, and Cassie always addresses that person when she talks. Cassie can be told to talk to the other person, or to find, look at, go to, or follow any of the people or robots in the room. Initially, Cassie is looking at Stu. Cassie can also engage in conversations on a limited number of other topics in a fragment of English.

FevahrCassie grounds some of its symbols in perception by "aligning" some of its SNePS KL terms with sub-KL *descriptions.* A description is a pair, $\langle color, shape \rangle$, where each of *color* and *shape* is a symbol that can be used by the PML or lower levels to find the real or simulated objects. Table 1 shows the descriptions of Cassie, Stu, Bill, the color green, the color red, and the categories of people and robots in the ASCII simulation. Partial descriptions are unified to get full descriptions. For example, the full description of the

KL term	Entity	ASCII World Description
b1	Cassie	⟨World:cyan, World:circle⟩
b5	Stu	⟨World:yellow, World:square⟩
b6	Bill	⟨World:blue, World:square⟩
m21	green	⟨World:green, nil⟩
m25	red	⟨World:red, nil⟩
m19	people	⟨nil, World:square⟩
m22	robots	⟨nil, World:circle⟩

Table 1: Descriptions aligned with KL terms

green robot in the ASCII simulation is ⟨WORLD:GREEN, WORLD:CIRCLE⟩

The Fevahr environment also includes the directions *up, down, left,* and *right,* which are aligned with the symbols World:up, World:down, World:left, and World:right, respectively.

The Fevahr ASCII environment is easier than other simulated environments to change, since it doesn't require any graphical simulations of objects and actions. Every Fevahr action in the ASCII simulation is effected by just printing an appropriate message to the standard output stream.

3 Relevant Files

You may use these files without moving them from their current directories, but any file you need to modify, you must first copy into your own directory.

3.1 SNeRE Patches

These files must be loaded after loading SNePS 2.6, because the acting model used by Fevahr is a bit different from the one documented in the SNePS 2.6 User's Manual. (See §5.2.) It should not be necessary to modify these files.

- `/projects/robot/Fevahr/newachieve.cl`
- `/projects/robot/Fevahr/newact.cl`

3.2 Fevahr Files

These files define the Fevahr agent independent of whether the environment it acts in is real or simulated, or how the simulated environment is implemented.

`/projects/robot/Fevahr/lexicon.cl` The lexicon. New words must be added. Use existing words as a pattern.

`/projects/robot/Fevahr/grammar.cl` The parsing/generation grammar. Changing this file requires a knowledge of GATNs.

`/projects/robot/Fevahr/fevahr.sneps` SNePSUL file of background knowledge defining FevahrCassie's GLAIR Knowledge Level (KL). Must be changed to incorporate new entities and new actions. This file also loads five others. You must change the paths of any of them that you have moved (and modified).

`/projects/robot/Fevahr/pml.cl` A Lisp file defining various functions for FevahrCassie's GLAIR Perceptuo-Motor Level (PML), including one Fevahr primitive action function (`find-deicticfun`). The function `recognize` must be changed to incorporate new entities/objects and classes of entities/objects.

`/projects/robot/Fevahr/pml2.cl` A Lisp file defining FevahrCassie's primitive action functions, along with a few helper functions. For some reason, this file cannot be compiled, and that's really what distinguishes it from `pml.cl`.

3.3 World Files

These files are different for different simulations of the agent environment. The files listed here are for the Fevahr ASCII simulation.

`/projects/robot/Fevahr/Ascii/afevahr.sneps` SNePSUL file that loads two other files, and then defines the alignments. The alignments must be modified to incorporate new entities, classes, and perceptual properties. The file paths must be modified if either of these two files have been moved (and modified).

This is the file to `demo` or `intext` to start operating the FevahrCassie ASCII version.

`/projects/robot/Fevahr/Ascii/world.cl` Lisp file in the `World` package defining the objects and actions of the ASCII world. Must be modified for new objects, properties, and actions.

4 Categories, Properties, and Individuals

4.1 Introduction

In this section, I will begin to discuss the entities currently in FevahrCassie's model of her world, and how to add new entities. I will give the syntax and (intensional) semantics of each formal construct by showing:

- *A logic-like term that denotes the entity*;
- `The SNePSUL code for building a SNePS term that represents the entity`;
- an informal English gloss of the intensional semantics of the term.

First, we have lexicalized entities and strings:

"*lx*" `(build lex lx)` the entity lexicalized as *lx*.

