IKRIS Scenarios Inter-Theory (ISIT)

Jerry Hobbs

with the IKRIS Scenarios Working Group

Contributions from Danny Bobrow, Chris Deaton, Mike Gruninger, Pat Hayes, Arun Majumdar, David Martin, Drew McDermott, Sheila McIlraith, Karen Myers, David Morley, John Sowa, Marco Valtorta, and Chris Welty.

IKRIS Scenarios Fundamentals

Events

Most ontologies dealing with processes or scenarios will have something corresponding to events. PSL uses the term "activity_occurrence" for events. Cyc uses the term "Event". Other possible terms have drawbacks. "Action" connotes an event that has an agent, so it is only a subclass of events. "Process" and "Procedure" connote complex events that have subevents, and we need a term that will cover primitive events as well. One can argue that "activity" and "activity occurrence" have the same problem. For these reasons, we have chosen the term "Events".

Types and Tokens

We can refer to both event types and event tokens. PSL uses the terms "activity" and "activity_occurrence" for these two. Cyc uses the terms "EventType" and "Event". Another possible pair of term is "EventClass" and "EventInstance", but this biases one toward interpretation of types as classes, which may not be desirable (see below).

It may seem desirable to make the type-token distinction explicit in the terms, and thus use "EventType" and "EventToken". But as Pat Hayes has pointed out, an event token may be a token of many types of events. A specific event may be a token of a running event type, an exercising event type, a winning event type, and numerous other types. Essentially, "token" is a relation between specific events and event types. An event token is simply an event. Thus, we will use the Cyc terms "EventType" and "Event".

Events do not necessarily have to occur in the real world. They can occur in possible worlds as well. So the type-token distinction crosscuts the real-hypothetical distinction. If an Event occurs in the real world, this is something that can be explicitly stated, with a predicate like "Rexists" (a la Hobbs, 1985), "Real", or "Occurs".

One could also have a "occursIn" predicate that takes a possible world or some other notion of context as an argument. We can revisit this issue later.

The principal relation between EventTypes and Events is that one of the latter can be an instance of one of the former. The terminology for this in PSL is "occurrence_of"; in Cyc, "isa". Another possibility is "instanceOf". The problem with "isa" is that its use has a long history of confusion between implication/subset ("elephant isa mammal") and predication/membership ("Clyde isa elephant"). So we will use "instanceOf".

The articulation or bridging axioms for PSL, Cyc, and the inter-theory are as follows. (Coloned predicates are in the theory indicated by the label. Uncoloned predicates are in the inter-theory.)

```
    (forall (x)
            (if (EventType x)
                (exists (y)
                        (and (psl:activity y)
                             (forall (e)


                                     (iff (psl:occurrence_of e y)
                                          (instanceOf e x)))))))

    (forall (y)
            (if (psl:activity y)
                (exists (x)
                        (and (EventType x)
                             (forall (e)
                                     (iff (psl:occurrence_of e y)
                                          (instanceOf e x)))))))

    (forall (x)
            (if (EventType x)
                (exists (y)
                        (and (cyc:genls y cyc:Event)
                             (forall (e)
                                     (iff (cyc:isa e y)
                                          (instanceOf e x)))))))

    (forall (y)
            (if (cyc:genls y cyc:Event)
                (exists (x)
                        (and (EventType x)
                             (forall (e)
                                     (iff (cyc:isa e y)
                                          (instanceOf e x)))))))
```

These axioms are written in terms of the inter-theory's "x" and the resource theory's "y" to protect each resource theory from problematic properties it might inherit from another resource theory. For example, in Cyc EventTypes are classes; classes/sets are full-fledged individuals in Cyc. In PSL activities are full-fledged individuals, but they are not classes. The inter-theory is neutral on the issue. Stronger commitments can be triggered, as follows.

```
    (forall (x)
            (if (TypesAreClasses)
                (iff (cyc:genls x cyc:Event)
                     (EventType x))))

    (forall (x)
            (if (not (TypesAreClasses))
                (iff (psl:activity x)
                     (EventType x))))
```

The bridging axioms for event tokens are as follows:

```
    (forall (x)
            (iff (cyc:isa x cyc:Event)
                 (Event x)))

    (forall (x)
            (iff (psl:activity_occurrence x)
                 (Event x)))
```

States and Events

An event is normally something that involves a change of state, although one might want to argue that waiting is an event and does not involve a change of state (except on the clock). By contrast, a state is some property or relation or collection of properties and relations that hold generally over an interval of time; that is, the relevant properties and relations do not change during the course of a state's holding. A state is homogeneous, in the sense that if a state holds over an interval of time, it holds (or a state of the same type holds) over any subinterval. To describe a state is to provide a partial description of the world over a period of time.

There is another notion of "state" often used in computer science, e.g., in denotational semantics. It refers to the total state of the relevant world at a given instant; we can call it a "w-state". At the next instant, we are in a different w-state. This kind of state does not persist over time. Neither PSL nor Cyc has a notion of w-state, and we will not need it here. This notion of state will consequently be ignored in the IKRIS Scenarios effort.

The other notion of state is the one in ordinary language. It is a property or relation holding, perhaps for some interval of time, or perhaps only for an instant. It can persist over time and is only one aspect of the total world state at any given moment. An example would be the road being slippery. It is a reification of a proposition being true for a while. Other examples are "George Bush's being President", "Jerry's believing that Bush is a bad president", "Block A's being on top of Block B", and "John's being retired". It is generally a property of an entity or a relation between entities or a combination of such properties and relations of entities. We could thus call it an "e-state", in contrast to "w-state". But since we will not need the notion of "w-state", we can just dispense with the "e-" and call it a "State". Corresponding to "EventType" and "Event", we will have the terms "StateType" and "State". As with all polysemous words, it is important to keep in mind the sense in which the term "State" is used. It does not correspond to the use of the word in computer science (or in political geography).

The reason one would like to reify states is that they can themselves have various properties and relations, and treating them as individuals in the domain of discourse leads to a simple representation for this. For example, states can be causes and effects:

    The road's being slippery caused the accident.
    The freezing rain caused the road to be slippery.

They can be located in time and space and can be qualified in various ways:

    The road was extremely slippery last night between New Haven and
        Hartford.

They can be the objects of perception and cognition:

    John saw that the road was slippery.
    John sensed the road's slipperiness immediately.

In language, they can be nominalized and referred to pronominally:

    The road's slipperiness worried John.
    The road was slippery, and John knew it.

Representating these sentences is more straightforward in a notation that reifies states.

There have been prominent approaches in AI that have reified events but have not reified states. These approaches have generally developed perfectly adequate other means for doing the work done by reified states. In this effort, we do not intend to try to convince anyone to adopt one approach or the other. Rather, our goal is to bridge between the two classes of approaches.

A State is the same as Cyc's StaticSituation. So the bridging axioms are as follows:

```
    (forall (x)
            (iff (cyc:isa x cyc:StaticSituation)
                 (State x)))

    (forall (x)
            (iff (cyc:genls x cyc:StaticSituation)
                 (StateType x)))
```

There is no entity in PSL that corresponds precisely to States. Suppose we perform a sequence of lifting a block and then lowering it. In between those two activities or events, the block is up. There is no individual in the PSL ontology that corresponds to that condition holding at that moment. However, the work to be done by States in the inter-theory and by StaticSituations in Cyc can be done in a somewhat different way by fluents in PSL. A fluent is a reified eternal propostion that can be the first argument of the predicates "Holds(f,e)" and "Prior(f,e)". "Prior(f,e)" means that the fluent f is true before the beginning of the activity occurrence e. "Holds(f,e)" means that the fluent f is true after the end of the activity occurrence e. In the above situation, the fluent "(up BlockA)" is true after the lifting and not after the lowering.

