Design of a Process Ontology: Vocabulary, Semantics, and Usage

Stuart Aitken[1] and Jon Curtis[2]

[1] Artificial Intelligence Applications Institute,
University of Edinburgh
stuart@aiai.ed.ac.uk
http://www.aiai.ed.ac.uk
[2] Cycorp Inc., 3721 Executive Center Drive,
Austin, Texas 78731
jonc@cyc.com
http://www.cyc.com

Abstract. This paper describes an ontology for process representation. The ontology provides a vocabulary of classes and relations at a level above the primitive event-instance, object-instance and timepoint description provided by ontologies such as the Process Specification Language. The design of this ontology balances two main concerns: to provide a concise set of useful abstractions of process, and to provide an adequate formal semantics for these abstractions. The aim of conciseness is to support knowledge authoring - ideally a domain expert should be able to author knowledge in the ontology - providing a sufficiently advanced toolset and interface has been implemented to support this task.

1 Introduction

The Rapid Knowledge Formation project (RKF) [8] aims to develop powerful tools to enable domain experts to author knowledge directly. Ideally, the role of the knowledge engineer is confined to the design of generic templates for complex patterns of knowledge, such as processes. The design of specific tools, such as those for process knowledge, is complemented by tools supporting the more typical types of knowledge specified in the ontology, namely, classes, relations and rules. These tools also require knowledge-engineering knowledge in order to function adequately. In practice, the use of the two types of tools is interconnected, e.g. defining a new process will involve defining a new collection to represent the event-type.

This paper describes an ontology for process representation which allows processes to be described purely at the type-level. The semantics of these relations is expressed in terms of more primitive event-to-object relations in the Cyc ontology[6, 7]. A similar semantics in terms of the Process Specification Language (PSL)[11] types and relations has also been developed in the course of this work showing the (relative) independence of the type level and the ground level.

The process ontology aims to provide a concise set of useful abstractions of process which apply across numerous domains. The first domain studied was processes in cell biology, where models were derived from a textbook [1]. Latterly, we are considering military courses of action. The process ontology augments an existing theory of scripted events in Cyc, which is the theory behind the powerful user interface tools developed and tested in RKF. And with the release of an open-source version of the the Cyc system, the design of Cyc ontologies and tools becomes more relevant to the AI community.

The connection between the ontology and the user tools is the subject of Section 2, we then outline the existing Script vocabulary and then present the extensions to it. The theories of participants, conditions and effects and of repeated processes are documented in Section 3. An example of a complex process represented in this framework is given in Section 4. Finally, we consider related work in Section 5, and draw some conclusions in Section 6.

2 The RKF Tools and Ontology

The Cyc knowledge base currently contains more than 100,000 concepts, and 1.4 million axioms and rules [9]. Cyc's knowledge is represented using CycL, a highly expressive language based on second order logic. Assertions are made in a specific context, known as a Microtheory. The Cyc tools developed in the RKF project provide the core functionality of the KRAKEN knowledge-entry system. KRAKEN incorporates powerful natural language tools that allow the user to interact with the system through simple questions and statements in English. As CycL is natural-language independent, the parsing components of the interface use Cyc's lexical and syntactic knowledge to produce intermediate logical representations; these record many of the syntactic features of the English input strings needed to resolve various semantic issues, such as quantifier scope. The resulting underspecified representation must be 'finalised' to construct valid CycL. This 'finalisation' process proceeds via both syntactic and semantic (knowledge-driven) transformation rules. The interface also acquires lexical knowledge from the user as new concepts and facts are entered. More details of the NLP components can be found in [9].

The User Interaction Agenda (UIA) provides tools for the following tasks: creating concepts, predicates and individuals; classifying concepts; identifying and asserting relations applicable to new (and existing) concepts; and formulating rules for the use of these concepts in reasoning.

The UIA tools include the *precision suggestor* for placing a concept appropriately in the hierarchy. This tool identifies a small set of possible generalisations and specialisations of a new concept and suggests these to the user. The user can select from these alternatives - a more effective technique than browsing the entire ontology. In order to facilitate the entry of assertions at the right level of generality, this tool can also be applied to any assertion proposed by the user.