$'w'$ `(build beg (build word w) end (build word w))` the string consisting of the word "*w*".

Since SNePS terms (nodes) represent mental entities, multiple mental entities may correspond to the same object in the world—we say that the entities are coextensional. This is formalized as

Equiv(*e1*, *e2*) `(build equiv (e1 e2))` *e*1 and *e*2 are coextensional.

4.2 Categories

4.2.1 Current Categories

The category (class) hierarchy is given using the *Subclass* relation:

Subclass(*c1*, *c2*) `(build subclass c1 superclass c2)` *c1* is a subcategory (subclass) of *c2*.

The initial category hierarchy is:

Subclass("*agent*", "*thing*")
Subclass({"*person*", "*robot*"}, "*agent*")
Subclass({"*man*", "*woman*", "*astronaut*"}, "*person*")
Subclass("*FEVAHR*", "*robot*")

Recall that *Subclass*({"*person*", "*robot*"}, "*agent*") means that both the category "*person*" and the category "*robot*" are subcategories of the category "*agent*". The perceivable categories are *"person"* and *"robot"*. See Table 1.

4.2.2 Adding Categories

To add a perceivable category of entities/objects, for example to add birds:

1. If necessary, add the word that designates the category to the file, `lexicon.cl`. For example add

   ```
   ("bird" ((ctgy . n)))
   ```

2. Add a SNePSUL form to the file `fevahr.sneps` to build a node representing the category, a SNePSUL variable whose value is that node, and at least one member of the category. For example,

   ```
   (describe (assert member #aBird class (build lex "bird") = birds))
   ```

3. Add any desired background knowledge about the new category to `fevahr.sneps`.

4. Choose a symbol to represent the *"shape"* of the new category of entities/objects, for example `avian`. Add that symbol to the list of symbols being exported from the `World` package in the file `world.cl`. For example, add `#:avian` to the list `(:export ...)`.

5. Align the node representing the category with its *shape*, by adding an appropriate dotted pair to the `*alignments*` association list in the file `afevahr.sneps`. For example, add

   ```
   (cons (sneps:choose.ns #2!(*birds))
         '(nil World:avian))
   ```

4.3 Properties

4.3.1 Current Properties

The properties known by FevahrCassie are *"locative"*, *"negative"*, *"green"*, and *"red"*, of which the last two are perceivable properties (see Table 1).

4.3.2 Adding Properties

To add a perceivable property:

1. If necessary, add the word that designates the property to the file, `lexicon.cl`. For example add

   ```
   ("black" ((ctgy . adj)))
   ```

2. Add a SNePSUL form to the file `fevahr.sneps` to build a node representing the property, a SNePSUL variable whose value is that node, and at least one entity having that property. For example,

   ```
   (describe (assert object #aCrow property (build lex "black") = black))
   ```

3. Choose a symbol to represent the perception (*"color"*) of the new property, for example `black`. Add that symbol to the list of symbols being exported from the `World` package in the file `world.cl`. For example, add `#:black` to the list `(:export ...)`.

4. Align the node representing the property with its perception, by adding an appropriate dotted pair to the `*alignments*` association list in the file `afevahr.sneps`. For example, add

   ```
   (cons (sneps:choose.ns #2!(*black))
         '(World:black nil))
   ```

4.4 Complex Categories

4.4.1 Current Complex Categories

A complex category is represented as

CompCat(m, c) `(build classmod` *m* `classhead c)` the complex category m c.

FevahrCassie's current complex categories are

CompCat("locative", "relation")
CompCat("negative", "adjective")
CompCat("green", "robot")
CompCat("red", "robot")

The latter two are perceivable complex categories.

4.4.2 Adding Complex Categories

To add a perceivable complex category of perceptually indistinguishable members:

1. If necessary, add the word that designates the main category and the word that designates the distinguishing property to the file, `lexicon.cl`. For example add

   ```
   ("black" ((ctgy . adj)))
   ("bird"  ((ctgy . n)))
   ```

2. Add a SNePSUL form to the file `fevahr.sneps` to build nodes representing the main category, distinguishing property, and subcategory, SNePSUL variables whose values are those nodes, and at least one member of the category. For example,

   ```
   (describe (assert member #aBlackBird
                     class (build classmod (build lex "black") = black
                                 classhead (build lex "bird") = birds) = blackBirds))
   ```

3. Add any desired background knowledge about the new subcategory to `fevahr.sneps`.

4. Choose a symbol to represent the *"shape"* of the new main category of entities/objects, for example `avian`, and a symbol to represent the perception (*"color"*) of the new distinguishing property, for example `black`. Add those symbols to the list of symbols being exported from the `World` package in the file `world.cl`. For example, add `#:avian` and `#:black` to the list `(:export ...)`.