A State and a fluent are different in that a State is generally temporally bounded, whereas a fluent is eternal. Thus, there was a state of George Washington's being alive that occurred during the interval from 1732 to 1799. That State does not exist today, in the same way that the Event of George Washington's crossing the Delaware does not exist today; it existed in December 1776. By contrast, the fluent "(alive GW)" does exist today; it is eternal; it just happens not to hold today.

There can be many States corresponding to a single fluent. The fluent "(SunShiningOn LA)" holds on many days and never holds at night (in LA). Each of these holdings can be viewed as a separate State, e.g., the Sun's shining on LA on September 7, 2005, is a State, and the Sun's shining on LA on September 8, 2005, is a different State. (We can also have the State that is the aggregate of the two.)

But the notions of "State" and "fluent" are clearly closely related.

To bridge between these two perspectives, we will say that every State has a corresponding fluent, that fluent that describes the State, or holds in circumstances when the State exists. So the fluent we might represent as "full(GLASS1)" is the fluent corresponding to the State of GLASS1 being full between 10 and 11 am today. (Hobbs (1985) uses the notation, "full'(e,GLASS1,T)" to mean that e is the state of GLASS1 being full during time interval T.) There is one fluent for many States. One can think of the State as having a time argument and the fluent as representing the same proposition without the time argument, although this is not strictly necessarily true.

We will use the inter-theory predicate "fluentFor" to express the relation between an State and its corresponding fluent. The meaning of "fluentFor" is constrained by at least the following axioms:

```
    (forall (e f)
            (if (fluentFor f e)
                (and (psl:fluent f) (State e))))

    (forall (e)
            (if (State e)
                (exists (f)(fluentFor f e))))
```

That is, "fluentFor" is a relation between a PSL fluent and an inter-theory State, and there is a fluent corresponding to every State.

Before we further constrain "fluentFor", it will be useful to have a way of saying that a fluent holds at or for a particular time instant or interval. In PSL, "holds" is a relation between a fluent and an activity occurrence. We will assume, quite reasonably, that fluents only change as the result of activity occurrences, or Events. Thus, for a fluent to hold for a time is for some Event to make the fluent hold before that time where no other Event between the first Event and the time makes the fluent not hold.

First, it will be convenient to augment the time ontology with one predicate. A prefix of "t:" before a predicate means that it comes from the OWL-Time ontology.

```
    (forall (t1 t2)
            (iff (t:before/= t1 t2)
                 (or (t:before t1 t2) (= t1 t2))))
```

The predication "holdsFor(f,t)" says that the fluent holds for the instant or interval t. t may or may not be the entire time during which it holds. The definition of "holdsFor" is complicated somewhat by the necessity of taking infinite intervals into account, where infinite intervals are taken to have no beginning and/or no end.

```
    (forall (f t t1)
       (if (t:begins t1 t)
          (iff (holdsFor f t)
             (exists (o1)
                (and (psl:activity_occurrence o1)
                   (psl:holds f o1)
                   (t:before/= (psl:end_of o1) t1)
                   (not (exists (o2)
                           (and (psl:activity_occurrence o2)
                                (psl:falsifies o2 f)
                                (t:before/= (psl:end_of o1)
                                            (psl:end_of o2))
                                (forall (t2)
                                   (if (t:ends t2 t)
                                      (t:before (psl:end_of o2) t2)
         ))))))))))
```

This says that if t has a beginning t1, then f holds for t if and only if there is an Event or activity occurrence o1 beginning before or at the same time as t which makes the fluent f true, and there is no event or activity occurrence o2 after o1 and before the end of t that makes f false. The latter condition is embedded within an implication to accommodate positively infinite intervals.

The following axiom handles the case of negatively infinite intervals.

```
    (forall (f t)
       (if (not (exists (t1) (t:begins t1 t)))
          (iff (holdsFor f t)
               (forall (o)
                  (if (and (psl:activity_occurrence o)
                           (forall (t2)
                                   (if (t:ends t2 t)
                                      (t:before/= (psl:end_of o) t2))))
                           (psl:holds f o))))))
```

That is, if t has no beginning, then f holds for t if and only if f holds after any event that ends before the end of t, if t has an end.

Now we can state two more axioms constraining the interpretation of the predicate "fluentFor". If a State e obtains during a time t, then the fluent f for e holds for t.

```
    (forall (e f t)
       (if (and (State e) (t:during t e) (fluentFor f e))
           (holdsFor f t)))
```

If a fluent f holds for a time t, then there is a state e whose fluent is f and which holds during time t.

```
    (forall (f t)
       (if (and (psl:fluent f) (holdsFor f t))
           (exists (e)
                   (and (fluentFor f e) (t:during e t)))))
```

We should decide how states are to be individuated. For example, if there is a state of John's sleeping from 8pm to midnight, is there another state of John's sleeping from 8pm to 9pm? If states are interpreted as Pat Hayes's histories, that is as a chunk of space-time, then the answer is yes. In that case, we can strengthen the above axiom by saying that there is a state e whose fluent is f and whose time span is t.

```
    (forall (f t)
       (if (and (psl:fluent f) (holdsFor f t))
           (exists (e)
                   (and (fluentFor f e) (t:timeSpanFor t e)))))
```

(Recall that "(t:during e t)" only says that the state e obtains all through time t, whereas "(t:timeSpanFor t e)" says that t is the entire time during which e obtains.)

We can view a fluent as a StateType:

```
    (forall (f) (if (psl:fluent f) (StateType f)))
```

But the converse does not hold. If time is continuous and thus cannot be modelled as a sequence PSL activity occurrences, then there can be no temporal properties as parts of fluents. So there is no fluent corresponding the the StateType of Pat's wearing a hat on a Wednesday. (This is a StateType that is instantiated whenever Pat wears a hat on a Wednesday.) Moreover, one can imagine an ontology that is richer in states than in events, where there may be states that are not viewed as brought about as events. Since all fluents either hold initially or are brought about by activity occurrences, we could not have a fluent corresponding to such StateTypes in an ontology like this.

Eventualities

It will be useful to have a term that covers both States and Events. As illustrated above, States and Events occur in English sentences in mostly the same contexts. As Chris Deaton has pointed out, for neither States nor Events can you go directly from an English sentence to a unique instance; you always go via types; both share the property of being nonrepeatable. Both have role relationships with the entities participating in them. Both can be located in time, and at least in the case of physical States and Events, both can be located in space.

There are so far two contenders for a cover term. Hobbs (1985 and subsequently) uses the term "eventuality". It was first used as a cover term for states and events by the linguist Emmon Bach. Cyc uses the term "Situation". Neither word is ideal. The American Heritage Dictionary gives the following definitions for the senses closest to the desired concept:

"eventuality":	Something that may occur; contingency; possibility.
"situation":	A combination of circumstances at a given moment; a state of affairs.

"Eventuality" can be used for both states and events, as seen in the following examples:

> The road might be slippery [state], and in that eventuality I won't know what to do.

> The car might skid [event], and in that eventuality I won't know what to do.

It covers both real and hypothetical events and conditions, which is desirable in the term we are looking for. However, it connotes something in the future, which is not part of the meaning we want.

> ? The road was slippery, and in that eventuality I didn't know what to do.

"Situation" does not have the connotation of something in the future, but it does connote "static", and is thus not a good cover term for events.

> The road was slippery [state], and that situation surprised me.

> ? The car suddenly skidded [event], and that situation surprised me.

Moreover, "situation" is a very loaded term in AI and the philosophy of language, meaning many different things to different people. "Eventuality" carries less of a load.