The *salient descriptor* aims to add a minimal, appropriately general set of relevant assertions about new (or existing, but under-ontologized) concepts. This

tool queries the user, in English, for the additional information using general, context-dependent knowledge-acquisition rules. For example, on entering knowledge about a new type of cellular process, the user will be asked to identify the kinds of cells that are involved in this process, and where in these cells the process takes place. Prompting these queries are knowledge-acquisition rules of the form, *if P(a), it is useful to know Q(a)*. Again, the precision suggestor can be used against the output of any salient descriptor interaction, to help ensure that the right level of generality has been achieved.

The *process descriptor* assists the user to enter descriptions of structured event types, or Scripts. A Script is a typical pattern of events that can be expected to re-occur - 'dining in a restaurant' and 'brushing one's teeth' being well known examples. The tool allows the various steps of a process to be defined and ordered, and for the types of actors and roles in the various steps to be identified. The precision suggestor and salient descriptor tools are called upon as necessary. For example, if a process-step is defined as a kind of creation event, the precision suggestor will ask the user whether it might, in fact, be a kind of physical creation event. On confirming this suggestion, the user will be asked by the salient descriptor to identify a class of tangible things that are created in any such event. If the process-step in question has been defined as a process, the process descriptor will use the knowledge gained during the precision suggestor and salient descriptor interactions to propose new candidate roles and actor types. The tools thus interact in concert, with new and existing knowledge determining the applicability of each interaction: The knowledge in the Cyc Knowledge base drives the interactions that create and refine new knowledge. In turn, the knowledge gained during these interactions drives additional knowledge creation and refinement steps.

The relevance of a process ontology becomes all the more evident when one realizes that knowledge creation and refinement is itself a process, describable as a Script in CycL. Thus a rich and inferentially powerful theory of Scripts is potentially useful beyond giving the KRAKEN system the resources to guide the user in defining new processes. With the knowledge that every UIA knowledge-entry session is itself an instantiation of a kind of Script, Cyc will be able to 'follow' the Script, anticipating decision points and user actions, and, overall, more effectively guide knowledge-entry sessions from start to finish, in much the way a genuinely intelligent agent would. Though this use of a process ontology in AI-driven knowledge entry will not be discussed in the scope of this paper, it is a goal worth keeping in mind. It is notable that some portion of RKF year 2 resources have been allocated for AIAI and Cycorp to do serious research in this area. Thus one way to interpret the results reported here is to think of them as important stepping stones towards the larger goal of using a process ontology as a core component of knowledge-entry tools, generally.

3 The Process Theory: An Extension of Scripts

This section presents three extensions to the Script theory: Participants, Conditions and Repetition. The Participants theory extends the existing vocabulary for identifying the objects that play a role in a scripted event or a Scene. The Conditions theory is a new theory for specifying the preconditions and effects of a Scene. The Repeated Scripts theory provides the semantics for the repetition of an Event type, within the Process theory. These theories all provide a type-level vocabulary, and are grounded at the instance-level which provides the semantics. As a consequence of the type-level definitions, the problem of identity arises, i.e. which instance plays a given role in an event, given that only its type is specified. Vocabulary for stating identity properties is also presented. An example of the use of the Process theories is presented in Section 4.

3.1 The Script Theory

The existing approach to participants in Scripts can be summarised as follows. A *Script* is an event-type whose instances have subevents, i.e. *Scripts* are composite events. A number of relations are defined which state that a *Type* plays a *role* in a *Scene* (within a *Script*). These assert that some instance of an *Object-Type* plays a role in a Scene. More accurately, the object-instance plays a role in an event-instance of type Scene, which is a subevent of (an instance of) the Script. The relation *typePlaysRoleInScript* holds of a Script, Type and role, with the interpretation given above.

As Scripts are specified as a conjunction of type-level assertions, some of which assert the existence of *some* object, the problem of identifying those objects across Scenes arises. The object acted on by the process as a whole may be acted on by one or more subprocesses, or the output of one subprocess may be identified with (be the same object as) input to another subprocess. More formally, several models of identity are defined, including: 1) All objects of Object-Type playing a role in the Script are identified with all objects of that type playing any role in any Scene, 2) In Scenes 1 and 2 in a Script, any object playing role-1 in Scene-1 plays role-2 in Scene-2.

