TSC QP Thoughts

latest: 20050723

Thinking about thenCreate & ifNotActors

```
c:      C.T-CELL.ACTIVATION.1
        level           basic
        subOf           phys.process
        instanceOf      process.rule
        myCreator       rbt
        contexthuman.immunology
        ifActors        ( ( v.antigen ( *v.antigen ) true )
                          ( c.T-cell ( *c.T-cell ) true )
                          ( virus ( *virus ) true )
                          ( i.macrophage ( *i.macrophage ) true )
                          ( mMHC1 ( *MHC1 ) true ) )
        ifNotActors     ( ( act.c.t-cell ( *act.c.t-cell ) true ) )
        ifRelates       ( ( presents ( *i.macrophage *v.antigen ) true ) )
        thenCreate      ( *act.c.t-cell )
        thenActors      ( ( act.c.t-cell ( *act.c.t-cell ) true ) )
        thenSay                 " *Cytotoxic T-cell activated. "
```

Here, we tested for an `episode` that doesn't have a particular actor. This rule is restricted to the ***construction of a new actor***. That's the only condition under which it fires. It makes no other relations or states. Thus, if the actor already exists, this rule won't fire.

As I recall, `thenCreate` creates an *existential* actor, one with its own identity. That identity then propagates throughout the envisionment. See **False Actors**, below.

Thus, `thenCreate` has the job of making a named binding which is then used in the following `thenActors` clause. That code has been added to `BindingEngine`, `Environment`, and `FillinNextEpisodeAgent`. What is missing here is to create an existence in the KB itself to reflect the new actor, placing it as an `instanceOf` the class that is instantiated in the `Model` (envisionment).

But...

What if the actor does exist, and we want to add another of the same class?

That remains an open question.

For now, this pair:

```
thenCreate    ( *act.c.t-cell )
thenActors    ( ( act.c.t-cell ( *act.c.t-cell ) true ) )
```

is wired up as allowing to create one or more existential actors. The parens in thenCreate are ignored: as many new actor bindings can be created in the usual space-delimited fashion, for instance:

```
thenCreate    ( *act.c.t-cell  *act.foo )
thenActors    ( ( act.c.t-cell ( *act.c.t-cell ) true )
                ( act.c.t-cell ( *act.foo ) true ) )
```

Creating an actor as an instance of a particular class, consider:

```
thenCreate    (<class> *var)
e.g.
thenCreate    (Hunan *foo)
```

Semantics of ifNotRelates & ifNotStates

Initially, `ifNotRelates` and `ifNotStates` might have been created to prevent rules from firing twice on the same episode. Another way to prevent that could be an internal check to ensure that a rule didn't fire before (by checking `myMechanism`). They are used, e.g.:

```
ifNotRelates  ( ( binds ( *helper.T-cell *antigen ) true )
                ( binds ( *b-cell *antigen ) true )
                ( binds ( *macrophage *antigen ) true ) )
thenRelates   ( ( binds ( *macrophage *antigen ) true )
                ( binds ( *helper.T-cell *antigen ) true )
                ( binds ( *b-cell *antigen ) true ) )
```

Basically, that looks like a simple test to see if those relations don't exist and to fill them in if they don't.

But, what happens if one or more of them, but not all of them, already exists? Great question. Clearly, the semantics of the rule as written suggest that the rule will not fire. General `FillinNextEpisode` semantics call for an implied `AND` between slot values. Change that to an `OR` and this rule could fire and the behaviors could be tuned to behave as an `OR` condition warrants: fillin any which does not yet exist.

Loops, Cycles, & Spirals

Many moons back, we set out to model the citric acid cycle with TSC. There was a discovery made while building an envisionment. The pictorial diagram, and the name of the process clearly indicate a cycle is at work. The discovery was that, when the envisionment returned to its initial starting point, additional actors existed. We labeled that a spiral, rather than a pure cycle.

A definition for a cycle, a *loop* would be that the new episode built during `FillinNextEpisode` would be a precise equivalent to a previous episode. Otherwise, something else has been detected. That is, equivalence in a loop requires the same actors in place, as well as the same relations and states.

If a loop is not detected, but if the new episode is a superset (has more actors) of a previous episode, then a spiral of the outward type has been detected.

If a loop is not detected, but if the new episode is a superset (has less actors) of a previous episode, then a spiral of the inward type has been detected.

For now, loop detection in TinyTSC is being restricted to precise equivalence. This notion will need revisiting.

What does this mean? In the citric acid cycle, the cyclic nature will not show up until the second pass through, which may actually then return to the beginning of the second cycle.

A reason for requiring strict equivalence is this. If there are more actors involved, then, possibly, different rules can fire, changing the nature of the envisionment. If there are less actors involved, same story. If you loop a super or subset back to some previous episode, you mask the possibility of different rules firing.