So in this exposition, we will use the term "Eventuality", but we are open to changing it to something better

As mentioned above, an eventuality can be located in time. We might also want to say it can be located in space, and this is certainly true for physical States and Events. But for more abstract States, it is harder to pin down a location in space. Is the State of America's being a democracy coterminous with the physical location of America?

It seems strange to say

    In Europe, America is not a democracy.

except metonymically. Is John's being retired coterminous with John's body, so that it is not true in my office right now? Where we are dealing with physical events and can locate them in space, one possible interpretation of Eventualities is as Pat Hayes's histories, as seen above, and, also as seen above, this interpretation can often be useful in deciding thorny issues.

The predicates "EventualityType" and "Eventuality" are defined as follows:

```
    (forall (x)
            (iff (EventualityType x)
                 (or (EventType x) (StateType x))))

    (forall (x)
            (iff (Eventuality x) (or (Event x) (State x))))
```

The bridging axioms for Cyc are as follows, although these can be derived from the above two axioms and previous bridging axioms:

```
    (forall (x)
       (iff (cyc:genls x cyc:SituationType)
            (EventualityType x)))

    (forall (e)
       (iff (cyc:isa e cyc:SituationType)
            (Eventuality e)))
```

To bridge between Eventualities in the inter-theory and activities and fluents in PSL would have to be indirect, unpacking Eventualities into Events and/or States and using the bridging axioms for those.

We should keep in mind that the boundary between States and Events (and objects) is not always clear. Is rain a state, an event, or an object? We think of fog as a state and a cloud as an object. English has a way of turning one into the other; "John ran" seems like it describes an event; "John was running" seems like it describes a state.

Preconditions

Overview

We begin with a note on PSL fluents and activities as eventuality types. We then posit a predicate "precondition" that will apply to eventuality types. We define the predicate as it is applied to fluents and activities in terms of the current PSL language. This consititutes the interface between the inter-theory and PSL.

The inter-theory predicate "precondition" applies to types, whereas the Cyc precondition predicates apply to tokens. We introduce a predicate "preconditionToken", relate it to "precondition", and present the axioms that articulate it with the Cyc predicates.

Finally, we present the restrictions on "precondition" that align it with the FLOWS/SWSO precondition predicate.

Eventuality Types, Fluents, and Activities

In the write-up on Fundamentals, we said that a PSL fluent can be viewed as a StateType.

```
    (forall (f) (if (psl:fluent f) (StateType f)))
```

However, we cannot say that that every StateType is a PSL fluent. There we said, "If time is continuous and thus cannot be modelled as a sequence of PSL activity occurrences, then there can be no temporal properties as parts of fluents. So there is no fluent corresponding the the StateType of Pat's wearing a hat on a Wednesday. (This is a StateType that is instantiated whenever Pat wears a hat on a Wednesday.) Moreover, one can imagine an ontology that is richer in states than in events, where there may be states that are not viewed as brought about by events. Since all fluents either hold initially or are brought about by activity occurrences, we could not have a fluent corresponding to such StateTypes in an ontology like this."

Esoteric Example: Let's take as our model one in which eventualities (states and events) are Pat Hayes's chunks of space-time and in which eventuality types are lambda expressions that describe those chunks. A token is an instance of the type if the lambda expression correctly describes the chunk. Fluents are a subset of the lambda expressions, the ones that include no temporal properties.

Warning: This is not _THE_ model for eventuality types and tokens, only a possible model that may sometimes clarify intuitions.

The same issues arise when trying to relate inter-theory (and Cyc) event types and PSL activities. They are not the same, because we can talk about event types that have temporal properties, such as Pat's wife cooking him dinner on a Sunday, which is instantiated every time Pat's wife cooks him dinner and it happens to be Sunday that day.

Just as we stipulated that PSL fluents are state types, we can stipulate that PSL activities are event types.

```
    (forall (a) (if (psl:activity a)(EventType a)))
```

Ontology Mismatches

There are three mismatches between the way PSL and Cyc handle preconditions.

1. PSL has no explicit treatment of preconditions, but it does have the notion of "legal" occurrences of activities.

2. The most straightforward way of defining "precondition" in PSL makes it a relation between a fluent and an activity, whereas Cyc allows any eventuality (Situation) to be a precondition to any other eventuality (Situation).

3. Cyc precondition predicates take tokens rather than types as their arguments.

We will introduce an inter-theory predicate "precondition" that will take eventuality types as its arguments. We will define it in terms of PSL concepts when its arguments are restricted to fluents and activities. We will then relate it to the Cyc precondition predicates.

Preconditions in PSL

The predicate "precondition" takes eventuality types as its arguments.

```
   (forall (e1 e2)
           (if (precondition e1 e2)
               (and (EventualityType e1)(EventualityType e2))))
```

We can define the predicate "precondition" for fluents and activities in terms of PSL predicates. But since the class of event types is larger than the class of activities and since the class of state types is larger than the class of fluents, the axioms will not define "precondition" in general. That can't be done in PSL.

For example, since fluents and activities cannot have temporal qualifiers, we can't state in PSL that Pat's wearing a hat on Wednesday is a precondition for Pat's wife's fixing him dinner on Sunday. We can state in PSL that Pat's wearing a hat is a precondition for Pat's wife's fixing him dinner, but that is not the same, and neither implies the other, since in PSL the precondition has to be true immediately before the event.

Our approach will thus be to _constrain the interpretation_ of the inter-theory predicate "precondition" by these axioms relating it to PSL, even though we can't define it. It will acquire further constraints as we relate it to other existing theories and frameworks.

PSL does not have an explicit predicate expressing a precondition relation between fluents and activity occurrences. But the same work is done in PSL by the predicate "legal". An activity occurrence is legal if it is possible for it to happen in a given state of the world. That is, it is legal if all its preconditions obtain. We can turn this around and say that if a fluent holds prior to all occurrences of some activity and there are no occurrences of that activity when the fluent doesn't hold prior to it, then the fluent is

a precondition for that activity. The reason for our choice of
variable names will emerge below.

```
    (forall (f1 a2)
            (if (and (psl:fluent f1)(psl:activity a2))
                (iff (precondition f1 a2)
                     (forall (o2)
                             (if (and (psl:occurrence_of o2 a2)
                                      (psl:legal o2))
                                 (psl:prior f1 o2))))))
```

This takes care of the case where the first argument is a fluent and
the second an activity. Now we need to handle the other possibilities.
A good way to visualize the cases is by imagining an occurrence of an
activity a1 resulting in a fluent f1, which is a precondition for an
activity a2, which results in a fluent f2. The four cases we need to
consider are then illustrated as follows:

```
    1. (precondition f1 a2):         f1 --> a2

    2. (precondition a1 a2):  a1 --> f1 --> a2

    3. (precondition f1 f2):         f1 --> a2 --> f2

    4. (precondition a1 f2):  a1 --> f1
```

In Case 2 the activity a1 is the only possible cause of the fluent f1,
which is a precondition for a2. The definition in this case is thus
as follows.

```
    (forall (a1 a2)
            (if (and (psl:activity a1)(psl:activity a2))
                (iff (precondition a1 a2)
                     (exists (f1)
                        (and (precondition f1 a2)
                             (forall (o1)
                                (if (psl:achieved f1 o1)
                                    (psl:occurrence_of o1 a1))))))))
```

Note that "(precondition f1 a2)" is defined because it was covered
in Case 1.