The general approach is to associate an instance of an Object-Type with an instance of an Event-Type (by an existential quantification), and to state identity conditions by additional assertions. These are equivalent to rules of the form: *(implies "?OBJECT plays role in event-1" "?OBJECT plays role in event-2")*

3.2 Participants in Processes

Processes are formalised as Scripts. However, the representation of participants is modified for Processes. Firstly, an explicit count of objects of the given type which play a role in the event must be specified. Secondly, it is necessary to know which objects of that type play a role in a specific event: the existing Cyc predicate *actors* is used for this purpose. The resulting models are compatible with the original Script models, and due to the explicit count of participants, the new

formulation contains the information required for process instances to be created. As the process description is really a specification, we cannot immediately derive a ground instance of the process (a model in terms of event-instances and objects) from it, but can validate a ground model against the type level description.

The relations *actorTypeInScriptCount* and *actorTypeInSceneCount* state the number of things of a given type that play any role in a Script or Scene. These relations have the following rules which conclude with the Cyc relation *relationInstanceExistsCount*, specifying the number of *instances* of *?TYPE* for which *(actors ?EVENT instance)* holds. *actors* is used as the most general predicate relating events and instances, it will be specialised during process modelling.

```
F: (implies (and (actorTypeInScriptCount ?TYPE ?SCRIPT ?INT)
                 (instantiatesScript ?EVENT ?SCRIPT))
            (relationInstanceExistsCount
                actors ?EVENT ?TYPE ?INT)).
```

The rule given below shows that it is the *?OBJECT* identified in the *actors* assertion that the *role* holds of, in addition to the *?OBJECT* being of *Type*.

```
F: (implies
      (and (instantiatesScript ?EVENT ?SCRIPT)
           (isa ?OBJECT ?TYPE)
           (actors ?EVENT ?OBJECT)
           (allInstancesOfTypePlaysRoleInScript ?SCRIPT ?TYPE ?ROLE))
      (?ROLE ?EVENT ?OBJECT)).
```

The relation *allInstancesOfTypePlaysRoleInScript* is a specialisation of *typePlaysRoleInScript*. The following KE suggestion rules are defined to encode the knowledge acquisition requirements as they apply to Processes. This type of information is important as it drives the suggestion mechanisms of the GUI tools. For a ScriptedEventType, an *actorTypeInScriptCount* is expected, for which a specific Role is also expected, therefore the following two rules are defined:

```
F: (implies (isa ?SCRIPT ScriptedEventType)
            (keStrongSuggestion ?SCRIPT
              (thereExists ?TYPE (thereExists ?INT
                (actorTypeInScriptCount ?TYPE ?SCRIPT ?INT))))).

F: (implies
    (and (actorTypeInSceneCount ?TYPE ?SCRIPT2 ?SCRIPT1 ?INT)
         (isa ?SCRIPT1 PrimitiveEventType))
    (keStrongSuggestion ?SCRIPT2
      (thereExists ?ROLE
        (and (isa ?ROLE BinaryRolePredicate)
             (typePlaysRoleInScene ?SCRIPT2 ?TYPE ?SCRIPT1 ?ROLE))))).
```

The suggestion for a *typePlaysRoleInScene* assertion may be followed by specialisation to *allInstancesOfTypePlaysRoleInScript* where applicable.

Aggregate Processes When describing processes at an aggregate level, which, in the cell biology domain, is a process at a level above that of a single DNA-RNA transcription episode, the types of the participants will be known but the exact number may be unimportant. In order to be able to describe the numbers of participants in an action where there are an unspecified number, we make use of the the Cyc type *NonNegativeIntegerExtent* in the definition of *actorTypeInScriptCount*. This type can be instantiated by positive integers, ranges of integers, and qualitative values. The latter allowing us to state that *few* or *many* objects of Type participate.