5. Align the node representing the main category with its shape, and the node representing the distinguishing property with its perception, by adding appropriate dotted pairs to the `*alignments*` association list in the file `afevahr.sneps`. For example, add

```
(cons (sneps:choose.ns #2!(*black))
      '(World:black nil))
(cons (sneps:choose.ns #2!(*birds))
      '(nil World:avian))
```

6. Give the agent the ability to recognize indistinguishable members of the subcategory by adding the new subcategory to the `recognize` function in the file `pml.cl`. For example, change

```
(let ((individuals '(Bill Stu I))
      (groups (list (* 'green-robots) (* 'red-robots))))
      ...
    )
```

to

```
(let ((individuals '(Bill Stu I))
      (groups (list (* 'green-robots) (* 'red-robots) (* 'blackBirds))))
      ...
    )
```

4.5 Individual Entities

4.5.1 Current Individual Entities

Individual entities are represented with the help of the following forms:

$Isa(x, c)$ `(build member x class c)` x is a member of the category c.

$Name(e, n)$ `(build object e propername n)` n is the proper name of the entity e.

FevahrCassie knows about the following perceivable individual entities

$Isa(b1, \text{“}FEVAHR\text{”})$
$Name(b1, Cassie)$
$Isa(\{b5, b6\}, \text{“}person''\text{”})$
$Name(b5, Stu)$
$Name(b6, Bill)$
$Isa(b7, CompCat(\text{“}green\text{”}, \text{“}robot\text{”})$
$Isa(\{b8, b9, b10\}, CompCat(\text{“}red\text{”}, \text{“}robot\text{”}))$
$Isa(b11, \text{“}woman\text{”})$
$Isa(b12, \text{“}man\text{”})$

and the following nonperceivable entities

$Isa(\{\text{“}left\text{”}, \text{“}right\text{”}, \text{“}up\text{”}, \text{“}down\text{”}\}, \text{“}direction\text{”})$
$Isa(\{\text{“}dial\text{”}, \text{“}eat\text{”}, \text{“}follow\text{”}\}, transparent)$
$Isa('fake', CompCat(\text{“}negative\text{”}, \text{“}adjective\text{”}))$
$Isa('near', CompCat(\text{“}locative\text{”}, \text{“}relation\text{”}))$

4.5.2 Adding Perceivable Individual Entities

A perceivable object

- may be recognizable due to its own idiosyncratic *color* and *shape,*
- or it may inherit its *shape* from some perceivable category it is a member of and have its own idiosyncratic "*color*,"
- or it may get its *color* from one of its perceivable properties and have its own idiosyncratic *shape*,
- or it may get its *color* from one of its perceivable properties, and inherit its *shape* from some perceivable category it is a member of.
- or it may get both its *color* and its *shape* from a perceivable subcategory it is a member of.

To add a percepually recognizable entity/object:

1. If necessary, add the words (adjectives, common nouns, proper nouns) that will be used to refer to the entity to the file, `lexicon.cl`.
2. Add the perceivable (complex) categories the entity is to be a member of, and its perceivable properties, by following the instructions in the preceding subsections.
3. Add a SNePSUL form to the file `fevahr.sneps` to build a node representing the entity, a SNePSUL variable whose value is that entity, and nodes to give the entity the appropriate properties and category memberships. In the following examples, I will use `Bart` as the SNePSUL variable whose value is the node denoting the new entity, whether or not it has a proper name.
4. Add any desired background knowledge about the new entity to `fevahr.sneps`.
5. If the new entity is to have its own idiosyncratic *color* and/or *shape,* choose a symbol (symbols) and add it (them) to the exported symbols of the `World` package as in the instructions above.
6. If the new entity is to have its own idiosyncratic *color* and/or *shape,* align it with that (those) feature(s) by adding to the `*alignments*` association list in the file `afevahr.sneps`. For example, you might add

```
(cons (sneps:choose.ns #2!(*Bart))
      '(World:black World:avian))
```

   or

```
(cons (sneps:choose.ns #2!(*Bart))
      '(nil World:avian))
```

   or

```
(cons (sneps:choose.ns #2!(*Bart))
      '(World:black nil))
```

   If the new entity will get its *color* from a property and its *shape* from a category, there is no need to add it to the `*alignments*` list.