In Case 3 the fluent f1 is a precondition for every activity
occurrence that results in fluent f2.

```
    (forall (f1 f2)
            (if (and (psl:fluent f1)(psl:fluent f2))
                (iff (precondition f1 f2)
                     (forall (a2 o2)
                             (if (and (psl:occurrence_of o2 a2)


                                      (psl:achieved f2 o2))
                                 (precondition f1 a2))))))
```

In Case 4 the only way to make fluent f1 hold is by an occurrence of
activity a1.

```
    (forall (a1 f1)
            (if (and (psl:activity a1)(psl:fluent f1))
                (iff (precondition a1 f1)
                     (forall (o)
                             (if (psl:achieved f1 o)
                                 (psl:occurrence_of o a1))))))
```

The extension of "precondition" to Cases 2-4 is not necessitated by the nature of PSL, but by the treatment of preconditions in Cyc.

Articulation with Cyc Predicates

The first problem with linking the predicate "precondition" with Cyc predicates is that the inter-theory "precondition" takes eventuality types as arguments, whereas the Cyc precondition predicates take eventuality (Situation) tokens. So we first introduce a predicate "preconditionToken" that applies to eventuality tokens in an obvious and hopefully correct way. The variable c1 and c2 will be used for types, e1 and e2 for tokens.

If there is a precondition relation between eventuality types, there is a corresponding preconditionToken relation between eventuality tokens.

```
    (forall (c1 c2)
            (if (precondition c1 c2)
                (exists (e1 e2)
                        (and (instanceOf e1 c1)(instanceOf e2 c2)
                             (preconditionToken e1 e2)))))
```

We can't make this axiom an if-and-only-if rule, because the "instanceOf" relations always have to be in the consequent. But we can say that a precondition relation between tokens implies a precondition relation between _some_ pair of types.

```
    (forall (e1 e2)
            (if (preconditionToken e1 e2)
                (exists (c1 c2)
                        (and (instanceOf e1 c1)(instanceOf e2 c2)
                             (precondition c1 c2)))))
```

The constraints on the arguments of "preconditionToken" are as follows:

```
    (forall (e1 e2) (if (preconditionToken e1 e2)
                        (and (Eventuality e1)(Eventuality e2))))
```

We can then relate the Cyc precondition predicates to the inter-theory predicate "preconditionToken".

The Cyc predicate "preconditionFor-Events" is defined in terms of the inter-theory as follows. As before, Cyc predicates are prefixed with "cyc:"; inter-theory predicates have no prefixes.

```
   (forall (?COND ?EVENT)
           (iff (cyc:preconditionFor-Events ?COND ?EVENT)
                (and (Event ?COND)(Event ?EVENT)
                     (preconditionToken ?COND ?EVENT))))
```

Now that we have made this link, we can use Cyc axioms to tell us, for example, that EVENT is not a precondition for COND and that EVENT starts after the beginning of COND. However, see the note below on one of the Cyc axioms.

Cyc has a predicate "situationIsSuchThat" which relates an eventuality/situation to its corresponding proposition. For example, the event of John's running would be linked to the proposition "run(John)", and the state of John's sitting would be linked to the proposition "sit(John)". In a sense, the predicate does the same kind of linking work for us between eventualities/situations and propositions that "fluentFor" does between states and fluents. Rather than invent an inter-theory predicate to do the same thing, we will simply use the Cyc predicate.

The Cyc predicate "preconditionFor-SitProp" is a relation between an eventuality/situation and a proposition. It says that the eventuality's occurrence is a precondition for the proposition's being true. We coerce the proposition into the corresponding situation.

```
   (forall (?PROP ?SIT2)
           (iff (Cyc:preconditionFor-SitProp ?SIT1 ?PROP)
                (exists (?SIT2)
                        (and (Cyc:situationIsSuchThat ?SIT2 ?PROP)
                             (preconditionToken ?SIT1 ?SIT2)))))
```

The Cyc predicate "preconditionFor-PropSit" is a relation between a proposition and an eventuality/situation. It says that proposition's being true is a precondition for the eventuality.

```
   (forall (?PROP ?SIT2)
           (iff (Cyc:preconditionFor-PropSit ?PROP ?SIT2)
                (exists (?SIT1)
                        (and (Cyc:situationIsSuchThat ?SIT1 ?PROP)
                             (preconditionToken ?SIT1 ?SIT2)))))
```

Note on the Subevent Axiom for "preconditionFor-Events": The axiom

```
   (if (and (Cyc:preconditionFor-Events ?COND ?EVENT)
            (Cyc:subEvents ?EVENT ?SUB))
       (Cyc:preconditionFor-Events ?COND ?SUB))
```

is probably not correct, given the most likely interpretations of the predicates. Consider a composite event StandUp-SitDown, which is comprised of a StandUp followed by a SitDown. Both the StandUp and the SitDown would be subevents of the composite event. A precondition for the composite event would be that one has to be sitting down. But this is not a precondition for the subevent SitDown; quite the contrary.

Articulation with FLOWS/SWSO

In FLOWS/SWSO, preconditions are characterized as follows:

> "A precondition of an atomic process is a formula that states that the atomic process cannot be executed until this formulae [sic] holds."

The specification of preconditions is restricted to atomic processes because of the difficulty in maintaining consistency among the preconditions of a composite process and those of the atomic processes of which it is comprised (cf. the Cyc subevent axiom).

First we need to spell out in the inter-theory what an atomic process is. It is at least a process or event which does not have subevents. A consequence of this, that should follow once we have developed a treatment of failures and interruptions, is that an atomic process cannot fail or be interrupted. We employ the predicate "subevent" that will be explicated in subsequent IKRIS Scenarios notes; "(subevent e1 e2)" means that e1 is a subevent of e2.

```
   (forall (e)
           (iff (atomic e)
                (and (Event e)
                     (forall (e1) (not (subevent e1 e))))))
```

One may also want to say that atomic events are instantaneous. This seems to be controversial, so we will place a trigger condition on it.

```
   (forall (e)
           (if (and (AtomicInstantaneous)(atomic e))
               (exists (t)
                       (and (t:instant t)(t:timeSpanOf t e)))))
```

The FLOWS precondition predicate is between a formula, i.e., a proposition, and a process, i.e., an event. To coerce from the proposition to the corresponding state, we will use the Cyc predicate "situationIsSuchThat".

```
   (forall (e p)
           (iff (flows:precond e p)
                (exists (e1 c1)
                        (and (atomic e)
                             (cyc:situationIsSuchThat e1 p)
                             (instanceOf e1 c1)
                             (precondition c1 e)))))
```

That is, to do the mapping between the FLOWS predicate "precondition" and the inter-theory predicate "precondition", we have to restrict the process argument to atomic processes and coerce the FLOWS proposition into an inter-theory eventuality type. Both the type c1 and the token e1 are introduced because the Cyc predicate situationIsSuchThat maps propositions into eventuality _tokens_ and it seems safer to relate the FLOWS/SWSO precondition predicate to the basic inter-theory predicate "precondition".

Effects

Overview

In this section we introduce the predicate "effect" which is a relation between two eventuality types. In a manner analogous to our treatment of preconditions, we link it with PSL by extending it from an "achieved" relation between an activity occurrence and a fluent to a relation between eventuality types in general. Cyc has predicates corresponding to both type-type effect or causality and token-token causality. They are weakly related in Cyc. In this note we strengthen that relation somewhat, in the interests of constraining the interpretations of our predicates as much as possible. Finally, we present the restrictions on the FLOWS/SWSO "effect" predicate and define it in terms of PSL predicates.

We will use the predicate "effect", which takes eventuality types as its arguments.

```
   (forall (e1 e2)
           (if (effect e2 e1)
               (and (EventualityType e1)(EventualityType e2))))
```

The expression "(effect e2 e1)" says that eventualities of type e1 have effects of type e2, i.e., that an e2-type state or event is an effect of an e1-type state or event.

Note that the order of arguments follow the order in the English sentence "e2 is an effect of e1" rather than in the causal/temporal order "e1 then e2".