Identity Explicit assertions that the object(s) playing a role in one Scene is(are) the same as those in another Scene, or in the Script as a whole, are required. The problem of identifying instances from subevent to subevent arises from the type-level approach where properties of Scenes are stated in the context of Script, but otherwise in isolation from each other. For example, the following rule states that the objects playing *role-1* in the Script also play *role-2* in subevents of type *Scene*. The rule illustrates the Script/Scene model as the variable *?EVENT* is the instance of the Script, and *?SUBEVENT* is the instance of the Scene.

```
F: (implies (and
        (sameInstancePlaysRoleInScene ?TYPE ?SCRIPT ?SCENE ?ROLE1 ?ROLE2)
        (instantiatesScript ?EVENT ?SCRIPT)
        (subEvents ?EVENT ?SUBEVENT)
        (isa ?SUBEVENT ?SCENE)
        (isa ?OBJ ?TYPE)
        (?ROLE1 ?EVENT ?OBJ))
     (?ROLE2 ?SUBEVENT ?OBJ)).

F: (implies (and
      (sameInstancePlaysRoleInScene ?TYPE ?SCRIPT1 ?ROLE1 ?SCENE1 ?ROLE2)
      (sameInstancePlaysRole ?TYPE ?SCENE1 ?SCENE2 ?ROLE2 ?ROLE3))
    (sameInstancePlaysRoleInScene ?TYPE ?SCRIPT1 ?ROLE1 ?SCENE2 ?ROLE3)).
```

The second rule above shows that the Script-to-Scene identity relation can be inferred, therefore it need not be exhaustively enumerated in the modelling exercise.

In aggregate processes, the objects participating in one sub-process in a script must also be identified with objects participating in another. We introduce **some***InstancePlaysRole* to mean that of the set of things playing *role-1* in *Scene-1* **some** play *role-2* in *Scene-2*. This allows for a process to produce *Many* things of *?Type*, and for some to be consumed by one subsequent process, some by another.

```
F: (implies
    (and (someInstancePlaysRole ?TYPE ?SCENE1 ?ROLE1 ?SCENE2 ?ROLE2)
         (instantiatesScript ?EVENT ?SCRIPT)
         (subEvents ?EVENT ?SUBEVENT1)
         (isa ?SUBEVENT1 ?SCENE1)
         (subEvents ?EVENT ?SUBEVENT2)
         (isa ?SUBEVENT2 ?SCENE2)
         (isa ?OBJ1 ?TYPE)
         (?ROLE1 ?SUBEVENT1 ?OBJ1))
    (thereExists ?OBJ2 (and (isa ?OBJ2 ?TYPE)
                            (?ROLE1 ?SUBEVENT1 ?OBJ2)
                            (?ROLE2 ?SUBEVENT2 ?OBJ2)))).
```

In the general case, a set of instances is associated with a Scene. However, additional facts about identity can be derived when it is known that only one instance of ObjectType plays a role in a Scene. If we have an **instantiation** of a Scene where a particular instance, *Object-1*, plays this role, then in this case the set of instances is just the singleton set containing *Object-1*. These facts can be used in the Conditions theory, which is described below.

3.3 Conditions in Processes

The conditions and effects of scenes are also defined at the type-level. Only Scenes are treated as it is assumed that the conditions of Scripts are derivable from those of the constituent Scenes, and that the knowledge of conditions may also be used to construct new plans/process models from primitive components only.

preconditionOfScene holds of a Scene, a predicate, and a specification term. The semantics at the instance level are expressed in terms of the existing Cyc predicate *preconditionFor-PropSit* which holds of an *ELSentence-Assertible* and an *Event*. The predicate and the specification of *preconditionOfScene* determine the *ELSentence-Assertible*. The specification term selects among the objects that have been defined to play a role in the Scene, that is, all objects that are referred to in the conditions must be declared to play a role in the Scene. This is done using the Participant vocabulary.