7. Add the new object to the world by adding it and its description to the value of the constant `world` in the `world.cl` file. For example, add

```
(Bart black avian)
```

8. Give the agent the ability to recognize the new entity by adding it to the `recognize` function in the file `pml.cl`. For example, change

```
(let ((individuals '(Bill Stu I))
      (groups (list (* 'green-robots) (* 'red-robots))))
      ...
    )
```

to

```
(let ((individuals '(Bill Stu I Bart))
      (groups (list (* 'green-robots) (* 'red-robots))))
      ...
    )
```

9. Include the new object in the list of possible objects to point to in the function `attendToPoint` in the file `world.cl`, for example, by changing

```
(format t "out of: World::Bill World::Stu~%~
                ~T World::Greenie ~
                   World::RedRob1 World::RedRob2 World::RedRob3:  ")
```

to

```
(format t "out of: World::Bill World::Stu~%~
                ~T World::Greenie World::Bart ~
                   World::RedRob1 World::RedRob2 World::RedRob3:  ")
```

5 Acts, Times, Events, and Event Propositions

5.1 Acts

We view an act as something that can be performed by various agents, possibly including Cassie. Most acts are represented as

action(x) `(build action (build lex` ***action***`) object` ***x***`)`
The act of performing *action* on the object x.

The acts represented in this way for FevahrCassie are *find*(x), *follow*(x), and *help*(x).

Acts involving the actions *"go"*, and *"talk"* are represented as

$a(x)$ `(build action (build lex "`***a***`") to` ***x***`)` The act of a-ing to the object x.

the act involving the action *"look"* is represented by

look(x) `(build action (build lex "look") at` ***x***`)` The act of looking at the object x.

and the act involving the action *"stop"* is represented by

stop `(build action (build lex "stop"))` The act of stopping.

5.2 The Event Model

The FevahrCassie acting model differs from the one documented in the SNePS 2.6 User's Manual, by using events instead of acts in many of the acting rules. I will use the same "logical form" for an event as I do for an act, except that the event will have an additional first argument of an agent. The syntax and semantics I will use for events is

$action(a, x)$ `(build agent a act action(x))` The event of agent a's performing the act $action(x)$.

For example, the event $go(a, x)$ represents the event of the agent a performing the act $go(x)$.

For convenience, if *act* is an act and a is an agent, I will also use the form $perform(a, act)$ to denote the event of a's performing act *act*. For example, $perform(a, go(x))$ will be equivalent to $go(a, x)$.

5.3 Time

SNePS terms (nodes) are used to represent times (time intervals). Times may be related to each other by the following forms

$Before(t1, t2)$ `(build before t1 after t2)` Time $t1$ is before time $t2$.

$Subint(t1, t2)$ `(build subint t1 supint t2)` Time $t1$ is a subinterval of time $t2$.

The virtual arc `contains` is defined by a path-based inference rule (see §9) as going from a time t_1 to a time t_2 whenever t_2 is a subinterval of (is during) t_1.

The SNePSUL variable `NOW` always has as its value the current time, so while FevahrCassie is doing something, its time `contains` the value of `NOW`. If the time associated with an event is before `NOW`, that event was, as far as FevahrCassie is concerned, in the past.

5.4 Event Propositions

The proposition that some agent did something is represented by

$Time(e, t)$ `(build event e time t)` The time that event e was/is-being performed is t.

The $Time(e, t)$ proposition is also used when e is the relation of an agent's being near an object or another agent:

$near(a, o)$ `(build rel (build lex near) arg1 a arg2 o)` Agent a is near the agent or object o.

5.5 FevahrCassie's Initial Events

When FevahrCassie is started, she believes these time and event propositions.

$SubInt(b2, \{b3, b4\})$ ($b2$ is the initial `NOW`)
$Time(talk(b1, b5), b3)$
$Time(lookAt(b1, b5), b4)$

These mean that when FevahrCassie is started, she believes that she is talking to and looking at Stu.