Effects in PSL

Recall that the class of state types properly includes the class of fluents. So when the "effect" predicate is used in PSL for state types, it will be restricted to fluents.

In PSL, the predicate "achieved" between a fluent and an activity occurrence means that the fluent did not hold before the occurrence and did hold after it. This is the key property of effects.

```
    (forall (f a)
            (if (and (psl:fluent f)(psl:activity a))
                (iff (effect f a)
                     (forall (o)
                             (if (psl:occurrence_of o a)
                                 (psl:achieved f o))))))
```

That is, a fluent is an effect of an activity if and only if any occurrence of the activity achieves the fluent. An example of this is when an activity of a coffee cup falling to the floor has the effect that the coffee cup is on the floor.

This takes care of the case where the first argument is a fluent and the second an activity. Now we need to handle the other possibilities.

We often talk about activities or events being the effect of a fluent. For example, an effect of the slipperiness of the floor is John's falling. For an activity to be the effect of a fluent, it must be the inevitable outcome of that fluent. That is, activity a is an effect of fluent f provided whenever f holds, the only legal activity occurrences are those that have a as a subactivity. (This allows other things to happen concurrently.)

```
    (forall (a f)
            (if (and (psl:activity a)(psl:fluent f))
                (iff (effect a f)
                     (forall (o a1)
                             (if (and (psl:prior f o)(psl:legal o)
                                      (psl:occurrence_of o a1))
                                 (psl:subactivity a a1)
```

That is, the only things that can happen after f are occurrences of activities that include a.

An activity can be an effect of an activity. My swinging my arm can have the effect of my coffee cup falling to the floor. This can be captured by hypothesizing an intermediate fluent that causes the second activity.

```
    (forall (a1 a2)
            (if (and (psl:activity a1)(psl:activity a2))
                (iff (effect a2 a1)
                     (exists (f)
                             (and (psl:fluent f)(effect f a1)
                                  (effect a2 f))))))
```

That is, fluent f is an effect of activity a1 and activity a2 is an effect of f.

Finally, a fluent can have a fluent as an effect. The slipperiness of the ice can have John's leg being broken as an effect. This generally holds because there is an intervening activity.

```
    (forall (f1 f2)
            (if (and (psl:fluent f1)(psl:fluent f2))
                (iff (effect f2 f1)
                     (exists (a)
                             (and (psl:activity a)(effect a f1)
                                  (effect f2 a))))))
```

Activity a is an effect of f1 and f2 is an effect of a.

Articulation with Cyc

Cyc has a predicate that directly corresponds to "effect" -- "causesSitTypeSitType". So we can interdefine the two predicates:

```
    (forall (e1 e2)
            (iff (effect e2 e1)
                 (cyc:causes-SitTypeSitType e1 e2)))
```

Cyc also has a predicate that applies to eventuality tokens -- "causes-SitSit". It is not _defined_ (iff) in terms of "causesSitTypeSitType", but its meaning is _constrained_ (implies) by the Cyc axiom

```
    (forAll ?X
     (forAll TYPE1
      (forAll TYPE2
         (implies (and (cyc:causes-SitTypeSitType ?TYPE1 ?TYPE2)
                       (cyc:isa ?X ?TYPE1))
                  (thereExists ?Y
                      (and (cyc:isa ?Y ?TYPE2)
                           (cyc:causes-SitSit ?X ?Y)))))))
```

Or in terms of the inter-theory predicate "effect" we have the axiom

```
    (forall (c1 c2 e1)
            (if (and (effect c2 c1)(instanceOf e1 c1))
                (exists (e2)
                        (and (instanceOf e2 c2)
                             (cyc:causes-SitSit e1 e2)))))
```

As with preconditions, we can't state an if-and-only-if relation between "effect"/"causesSitTypeSitType" and "causes-SitSit". But we we would like to say that for any case of causality between particulars, there is a corresponding causal regularity between types that it instantiates.

However, we cannot state this in the way that might occur to us initially, saying that corresponding to a token-token causal relation, there is a type-type causal relation between types of those tokens. For example, we may say that someone's running with scissors had the effect that his face was cut. But it is not the case that every token of a running-with-scissors event type causes a token of a face-cutting event type.

We can get a clearer picture of what we want by examining it in terms of the framework in Hobbs (2005). Briefly, there is a distinction between the monotonic notion of a causal complex, and the nonmonotonic notion of "cause". The causal complex consists of everything that must hold or happen in order for the effect to happen, and the effect always occurs if the whole causal complex occurs. (The causal complex also contains only eventualities relevant to the effect, in a way that can be defined.) Out of all the states and events in the causal complex, we often pick one or several that count as "causes", for a variety of reasons, e.g., they don't normally occur, or they require some action on our part to make them occur. The trouble with causal complexes is that we almost never know them completely. So for commonsense reasoning we make do with mere causes. In AI, the preconditions plus the body of a planning operator constitute an attempt to capture the notion of a causal complex, and the body would normally correspond to the cause.

When planning what actions to take in a given situation, we make use of causal relations between types of causal complexes and types of

effects. We want to instantiate the causal complex in order to get an instance of the effect. But most of our causal knowledge is not about causal complexes but about "causes". So we instantiate the event type that is the "cause" and expect to get an instance of the effect. But sometimes things misfire because the rest of the causal complex is not in place.

When seeking to explain states or events that have actually occurred, we could in principle discover the entire causal complex that had the state or event as its effect. If we did, then we would have a causal complex-effect relation among types that was instantiated in tokens that actually occurred. But again, we normally won't be able to identify the entire causal complex, so we just identify the "cause". And here there won't necessarily be a type relation between the cause and effect.

In the scissors example, the causal complex contains not just the running with scissors event but also the state of the scissors being pointed toward the person's face just before contact. We can see the incident as a token of a causal relation between types, but one of those types is the whole causal complex, not just the state or event we picked out as the "cause".

Thus, the proper implication from token causality to type causality is

```
     (forall (e1 e2)
             (if (cyc:causes-SitSit e1 e2)
                 (exists (c1 c2 s)
                         (and (instanceOf e1 c1)(instanceOf e2 c2)
                              (member c1 s)(effect c2 s)))))
```

That is, the type c1 of the "cause" e1 is a member of a causal complex type s that has the type c2 as its effect, where the effect token is an instance of c2. This would accommodate the scissors example, but still would not allow causation strictly between particulars, with no support from a causal regularity.

Articulation with FLOWS/SWSO

In FLOWS/SWSO, effects, like preconditions, only apply to atomic processes. We defined "atomic" there. Effects can be conditional. The condition is a fluent that holds before the atomic process is executed and the effect occurs only if the condition holds. For example, the effect may be that your credit card balance is debited, and the condition is that your credit card has not expired. The FLOWS/SWSO predicate "effect" can be defined naturally in terms of PSL predicates.

```
    (forall (a f1 f2)
            (if (and (atomic a)(psl:fluent f1)(psl:fluent f2))
                (iff (flows:effect a f1 f2)
                     (forall (o)
                             (if (and (psl:occurrence_of o a)
                                      (psl:prior f1 o))
                                 (psl:achieves o f2))))))
```

Inputs and Outputs as Preconditions and Effects

The purpose of this section is to reduce the IOPE problem to the PE problem.

If we can define inputs and outputs in terms of preconditions and effects, then we can work on interoperability for only the latter concepts. In this note, until the final paragraph we are only talking about informational inputs and outputs, not consumable physical resources, e.g., in a manufacturing process.