Before considering the specification term, it is worthwhile to note that this term necessarily identifies a set of objects that are associated with the Scene as it is a type-level expression. We assume that the precondition of the event is satisfied if the predicate holds of a single member of this set. In the rule below, *?ROLE* and *?SUBEVENT* define the set of objects, of which *?E* is a member. *(?PRED ?E)* holds prior to the Scene.

```
F: (implies (and (preconditionOfScene ?SCENE ?PRED ``?ROLE'')
                 (isa ?ROLE BinaryRolePredicate)
                 (isa ?PRED UnaryPredicate)
                 (isa ?SUBEVENT ?SCENE))
    (thereExists ?E (and (?ROLE ?SUBEVENT ?E)
                         (preconditionFor-PropSit (?PRED ?E) ?SUBEVENT)))).
```

A Scene may have several preconditions. These are stated independently of each other, and have the interpretation that the conjunction of these *preconditionOfScene* assertions must hold for the event to be executable. The conditions may be unary, binary or ternary relations. When there is only one instance of an actor in the Scene, additional facts about the existentially quantified variable $?E$ can be derived - namely that it equates to the actor instance.

Analogously, the predicate *postconditionFor-PropSit* is introduced to represent the instance-level relation between the postcondition (formula) and the event. Postconditions of a Scene are similarly specified by *postconditionOfScene*. These assertions represent conditions (assertions) that hold after the Scene.

In order to construct the specification term, a number of functions are introduced. To show their use by example, the rules below cover the cases where a unary postcondition predicate, *?PRED*, is specified by the set of objects playing *?ROLE*, and where a binary predicate is specified by the collections *?COLL1* and *?COLL2*.

```
F: (implies
    (and (postconditionOfScene ?SCENE ?PRED (TypeArgSpec-UnaryFn ?ROLE))
         (isa ?ROLE BinaryRolePredicate)
         (isa ?PRED UnaryPredicate)
         (isa ?SUBEVENT ?SCENE))
    (thereExists ?E
         (and (?ROLE ?SUBEVENT ?E)
              (postconditionFor-PropSit (?PRED ?E) ?SUBEVENT)))).

F: (implies
    (and (postconditionOfScene ?SCENE ?PRED
                               (TypeArgSpec-BinaryFn ?COLL1 ?COLL2))
         (isa ?COLL1 Collection)
         (genls ?COLL1 SomethingExisting)
         (isa ?COLL2 Collection)
         (genls ?COLL2 SomethingExisting)
         (isa ?SUBEVENT ?SCENE))
    (thereExists ?E (thereExists ?F
       (and (actor ?SUBEVENT ?E)
            (isa ?E ?COLL1)
            (actor ?SUBEVENT ?F)
            (isa ?F ?COLL2)
            (postconditionFor-PropSit (?PRED ?E ?F) ?SUBEVENT))))).
```

All permutations of unary, binary and ternary predicates specified by roles and/or collections in any argument position are permitted. The predicate may also be negated in the pre/postcondition.

Identity The problem of establishing identity between the arguments of the pre/postconditions again arises. The predicate *identityInConditionsOfScene-Arg1Arg1* is introduced. It states that, for a given Scene, arg-1 of predicate $?P1$ is equal to

arg-1 of predicate ?$P2$, where both predicates occur in a pre or a postcondition relation (*?SREL1 ?SREL2*).

```
F: (implies
    (and (identityInConditionsOfScene-Arg1Arg1 ?SCENE ?SREL1 ?P1 ?SREL2 ?P2)
         (isa ?SUBEVENT ?SCENE)
         (isa ?P1 UnaryPredicate)
         (isa ?P2 UnaryPredicate)
         (?SREL1 (?P1 ?OBJECT1) ?SUBEVENT)
         (?SREL2 (?P2 ?OBJECT2) ?SUBEVENT))
      (equals ?OBJECT1 ?OBJECT2)).
```

Similarly, predicates specifying identity for arg1-arg2, arg1-arg3, arg2-arg2, arg2-arg3 and arg3-arg3 are defined. These relations provide the means to express at the type level the information that would be more usually encoded by bindings between variables. In the example below, a Strips-like action description for *(connect ?X ?Y)* is followed by the equivalent argument identity assertions for the *Connect* Scene:

```
Strips-like description:
Action:         (connect ?X ?Y)
Preconditions:  (near ?X ?Y)
Postconditions: (connectedTo ?X ?Y)

Type-level description of identity:
(identityInConditionsOfScene-Arg1Arg1 Connect
             preconditionFor-PropSit near
             postconditionFor-PropSit connectedTo)

(identityInConditionsOfScene-Arg2Arg2 Connect
             preconditionFor-PropSit near
             postconditionFor-PropSit connectedTo)
```

The identity conditions simply state that the ?X and ?Y in *near* in the preconditions must be the same ?X and ?Y in *connectedTo* in the postconditions.