6 Primitive Acts

6.1 Durative and Punctual Acts

The basic design of each agent includes a set of primitive acts it can perform. Each primitive act is either durative or punctual. A durative act may be "experienced from the inside," while a punctual act cannot be. For example, for FevahrCassie, *follow* is a durative act while *find* is a punctual act. After being asked to *Follow Bill,* Cassie says *I am following Bill*, and only when she is no longer following Bill will she say

I followed Bill. However, after being asked to *Find Bill,* she will say *I found Bill,* and will never say *I am finding Bill.*

When a primitive act is performed, it may cause the termination of another primitive act, or it may allow the other primitive act, if durative, to continue. For example, talking to Bill requires stopping talking to Stu, but permits continuing looking at whatever the agent was previously looking at. The implementation of this interaction is aided by the following functions, defined in the file `pml2.cl`:

`assert-punctual-act` ***act***: Create a new time t after the current `NOW`, and a new `NOW` after t. Believe that $Time(perform(b1, act), t)$ (Cassie performs ***act*** at time t), utter that belief, and perform forward inference on it.

`assert-durative-act` ***act***: If Cassie believes she is currently performing the ***act***, just utter that belief. Otherwise: using `end-durative-acts`, terminate other acts of the same action; create a new time t after the current `NOW`, and a new `NOW` as a superinterval of t; believe that $Time(perform(b1, act), t)$, utter that belief, and perform forward inference on it.

`continue-durative-acts` ***actions***: Find the times of all acts the agent is performing using any of the ***actions*** that aren't already believed to be before the current `NOW`, believe that those times are superintervals of `NOW`, and do forward inference on those beliefs.

`end-durative-acts` ***actions***: Find the times of all acts the agent is performing using any of the ***actions***, believe that those times are before `NOW`, and do forward inference on those beliefs.

The six primitive acts of FevahrCassie, other than those predefined as part of SNeRE, are (each entry shows a logical form of the act, whether the act is durative or punctual, and a description of the act performed):

1. $find(x)$ punctual

   If the agent is currently looking at a world object that fits the PML description of x,
   If the entity it is looking at, y, is not the same as x, believe $Equiv(x, y)$.
   Say, *I am looking at y.*
   Otherwise,
   Use `end-durative-acts` on looking.
   Use `World:find` to find an object that fits the PML description of x.
   Let y be the entity the found object looks like.
   If x and y are not the same, believe $Equiv(x, y)$.
   Use `assert-punctual-act` on $find(y)$.
   Use `end-durative-acts`, on following.
   Use `assert-durative-act` on $look(y)$.
   Use `continue-durative-acts`, on talking.

2. $finddeictic(x)$ punctual
   (This act is triggered directly from the grammar, when a noun phrase is *this*, *that*, *this* ⟨*common noun*⟩, or *that* ⟨*common noun*⟩.)

   Use `World:attendToPoint` to solicit a pointing gesture from the user, and look at the object pointed to.
   Let y be the entity that object looks like.
   If the agent believes it's already looking at y,
   Say *I am looking at y.*
   Otherwise,
   Use `end-durative-acts` on following.
   Use `assert-durative-act` on $look(y)$.
   Use `continue-durative-acts` on talking.

3. *follow*(x) durative

   Use `World:followFocussedObject` to follow whatever object it's looking at.
   Use `assert-durative-act` on *follow*(x).
   Use `continue-durative-acts` on looking and talking.

4. *go*(x) punctual

   If x is a direction,
   Use `end-durative-acts` on following.
   Use `World:goDir` to move in the x direction.
   Use `assert-punctual-act` on *go*(`x`).
   Use `continue-durative-acts` on talking and looking.
   Otherwise,
   If currently *near*($b1, x$), say so.
   Otherwise
   Use `end-durative-acts` on following.
   Use `World:goToFocussedObject` to go to whatever object it's looking at.
   Use `assert-punctual-act` on *go*(x).
   Believe *near*($b1, x$), say so, and do forward inference on the belief.
   Use `continue-durative-acts` on talking and looking.

5. *stop* punctual

   Use `World:stop` to stop looking at anything.
   Use `end-durative-acts` on following and looking.
   Use `assert-punctual-act` on stopping.
   Use `continue-durative-acts` on talking.

6. *talk*(x) durative

   Use `World:talkto` to start talking to x.
   Use `assert-durative-act` on *talk*(x).
   Use `continue-durative-acts` on following and looking.

6.2 ASCII World Primitive Acts

The primitive acts described above each bottom out by calling a function in the `World` package, defined in the file `world.cl`. These functions are:

`(find` ***description*** `)`
Finds an object that satisfies the ***description***, which may be incomplete. If sucessful, the global variable `World:FocussedObject` is set to the object.