In the write-up on preconditions, we introduced a "precondition" predicate that takes eventuality types as its arguments.

```
    (precondition e1 e2)
```

Suppose we also have a similar "effect" predicate:

```
    (effect e1 e2)
```

There is a distinction between an action and the agent of the action. Agents persist through time, whereas actions are PSL's activity occurences or Cyc's events (event tokens). The relation between the agent and the action is

```
    (agentOf a p)
```

i.e., agent a is the agent of action p. We will assume there is a population of agents, but we relegate the details of what counts as an agent to another ontology.

Inputs and outputs can then be grounded in an account of messages among agents. We will assume we have an ontology of messages (prefix "m:") which provides a primitive notion of "message". The expression

```
    (m:message m a b x y)
```

says that m is a messaging act in which a communicates to b the information y via the physical object or signal x. That is, x is the message or information-bearing object that is sent and y is its content. The separate ontology of messages needs to be developed. Cyc has one; Hobbs (2005) presents another one. But that is out of scope. Here we will say nothing about the nature of y; it may be a proposition or some nonpropositional concept.

We will assume that the underlying ontology of messages explicates two properties of messaging events:

```
    (m:sent s m)
    (m:received r m)
```

The first says that s is the state type of the originator of the messaging act m having done its part in sending the message. The second says that r is the state type of the messaging act being complete in that the content of the message has been received. We define these in terms of state types because that's what the "precondition" predicate requires.

In order to be an agent capable of sending and receiving messages, the agent has to have the capability of using the information in some fashion. We will say that when an agent is in some sense in possession of the content of the message, then that content is "availableTo" the agent.

```
    (m:availableTo y a t)
```

This concept is related to "sent" and "received" by the following axioms:

```
    (forall (m a b x y s t)
            (if (and (m:message m a b x y)
                     (m:sent s m)
                     (instanceOf e s)
                     (t:atTime e t))
                (m:availableTo y a t)))
```

That is, if an instance e of a state s of a message being sent holds or obtains at time t, where the message is sent by a and has content y, then y is available to a at time t.

```
    (forall (m a b x y r t)
            (if (and (m:message m a b x y)
                     (m:received r m)
                     (instanceOf e r)
                     (t:atTime e t))
                (m:availableTo y b t)))
```

That is, if an instance e of a state r of a message being received holds or obtains at time t, where the message is received by b and has content y, then y is available to b at time t.

If the agent has some sort of "cognitive state" and can be said to "know" things, then we can say if the agent knows some information, the information is available to the agent.

```
    (forall (a y t)
            (if (know a y t)
                (availableTo y a t)))
```

This axiom makes sense of the impulse to treat inputs and outputs as knowledge preconditions and effects. But using the broader concept of availability overcomes qualms about referring to knowledge when talking about very simple processes.

Inputs and outputs can be defined in terms of messages sent and received as preconditions and effects. They will both be relations between some kind of content and an eventuality type.

```
    (forall (y p)
            (if (input y p)(Eventuality p)))

    (forall (y p)
            (if (output y p)(Eventuality p)))
```

We have omitted constraints on y because explicating a theory of possible contents of messages would take us too far afield.

The definition of "input" is as follows:

```
    (forall (b p y)
            (if (agentOf b p)
                (iff (input y p)
                     (exists (m a x r)
                             (and (m:message m a b x y)
                                  (m:received r m)
                                  (precondition r p))))))
```

That is, content y is input to process or eventuality p if and only if there is a message event m from some a to the agent b of p in which the message is x and its content is y, there is the state type r of that message having been received, and r is a precondition for p.

The definition of "output" is as follows:

```
    (forall (a p y)
            (if (agentOf a p)
                (iff (output y p)
                     (exists (m b x s)
                             (and (m:message m a b x y)
                                  (m:sent s m)
                                  (effect s p))))))
```

That is, content y is output to process or eventuality p if
and only if there is a message event m from the agent a of p to some b
in
which the message is x and its content is y, there is the state type s
of that message having been sent, and s is an effect of p.

Defining inputs and outputs in terms of messages rather than in terms of availability or knowledge takes care of the case where the agent already knows the supposed "input" or does not reveal the supposed "output". It's not input unless someone puts it in, and it's not output unless the agent puts it out.

Consider an example that is as simple as McCarthy's thermostat. The agent is a calculator (or abacus even) capable of computing sums;

that's the action. The message is the user typing in the numbers (or moving the right number of beads). The output is the display. "Availability" for the calculator is simply having the received the input numbers and calculated the output number. For the abacus, availability is having a representation of a number on its display.

Most examples will be more interesting.

Optional inputs can be accommodated by extending the domain over which the y argument ranges, to include a symbol whose meaning is "null". When the input y is not null, the option is exercised. When it is null, the option is not. The transmission of a null message is a definite event, and not the same as the absence of a message. To say that an input is optional is to say that "null" is one possible value of y. To say that it is obligatory is to say that y cannot be "null". So suppose process p has two integer inputs, an optional y and an obligatory z. Then the precondition is

```
    (and (input y p)(input z p)
         (or (integer y)(= y null))
         (integer z))
```

Inputs and outputs can interact with other preconditions and effects. Suppose a process requires a credit card number y as input and then has to check that it is unexpired (at time t). The precondition is

```
    (and (input y p)(unexpired y))
```

We have defined inputs and outputs here in terms of messages. In any given system or framework, it is perfectly possible to treat "input" and "output" as primitive concepts. Grounding the concepts in messages in the inter-theory will enable such a system to interoperate with systems or frameworks that use a similar grounding or that have no notion of inputs and outputs, only preconditions and effects.

A question arises as to what relation there is between these informational inputs and outputs and material inputs and outputs in manufacturing processes. For example, one might call steel one of the "inputs" to a vehicle-manufacturing process and SUVs as one kind of "output". This is a different notion than what we have explicated here. However, a theory of physical inputs and outputs will be analogous to our account here, with a predication like

```
    (move m x a b)
```

replacing

```
    (message m a b x y)
```

The former expression says that m is a moving act in which x is moved from a to b. Corresponding to the informational notion of availability is the physical notion of "locationAt". In fact, our way of conceiving knowledge and communication among agents is generally via a spatial metaphor resting on the identification of "availability" or "knowing" with "locationAt". It is not surprising that the two theories will be analogous. Of course, one distinction between the two theories is that physical inputs are consumed by physical processes whereas informational inputs are not consumed by their processes.

SPARK as a Declarative Representation

The aim of this section is to show that the ostensibly procedural language SPARK can be viewed declaratively, and thus as something that doesn't just execute, but also is a means of encoding information about the structure of processes. This will enable us in subsequent work to relate SPARK with PSL, ResearchCyc and other event representation frameworks by means of the same sort of IKL articulation axioms we have been constructing so far. This will be particularly valuable since SPARK is relatively rich in control structures, or possibilities for the internal structure of events.

Consider a simple SPARK procedure:

```
   {defprocedure reportSpam                                     (1)
      cue:  [do: (forwardMessage $message)]
      precondition:  (IsSpam $message)
      body:  [do: (sendTo SpamCollection $message)]}
```

We would like to turn this into the following declarative representation in IKL:

```
   (and (procname e reportSpam)                                 (2)
        (cue e1 e) (forwardMessage' e1 $message)
        (precondition e2 e) (IsSpam' e2 $message)
        (body e3 e) (sendTo' e3 SpamCollection $message))
```

This relies on the reification of states and events, where if

```
   (see Pat Kim)
```

means that Pat sees Kim, then

```
   (see' e Pat Kim)
```

means that e is the event of Pat seeing Kim.

So the above IKL statement says that e has the procedure name "reportSpam", e1 is a cue for e where e1 is the action of forwarding $message, e2 is a precondition for e where e2 is the state of $message being spam, and e3 is the body of e where e3 is the action of sending $message to SpamCollection.