Planning We have implemented translation procedures that transform the type-level encodings of actions into PDDL and also construct a constraint theory in Cyc [2]. The semantics of the constraint theory are equivalent to those of the PDDL problem definition. Thus we have a dual representation of conditions: a Process semantics which is consistent with Process models, and a constraint semantics which is consistent with the PDDL action encoding. This allows us to call an external planner to perform plan generation when this type of reasoning is required. It is worth noting that the ability to plug in an external planner, though desirable, is not necessary. Cycorp has developed a hierarchical planner within Cyc, currently deployed as a part of their Cyc Secure (TM) product.

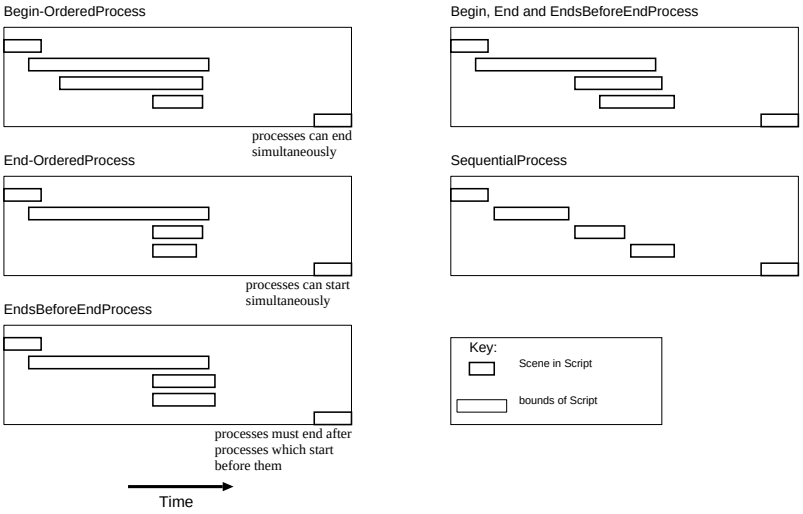

Fig. 1. Ordering constraints on Repeated Processes

3.4 Repetition in Processes

Biological models commonly include processes that are repeated. The number of repetitions may be known, or may be unspecified. Further, repetition may occur until a specific condition is achieved. Repeated actions are a common feature in descriptions of activity, for example, in Computer Science, many formalisations of procedural languages have been proposed to describe the *while* loop construct. The Repeated Process theory encapsulates the important features of activity models which include repeated activities, formulated in terms of Scripts.

Repeated events are modelled as ScriptedEventTypes, whose instances expand into a set of instances of the event-type being repeated. A number of properties of the expanded set of instances are identified. These include the ordering of the repeated instances, e.g. whether instances overlap, the number of repeated instances, and whether the repeated activity terminates. The Repeated Script extension is now described in detail. However, we do not give the CycL definitions of these concepts in this paper.

Properties of Repeated Processes A number of properties on the ordering and constitution of repeated ScriptedEventTypes are defined by the following collections. These properties capture important and distinguishing aspects of repeated events. In general, ScriptedEventTypes do not have the restricted properties defined below for repeated events.

Begin-OrderedProcess Subevent instances of a Begin-OrderedProcess start at distinct time points. As time points are totally ordered, the start times of subevent instances are also totally ordered. The constraints on subevent order are shown diagrammatically in Figure 1.