False Actors

As in: `thenActors (foo (*bar) false)`

Comment found in Immune.KB:

> *all rules below have been setup to work with "no false actors". This routine in QPT will take any actor which is "false" and delete it from the episode, then it goes and finds any relation or state referencing that actor, and deletes that too. Thus, we spend more time using then.create to bind up "new" variables so we can introduce new actors and their relations/states at runtime, plus we make heavy use of the "if.not..." constructs to keep rules from multiple firings.*

*See **Thinking about thenCreate** above.*

Algorithms

Consider this example:

```
c:      ELECTROPOS?
        instance.of     flow.pred
        i.take          sx
        i.give          flag
        arguments       *elem
        my.vars *truth *x
        algorithm       ( do  ( bindq *truth F )
                              ( if.true ( isa?  *elem 'atom )
                              \ we deduce it is electropos because it does
                              \ not have an electronegativity value
                            ( cond      ( ( notnull? ( value of *elem 'electronegativity ) )
                                                  ( bindq *truth F ) )
                                        ( ( notnull? ( value of ( value.of *elem 'synonym )
                                                                  'electronegativity ) )
                                        ( bindq *truth F ) )
                                        ( T ( bindq *truth T ) ) ) )
                                             ( return *truth ) )
```

It's really hard to recall just what we were doing here, but it looks like some simple Scheme code was roaming about in the KB doing stuff. I don't think it should be that hard to gen up a simple Scheme-like interpreter for TinyTSC. In fact, I would hope that TinyTSC will serve as a platform for all sorts of extensions, like the pseudo-natural language parser.

Pseudo-NL Parsing

I run into some not presently parsable *code, for instance:*

```
 c:  IS.REL.TO
        instance.of        flow.func
        i.take             symbol symbol  sx        \ symbol symbol
        i.give             none
        arguments             *art *noun  *memb           \ *cop *reln
        algorithm   ( do   ( display> "is rel to" debug )
                            ( cond   ( ( notnull? *art )
                                          ( do  ( edisplay *art )
                                                      ( edisplay *noun ) ) )
                                        ( T ( edisplay *noun ) ) )
                             ( display " is related to " )
                             ( loop.until    ( null? *memb )
                                             ( do       ( edisplay *memb )
                                                        ( bindq *memb ( rest *memb ) ) ) ) ) )
```

It's unclear what this is doing, except to guess that it is binding some variables that answer the question what something "is related to". I no longer have access to the original TSC forth source code – damned if I can figure out where I archived it except that, on the CD where it exists, the file is completely unreadable. So, reinvention of TinyTSC is just that: re-invention.

Legacy Stuff

Original TSC had some frames related to exporting to popsim*, a cellular automata engine. Some* episodes *were labeled* `instanceOf experiment`*. I am killing any use of the word* `experiment` *including any frame that was just an* `experiment`*. If a frame was also an* `episode`*, then I leave* `episode` *in and kill whatever is related to* `experiment`.

There was a slot called `spec.of` *in* `QPDEFS3.T`

```
c:      EXOTHERM.reaction.2
        subOf                     phys.process chem.process
        instanceOf                process.rule
        spec.of                   exotherm.reaction
```

I interpret that to be specialization of *which means the same as* subOf, *so the frame is now*

```
c:      EXOTHERM.reaction.2
        subOf                   phys.process chem.process exotherm.reaction
        instanceOf              process.rule
```

Found an interesting slot in QPDEFS3.T:

```
        ifToDo
```

```
c:      EXOTHERM.reaction.1
        subOf                   phys.process chem.process
        instanceOf              process.rule
        spec.of                 exotherm.reaction
        myCreator               jp2
        ifActors                ( ( heat.source ( *heat.source ) true )
                                          ( heat.sink ( *heat.sink ) true )
                                          ( gas ( *convective.medium ) true )
                                          ( heat.sink.sensor ( *t2 ) true )
                                          ( heat.source.sensor ( *t1 ) true ) )
        ifRelates               ( ( abuts ( *heat.sink *convective.medium ) true )
                                ( abuts ( *heat.source *convective.medium ) true ) )
        ifStates            ( ( reactive ( *heat.sink ) true )
                                          ( polymer ( *heat.sink ) true )
                                          ( increasing ( *t2 ) true ) )
        ifNotStates     ( ( ACCELERATING ( *T2 ) TRUE ) )              \ DJW
ADDED
        ifToDo                  ( ( increase ( *t1 ) true ) )
        thenStates              ( ( accelerating ( *t2 ) true ) )
```

Seems there ought to be a slot

```
        thenToDo
```

Didn't find one. Don't recall what that's all about.