We could also reify the states of something being a cue, precondition, or body of a procedure as well: (cue' e0 e1 e) says that e0 is the state or property of e1 being a cue for e.

Translation Rules

The following context-dependent translation rules effect this translation. Here TR<...> is the translation function, acting recursively on SPARK expressions. The asterisk * is used to indicate zero or more instances of the string it follows, and the expressions on the right and left side of the rule are kept in sequence. I've used / and \ as metalanguage brackets, just because I ran out of other brackets. The vertical bar | is a way of keeping things in place until they have been translated. After translation is complete, they are removed by the following rule:

```
   (p x|(q y)*) ==> (and (p x) (q y)*)                          (3)
```

where x and y stand for any sequence of arguments. "tag" stands for anything that can precede a colon in SPARK, such as "cue", "precondition", "body", and "do". Variables of the form "e" or "en", for some number n, appearing on the right side of a rule and not on the left are new variables.

```
1. TR<{defprocedure name defn}> ==> (procname e|TR<defn> name)

2. e|TR</tag: expr\*>           ==> e|(tag TR<expr> e)*

3. TR<[tag: expr*]>             ==> e1|(tag' e1 TR<expr>*)

4. TR<(p x*)>                   ==> e|(p' e TR<x>*)

5. TR<constant>                 ==> constant

6. TR<variable>                 ==> variable

7. TR<[variable]>               ==> variable
```

Here's a detailed example that illustrates this translation. You can skip to the semantics section if you believe the rules already.

Applying Rule 1 to procedure definition (1)

```
   TR<{defprocedure reportSpam
         cue:  [do: (forwardMessage $message)]
         precondition:  (IsSpam $message)
         body:  [do: (sendTo SpamCollection $message)]}>
```

yields

```
   (procname e|TR<cue:  [do: (forwardMessage $message)]
                  precondition:  (IsSpam $message)
                  body:  [do: (sendTo SpamCollection $message)]>
             reportSpam)
```

Applying Rule 2 to this yields

```
   (procname e|(cue TR<[do: (forwardMessage $message)]> e)
               (precondition TR<(IsSpam $message)> e)
               (body TR<[do: (sendTo SpamCollection $message)]> e)
```

```
             reportSpam)
```

Applying Rule 3 to this yields

```
   (procname e|(cue e1|(do' e1 TR<(forwardMessage $message)>) e)
               (precondition TR<(IsSpam $message)> e)
               (body e2|(do' e2 TR<(sendTo SpamCollection $message)>)
e)
             reportSpam)
```

Applying Rule 4 to this yields

```
   (procname e|(cue e1|(do' e1 e3|(forwardMessage' e3 TR<$message>)) e)
               (precondition e4|(IsSpam' e4 TR<$message>) e)
               (body e2|(do' e2 e5|(sendTo' e5 TR<SpamCollection>
                                            TR<$message>)) e)
             reportSpam)
```

Applying Rules 5 and 6 to this yields

```
   (procname e|(cue e1|(do' e1 e3|(forwardMessage' e3 $message)) e)
               (precondition e4|(IsSpam' e4 $message) e)
               (body e2|(do' e2 e5|(sendTo' e5 SpamCollection
                                            $message)) e)
             reportSpam)
```

Now we've translated all the way down to the bottom, and we can use
(3) to unwind this successively into a conjoined expression.

```
   (procname e|(cue e1|(and (do' e1 e3)(forwardMessage' e3 $message))
e)
               (and (precondition e4 e)(IsSpam' e4 $message))
               (body e2|(and (do' e2 e5)(sendTo' e5 SpamCollection
                                                 $message)) e)
             reportSpam)

   (procname e|(and (cue e1 e)(and (do' e1 e3)
                                   (forwardMessage' e3 $message)))
               (and (precondition e4 e)(IsSpam' e4 $message))
               (and (body e2 e)(and (do' e2 e5)
                                    (sendTo' e5 SpamCollection
$message)))
             reportSpam)

   (and (procname e reportSpam)
        (and (cue e1 e)(and (do' e1 e3)(forwardMessage' e3 $message)))
        (and (precondition e4 e)(IsSpam' e4 $message))
        (and (body e2 e)(and (do' e2 e5)
                             (sendTo' e5 SpamCollection $message))))
```

Then flattening out the and's gives us

```
   (and (procname e reportSpam)
        (cue e1 e)(do' e1 e3)(forwardMessage' e3 $message)
        (precondition e4 e)(IsSpam' e4 $message)
        (body e2 e)(do' e2 e5)(sendTo' e5 SpamCollection $message))
```

which except for the do's is the same as expression (2). More about "do" below.

Call this form SPARK Expressed Declarative, or SPARKED.

The Semantics of SPARKED

We introduce two predicates that describe the world and the action of the SPARK processor on the world.

```
   holds(State):    true iff the State holds
   executed(Task):  true iff the Task is executed by the processor
```

Then we can constrain the meanings of the top-level tags by the following axioms:

```
   (forall (e1 e)
           (if (and (cue e1 e)(executed e))

               (executed e1)))
```

That is, if e1 is a cue for e and e is executed, then e1 is executed.

```
   (forall (e2 e3 e)
           (if (and (precondition e2 e)(body e3 e)(holds e2)
                    (executed e3))
               (executed e)))
```

That is, if the precondition for a procedure holds and its body is executed, the procedure is executed.

The above translation rules translate logical operators into primed predicates. The semantics of the primed predicates can be defined as follows:

```
   (forall (e e1 e2)
           (if (and' e e1 e2)
               (iff (holds e)
                    (and (holds e1)(holds e2))))

   (forall (e e1 e2)
           (if (or' e e1 e2)
               (iff (holds e)
                    (or (holds e1)(holds e2)))))

   (forall (e e1)
           (if (not' e e1)
               (iff (holds e)
                    (not (holds e1)))))
```

Basic Task Components

Similar rules can be defined for the basic task components, such as do:, achieve:, and so on.

```
   (forall (e1 e2)
           (if (do' e1 e2)
               (iff (executed e1)(executed e2))))
```

For example, in the SPARK expression "[do: (paint $house red)]", the doing is executed if and only if the painting is executed.

```
   (forall (e1 e2)
           (if (achieve' e1 e2)
               (iff (executed e1)(holds e2))))
```

That is, an achieve is executed if and only if its operand holds.

```
   (forall (e1)
           (if (noop' e1)
               (executed e1)))
```

A noop is always executed.

The "fail:" task component requires a little more to be explicit than we have so far. I will assume for now that when we encounter "fail'" in declarative form, it has an argument for the fail event e1, an argument e2 for a condition under which the failure occurs, and the procedure e in which it is embedded, without saying how we would get the last of these in the translation process (e.g., a global variable).

```
   (forall (e1 e2 e)
           (if (fail' e1 e2 e)
               (iff (executed e1)
                    (and (holds e2)
                         (not (executed e))))))
```

That is, the fail is executed exactly when the condition e2 holds and the embedding procedure e is not executed.

The operators "conclude:" and "retract:" presuppose a knowledge base. I will use the predicate "known'" to represent the presence of some fact p in the knowledge base. Thus, "(known' e p)" says that e is the state of p being in the knowledge base. I'll say nothing about the nature of p. I will assume that if we conclude something that is already in the knowledge base, no change occurs, and similarly with retraction.

```
   (forall (e1 p t1)
           (if (conclude' e1 p)
               (if (and (executed e1)(t:timeSpanOf t1 e1))
                   (exists (e2 t2) (and (known' e2 p)
                                        (t:timeSpanOf t2 e2)
                                        (or (t:intMeets t1 t2)
                                            (t:intDuring t1 t2)))))))
```

That is, if e1 is a concluding of p, then immediately after the concluding, p is known.