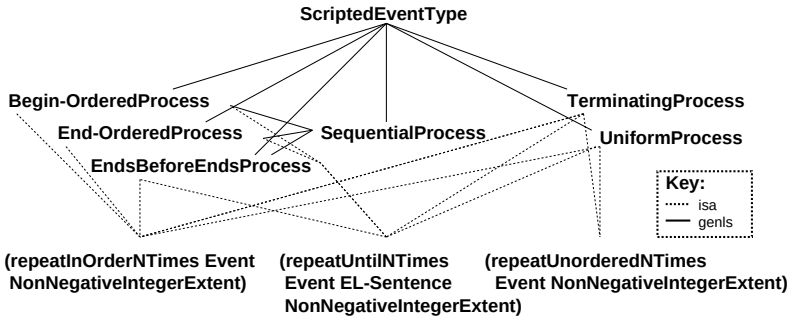

Fig. 2. Properties of Repeated Processes

End-OrderedProcess Subevent instances of a End-OrderedProcess end at distinct time points.
EndsBeforeEndProcess Subevent instances of an EndsBeforeEndProcess end before or at the same time as subevent instances which start before them.
SequentialProcess A SequentialProcess has no overlapping subevent instances.
TerminatingProcess An instance of a TerminatingProcess has a *lastSubEvents*. I.e. there is a subevent instance after which no other subevent instance begins, and as all activity instances have a begin and end point, there is a time point at which the composite process ends.
UniformProcess All subevent instances of a UniformProcess are instances of the same event-type, *subScriptedEventTypes* must be the same.

Figure 1 illustrates the three constraints Begin, End and EndsBeforeEnd, and also their combination. This combination plus the sequential specification are used to define the structure of repeated processes. The condition of uniformity requires all subevent instances to be of the same type, and termination ensures there is a last subevent instance.

Definitions of Repeated Processes Repeated Processes can now be defined in terms of the properties introduced above. The approach taken is to introduce functions which take event-types as arguments, and denote a new ScriptedEventType which is the repetition of the argument Event. The repeated Script is modelled as a new composite Script which expands into a uniform set of subevents. For example, the *RepeatInOrder* function is used to create a new composite Script, which itself consists of an overlapping sequence of events at the instance-level. The instance level is specified intensionally, it is not intended that the formalisation explicitly create large numbers of instances of repeated activities. Three types of repetition are identified: unordered, ordered, and repeat until a postcondition is achieved. The number of repetitions may be specified. The properties of *(Repeat*)* functions are given in Figure 2.

RepeatInOrder Each activity instance of *(RepeatInOrder ?EVENT ?INT)* has a positive number of subevent instances which are ordered. The number of

subevents may be specified by a positive integer, a range, or a qualitative value (i.e. a *NonNegativeIntegerExtent*). The subevent instances are Begin and End-Ordered, EndsBeforeEnd, Terminating and Uniform.
RepeatUnordered Each activity instance of *(RepeatUnordered ?EVENT ?INT)* has a positive number of subevent instances and the only ordering requirement is the existence of a last subevent. The event instances are Terminating and Uniform. The specification in terms of the number of event-instances is similar to that given above.
RepeatUntil A Script is repeated until the ELSentence-Assertible is achieved, i.e. it becomes the *postconditionFor-PropSit* of the last subevent. Each activity instance of *(repeatUntil ?EVENT ?PROP ?INT)* has a positive number of subevent instances which are ordered but may overlap. The subevent instances are Begin, End, EndsBeforeEnd ordered, Terminating and Uniform. Again, the number of repetitions is specified by a positive integer or a range.

An additional relation, *(sceneHasDefinition ?SCENE ?SCRIPT)*, is defined to allow an event-type to be modelled as a Scene in one process model, but to have an expansion (a Script) in another context. This relation allows the 'modularisation' of process models within the knowledge base.

Inferred Constituent and Identity Relations As repeated Scripts have a uniform composition, and are composed of event-types that have already been defined, their constituents, type-plays-role and identity properties can be derived from existing assertions. This is possible where the repeated event it itself a ScriptedEventType.