`(attendToPoint` ***description*** `)`
Prompts the user to enter the object being pointed to. The global variable `World:FocussedObject` is set to the object.

`(followFocussedObject)`
Prints a message that Cassie is starting to follow the object that is the value of `World:FocussedObject`.

`(goDir` ***d*** `)`
Prints a message that Cassie is moving in direction ***d***.

`(goToFocussedObject)`
Prints a message that Cassie is moving next to the object that is the value of `World:FocussedObject`.

`(stop)`
Sets `World:FocussedObject` to `nil`, and prints a message that Cassie has stopped.

`(talkto` *name*`)`
Prints a message that Cassie is starting to talk to *name*.

If you are building a graphically simulated environment, these functions must be rewritten to make appropriate modifications to the environment. If using an actual robot, these functions must be modified to send appropriate commands to the robot.

6.3 Acting Rules for Primitive Acts

The acting rules will be given using the following "logical forms" (used in SNePSLOG's Mode 3).

ActPlan(a, p) The way to perform a is by doing p.

GoalPlan(s, a) The way to achieve the goal s is by doing a.

PreCond(s, a) The state s is a precondition for doing a.

snsequence(a_1, a_2) First (begin to) do a_1, then (begin to) do a_2.

do-all(a_1, a_2) (Begin to) do a_1 and a_2 in a non-deterministic order.

1. $\forall(a, x)(Agent(a) \wedge Thing(x) \Rightarrow PreCond(near(a, x), follow(a, x)))$
   A precondition for a's following x is that a be near x.
2. $\forall(a, x)(Agent(a) \wedge Thing(x) \Rightarrow GoalPlan(near(a, x), go(a, x)))$
   A way to achieve the goal of a's being near x is for a to go to x.
3. $\forall(a, x)(Agent(a) \wedge Thing(x) \Rightarrow PreCond(look(a, x), go(a, x)))$
   A precondition for a's going to x is that a be looking at x.
4. $\forall(a, x)(Agent(a) \wedge Thing(x) \Rightarrow GoalPlan(look(a, x), find(a, x)))$
   A way to achieve the goal of a's looking at x is for a to find x.

6.4 Adding Primitive Acts

To add a new primitive act:

1. If necessary, add the words that will be used to tell Cassie to perform the act to the lexicon file, `lexicon.cl`.
2. Add a Lisp function to perform the act to `world.cl`. For the ASCII Fevahr, this should involve printing some message, and possibly setting or changing some global variables in the `World` package.
3. Using `define-primaction`, add Lisp code to perform the act to `pml2.cl`. This should, at least, call the `World` function you defined, and assert the belief that Cassie is performing (or has performed) the act, using `assert-durative-act` or `assert-punctual-act`. Deal with durative acts that should be ended or continued.
4. Attach the primitive action function defined in `pml2.cl` to the node representing the action by adding a form to the call to `attach-primaction` in the file `fevahr.sneps`.
5. Add any necessary acting rules concerning the new act to `fevahr.sneps`.

7 Non-Primitive Acts

7.1 Current Non-Primitive Acts

Non-primitive acts are specified using the SNeRE acting constructs. FevahrCassie only has two non-primitive acts:

1. $\forall(x)(\mathit{Thing}(x) \Rightarrow \mathit{ActPlan}(\mathit{look}(x), \mathit{find}(x)))$
   To look at something, find it.

2. $\forall(a)(\mathit{Agent}(a) \Rightarrow \mathit{ActPlan}(\mathit{help}(a), \mathit{do\text{-}all}(\mathit{talk}(a), \mathit{follow}(a))))$
   To help an agent, talk to it and follow it.

7.2 Adding Non-Primitive Acts

To add a new non-primitive act,

1. If necessary, add the words that will be used to tell Cassie to perform the act to the lexicon file, `lexicon.cl`.

2. Add the specification of the new non-primitive act to `fevahr.sneps` using the SNeRE acting constructs.

8 Additional Initial Rules

For completeness, the rest of FevahrCassie's initial rules are listed here, using the forms defined above and the additional form:

$\mathit{Has}(a, c, x)$ `(build object x rel c possessor a)` x is the c of a.

Each rule is given formally followed by its informal English gloss.

$\forall(a, c, x)[\mathit{Has}(a, c, x) \Rightarrow \mathit{Isa}(x, c)]$
If x is a's c, then x is a c.