The axiom for retracting is similar.

```
   (forall (e1 p t1)
           (if (retract' e1 p)
               (if (and (executed e1)(t:timeSpanOf t1 e1)
                        (t:ends t0 t1))
                   (forall (e2 t2)
                           (if (and (known' e2 p)
                                    (t:timeSpanOf t2 e2))
                               (not (or (t:starts t0 t2)
                                        (t:inside t0 t2))))))))
```

Here, the endpoint t0 of the execution of the retraction cannot start or be inside any interval during which p is known.

Control Structures

The control structures or compound task expressions can also be defined or constrained in terms of what counts as their being executed

```
   (forall (e e1 e2 t1 t2)
           (if (and (seq' e e1 e2)(t:timeSpanOf t1 e2)
                    (t:timeSpanOf t2 e2))
               (iff (executed e)
                    (and (executed e1)(executed e2)
                         (t:intBefore t1 t2)))))
```

It's not clear to me that "parallel:" means anything. For example, suppose Chris gives a task to me and a task to Karen to execute in parallel. Karen does her task right away. I put mine off til tomorrow. Have we done the tasks in parallel or not? Does there have to be an overlap in the time span of the two executions? A weak notion of "parallel" is given by this axiom:

```
   (forall (e e1 e2)
           (if (parallel' e e1 e2)
               (iff (executed e)
                    (and (executed e1)(executed e2)))))
```

The stronger notion of "parallel" requires some overlap in the executions.

```
   (forall (e e1 e2 t1 t2)
           (if (and (parallel' e e1 e2)(t:timeSpanOf t1 e2)
                    (t:timeSpanOf t2 e2))
               (iff (executed e)
                    (and (executed e1)(executed e2)
                         (exists (t)(and (t:inside t t1)
                                         (t:inside t t2)))))))
```

Conditionals are defined as follows:

```
   (forall (e e1 e2 e3)
           (if (if' e e1 e2 e3)
               (iff (executed e)
                    (or (and (holds e1)(executed e2))
                        (and (not (holds e1))(executed e3))))))
```

The expression "[wait:e1 e2]" means that the processor waits until e1 holds and then executes e2. It can be defined as follows:

```
   (forall (e e1 e2 t1 t2)
           (if (and (wait' e e1 e2)(t:timeSpanOf t1 e1)
                    (t:timeSpanOf t2 e2))
               (iff (executed e)
                    (and (executed e2)(intBefore t1 t2)))))
```

The expression "[try: e1 e2 e3]" means that the processor tries to do e1. If it succeeds, it goes on to do e2; otherwise, it does e3. For example, e3 may be the task of fixing what went wrong in doing e1. This operator cannot be defined with what has been introduced so far, because it involves the partial execution of e1, and we have no way of talking about partial executions yet. One approach would be to define the notion of "subevent". (We will want it eventually, in any case.) Then we can say that if a subevent of e1 is executed but e1 itself is not, then this will count as failure and e3 will be executed. Under this interpretation, the definition of trying is as follows:

```
   (forall (e e1 e2 e3)
           (if (try' e e1 e2 e3)
               (iff (executed e)
                    (or (and executed e1)(executed e2))
                        (exists (e0)
                                (and (subevent e0 e1)(executed e0)
                                     (not (executed e1))
                                     (executed e3))))))
```

The subevent relation can be characterized in terms of the basic and

compound task expressions and a transitivity axiom.

```
   (forall (e1 e2 e3)
           (if (and (subevent e1 e2)(subevent e2 e3))
               (subevent e1 e3)))

   (forall (e e1)
           (if (body e1 e)(subevent e1 e)))

   (forall (e1 e2)

           (if (do' e1 e2)(subevent e2 e1)))

   (forall (e e1 e2)
           (if (seq' e e1 e2)
               (and (subevent e1 e)(subevent e2 e))))
   (forall (e e1 e2)
           (if (parallel' e e1 e2)
               (and (subevent e1 e)(subevent e2 e))))

   (forall (e e1 e2)
           (if (if' e e1 e2)
               (and (subevent e1 e)(subevent e2 e))))
```

However, we may want to say that something is not a subevent unless it is actually executed. There are similar concerns for the next two axioms.

```
   (forall (e e1 e2)
           (if (wait' e e1 e2)(subevent e2 e)))

   (forall (e e1 e2)
           (if (if' e e1 e2 e3)
               (and (subevent e1 e)(subevent e2 e)(subevent e3 e))))
```

Iteration

To characterize an iteration we need a specification of the eventuality type that is being repeated and the set or sequence whose members are being iterated over. We need to be able to represent not only that on each iteration an instance of that eventuality type is occurring, but that that instance is obtained by "substituting" the next member of the set or sequence for the corresponding parameter in the eventuality type. For example, suppose a set of researchers takes turns getting up to give a talk. The eventuality type is "Someone gives a talk." Some of the eventuality tokens that are instances of the type might be "Chris gives a talk," "Jerry gives a talk," and "Karen gives a talk."

I will use the predication "(subst x e1 y e2)" to mean that parameter x plays the same role in eventuality type e1 that entity y plays in eventuality token e2. For example, if "(talk' e1 x)" and "(talk' e2 Chris)" both hold, then so does "(subst x e1 Chris e2)". The predicate "subst" is defined in Hobbs (1995), ``Monotone Decreasing Quantifiers in a Scope-Free Logical Form'', found at http://www.isi.edu/~hobbs/monotone-decreasing.pdf, or at http://www.isi.edu/~hobbs/disinf-chap2.pdf, pp. 46-49.

In fact, e1 and e2 need not be restricted to eventuality types and tokens, respectively. Thus, if in addition "(talk' e3 Jerry)" holds, then so does "(subst Chris e2 Jerry e3)". However, if e1 is an eventuality type and e2 an eventuality token and the "subst" relation holds between them, then the "subst" relation is a specialization of the "instanceOf" relation. I won't write this as an axiom since I would first have to have a way of saying x is a parameter, and that would take us too far afield.

The SPARK compound task expression "[forall: x e1 e2]" translates into "(forall' e x e1 e2)". The semantics of this is given by the following axiom:

```
   (forall (e x e1 e2)
           (if (forall' e x e1 e2)
               (iff (executed e)
                    (forall (y e3)
                            (if (and (subst x e1 y e3)(holds e3))
                                (exists (e4)
                                        (and (subst x e2 y e4)
                                             (executed e4))))))))
```

The SPARK compound task expression "[while: e1 e2 e3]" means that the processor repeats e2 as long as e1 is true and then does e3. The condition e1 and the task e2 must be eventuality types, but we will assume that an eventuality type is executed when one of its instances is executed, and that an eventuality type holds when one of its instances holds. Then the translated expression "(while' e e1 e2 e3)" can be defined recursively as follows:

```
   (forall (e e1 e2 e3)
           (if (while' e e1 e2 e3)
               (iff (executed e)
                    (or (and (not (holds e1))(executed e3))
                        (exists (e4 e5)
                                (and (holds e1)
                                     (seq' e4 e2 e5)
                                     (while' e5 e1 e2 e3)
                                     (executed e4)))))))
```

The first disjunct says that e3 is executed if e1 does not hold. The second disjunct says that if e1 does hold, then a sequence is executed, consisting of e2 followed by another while loop.

References

Hobbs, Jerry R., 2005. ``Toward a Useful Notion of Causality for Lexical Semantics'', Journal of Semantics, Vol. 22, pp. 181-209. (http://www.isi.edu/~hobbs/causality-jos.pdf)

Hobbs, Jerry R., 2005. ``An Ontology of Information Structure", Proceedings, 7th International Symposium on Logical Formalizations of Commonsense Reasoning, Corfu, Greece, pp. 99-106, May 2005.