$\forall(a, n)[\mathit{Name}(a, n) \Leftrightarrow \mathit{Has}(a, \text{“}\mathit{name}\text{”}, n)]$
The two ways of representing the proposition that n is a's name are equivalent.

$$\forall \mathit{action}[\mathit{Isa}(\mathit{action}, \mathit{transparent}) \Rightarrow [\forall(a, x1, x2, t)[\mathit{Time}(\mathit{action}(a, x1), t) \wedge \mathit{Equiv}(x1, x2) \Rightarrow \mathit{Time}(\mathit{action}(a, x2), t) \wedge \mathit{Equiv}(\mathit{Time}(\mathit{action}(a, x1), t), \mathit{Time}(\mathit{action}(a, x2), t))]]]$$
If an action is transparent, agent a perfoms it on $x1$ at time t, and $x1$ is coextensional with $x2$, then a also performs the action on $x2$ at time t, and the two propositions, that a performed the action on $x1$ at t and that a performed the action on $x2$ at t, are, themselves, coextensional.

$$\forall(a, x1, x2, t)[\mathit{Time}(\mathit{look}(a, x1), t) \wedge \mathit{Equiv}(x1, x2) \Rightarrow \mathit{Time}(\mathit{look}(a, x2), t) \wedge \mathit{Equiv}(\mathit{Time}(\mathit{look}(a, x1), t), \mathit{Time}(\mathit{look}(a, x2), t))]$$
If agent a looks at $x1$ at time t, and $x1$ is coextensional with $x2$, then a also looks at $x2$ at time t, and the two propositions, that a looked at $x1$ at t and that a looked at $x2$ at t, are, themselves, coextensional.

$$\forall(a, x1, x2, t)[\mathit{Time}(\mathit{go}(a, x1), t) \wedge \mathit{Equiv}(x1, x2) \Rightarrow \mathit{Time}(\mathit{go}(a, x2), t) \wedge \mathit{Equiv}(\mathit{Time}(\mathit{go}(a, x1), t), \mathit{Time}(\mathit{go}(a, x2), t))]$$
If agent a goes to $x1$ at time t, and $x1$ is coextensional with $x2$, then a also goes to $x2$ at time t, and the two propositions, that a went to $x1$ at t and that a went to $x2$ at t, are, themselves, coextensional.

$\forall(a1, a2, x, t)[Time(go(a1, x), t) \wedge Equiv(a1, a2)$
$\Rightarrow Time(go(a2, x), t) \wedge Equiv(Time(go(a1, x), t), Time(go(a2, x), t))]$
If agent $a1$ goes to x at time t, and $a1$ is coextensional with $a2$, then $a2$ also goes to x at time t, and the two propositions, that $a1$ went to x at t and that $a2$ went to x at t, are, themselves, coextensional.

$\forall(a1, a2, x1, x2, t)[Time(go(a1, x1), t) \wedge Equiv(a1, a2) \wedge Equiv(x1, x2)$
$\Rightarrow Time(go(a2, x2), t) \wedge Equiv(Time(go(a1, x1), t), Time(go(a2, x2), t))]$
If agent $a1$ goes to $x1$ at time t, and $a1$ is coextensional with $a2$, and $x1$ is coextensional with $x2$, then $a2$ also goes to $x2$ at time t, and the two propositions, that $a1$ went to $x1$ at t and that $a2$ went to $x2$ at t, are, themselves, coextensional.

$\forall(x, m, c)[Isa(x, CompCat(m, c))$
$\Rightarrow Subclass(CompCat(m, c), c)$
$\wedge \forall c2[Subclass(c, c2) \Rightarrow Subclass(CompCat(m, c), CompCat(m, c2))]]$
If x is an mc then mc is a subcategory of c, and, furthermore, if c is a subcategory of $c2$, then mc is a subcategory of $mc2$.

$\forall s[Isa(s, CompCat(\text{“}negative\text{”}, \text{“}adjective\text{”})) \Rightarrow \forall(x, c)[Isa(x, CompCat(s, c)) \Rightarrow \neg Isa(x, c)]]$
If “s” is a negative adjective, then if x is an sc, then x is not a c.
(This rule has temporarily been commented out.)

9 Path-Based Inference Rules

In this section are the path-based inference rules used by FevahrCassie.

```
(define-path class
    ;; If x is not a member of class c,
    ;;    then it is not a member of any subclass of c.
    ;; If x is a member of class c,
    ;;    then it is a member of any superclass of c.
    ;; However, this inheritance of superclasses is blocked
    ;;    by properties that are negative adjectives.
    ;; I.e., if you say
    ;;                  'former' is a negative adjective.
    ;; then a former teacher will not be considered to be a teacher.
    ;; However, it needs some negative adjective to be in the network.
    (or class
        (domain-restrict
         ((compose arg- ! max) 0)
         (compose class
                  (kstar (or classhead-
                             (compose superclass- ! subclass)))))
        (compose ! class
                 (kstar (or (domain-restrict
                             ((not (compose classmod lex word-
                                            (and beg- end-)
                                            member- ! class
                                            (domain-restrict
                                             ((compose classmod lex)
                                              "negative")
                                             classhead)
                                            lex)) "adjective" ) classhead)
                            (compose subclass- ! superclass))))))
```

```
(define-path member
    ;; The member relation is transparent across coextensional entities.
    (compose member (kstar (compose equiv- ! equiv))))

(define-path superclass
    ;; The superclass relation goes up the class hierarchy,
    ;; and also from a complex category to the main category.
    (compose superclass
             (kstar (or classhead (compose subclass- !
                                           superclass)))))

(define-path subclass
    ;; The subclass relation goes down the class hierarchy,
    ;; and also from a main category to a complex subcategory.
    (compose subclass
             (kstar (or classhead- (compose superclass- !
                                            subclass)))))

(define-path equiv
    ;; equiv is transitive.
    (compose equiv (kstar (compose equiv- ! equiv))))

(define-path before
    ;; before is transitive
    ;; and if t1 is before t2,
    ;;   then it's before any subinterval of t2.
    (compose before (kstar (or (compose after- ! before)
                               (compose supint- ! subint)))))

(define-path after
    ;; after is transitive
    ;; and if t1 is after t2,
    ;;   then it's after any subinterval of t2.
    (compose after (kstar (or (compose before- ! after)
                              (compose supint- ! subint)))))
(define-path supint
    ;; supint is reflexive and transitive
    (or time ; for reflexivity of a time that's a time of some event.
        ;; for transitivity
        (compose supint (kstar (compose subint- ! supint)))))

(define-path subint
    ;; subint is reflexive and transitive
    (or time ; for reflexivity of a time that's a time of some event.
        ;; for transitivity
        (compose subint (kstar (compose supint- ! subint)))))

(define-path contains
    ;; A virtual arc from one time to another
    ;; when the second is during the first.
    ;; For use by find and findassert
    (kstar (compose supint- ! subint)))
```

References

[1] Deepak Kumar. An integrated model of acting and inference. In Deepak Kumar, editor, *Current Trends in SNePS*, pages 55–65. Springer-Verlag Lecture Notes in Artificial Intelligence, No. 437, Berlin, 1990.

[2] Deepak Kumar. The SNePS BDI architecture. *Decision Support Systems*, 16(1):3–19, January 1996.

[3] Deepak Kumar and Stuart C. Shapiro. The OK BDI architecture. *International Journal on Artificial Intelligence Tools*, 3(3):349–366, March 1994.

[4] John F. Santore and Stuart C. Shapiro. Crystal cassie: Use of a 3-d gaming environment for a cognitive agent. In *Papers of the IJCAI 2003 Workshop on Cognitive Modeling of Agents and Multi-Agent Interactions*. IJCAII, 2003. in press.

[5] Stuart C. Shapiro. Embodied Cassie. In *Cognitive Robotics: Papers from the 1998 AAAI Fall Symposium, Technical Report FS-98-02*, pages 136–143. AAAI Press, Menlo Park, California, October 1998.

[6] Stuart C. Shapiro and Haythem O. Ismail. Anchoring in a grounded layered architecture with integrated reasoning. *Robotics and Autonomous Systems*, 43(2–3):97–108, May 2003.

[7] Stuart C. Shapiro, Haythem O. Ismail, and John F. Santore. Our dinner with Cassie. In *Working Notes for the AAAI 2000 Spring Symposium on Natural Dialogues with Practical Robotic Devices*, pages 57–61, Menlo Park, CA, 2000. AAAI.

[8] Stuart C. Shapiro and The SNePS Implementation Group. *SNePS 2.6 User's Manual*. Department of Computer Science and Engineering, University at Buffalo, The State University of New York, Buffalo, NY, 2002.