Two identity models are allowed: 1. *inheritIdentityToSubscenes Event-Type* holds, or 2. *inheritIdentityToSubscenes Event-Type* does not hold. In case 1, the same things play a role in the Script and in all Scenes. Additionally, the number of things playing a role in the Script is the same as the number playing a role in the Scenes. An example is the repeated elongation of RNA in RNA-transcription, where the **same** RNA molecule is modified in each step. In case 2, the things playing a role in each of the repeated events differ (each subevent has its own instances of the Object-Type). An example is the generation of many RNA molecules in the virus life cycle model: each instance of RNA-transcription outputs a **different** RNA molecule.

4 An Example Process: The Virus Life Cycle

The vaccinia virus life cycle is an example which makes use of all aspects of the Process ontology. The model describes the attachment of a virus to a cell, the movement of the core of the virus into the cell, and the RNA transcription and translation processes that occur as the virus replicates itself [10].

The temporal ordering of processes in the virus life cycle model was found to require *startsAfterStart*, *startsAfterEnd* and *endsAfterEnd* orderings. These are present in the Script vocabulary. There are two groups of subactivities of the

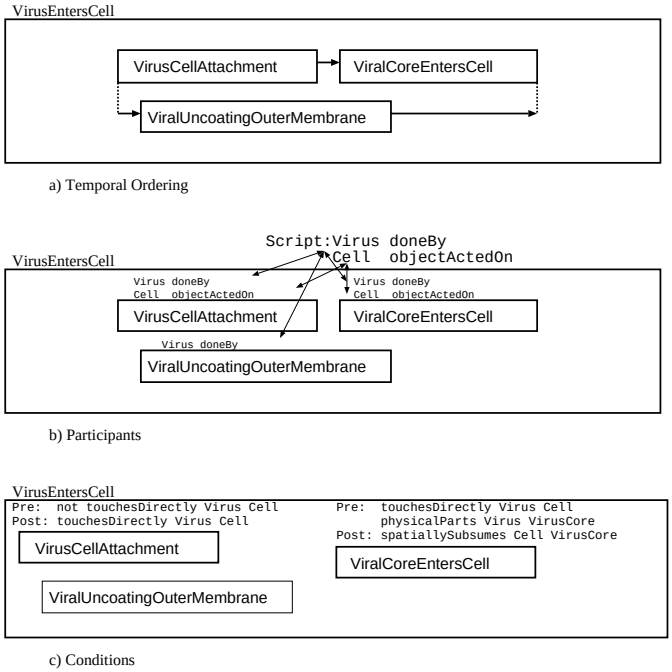

Fig. 3. The VirusEntersCell process

virus life cycle that can be distinguished (*VirusEntersCell* and *ViralTranscription-Late*), the remaining early and intermediate transcription activities are too inter-related to be able to introduce any other intermediate-level groupings. Figure 3 a) shows the subactivities of the VirusEntersCell process and their temporal ordering. The entire model was formalised in 144 type-level assertions relating to 14 activity types, but only a small fragment can be presented here.

The objects involved in each activity and the roles played are shown in Figure 3 b). The strong identity model is applies here: the same Cell and Virus are referred to in the Script and in all Scenes. The quantifier *many* is frequently used in the life cycle model and, consequently, **some***InstancePlaysRoleInScene* is also required to describe the 'flow of objects'. The temporal order can, in part, be deduced from the *input* and *output* roles[3].

The source document [10] describes the preconditions and effects of several actions, and these are represented using the precondition relations, and existing Cyc terms where possible, see Figure 3 c). The preconditions mainly refer to spatial location. Some overlap between the concepts of precondition and role were found in the text - the existence of something playing an input role was often described as a precondition. We take the approach of not duplicating the *ObjectType-plays-Role* information in the conditions of an action.

[3] However, Figure 3 does not illustrate this.

A more detailed model of RNA transcription has also been specified in a separate modelling exercise. This model describes the production of a single object, a molecule of RNA. By using the repetition operators, this model can be related to the MRNATranscription-Early process in the virus model:

```
F: (sceneHasDefinition MRNATranscription-Early
                       (RepeatUnordered DNARNATranscription Many-Quant)).
F: (unknownSentence (inheritIdentityToSubscenes
          (repeatUnordered DNARNATranscription Many-Quant) DNAMolecule)).
F: (unknownSentence (inheritIdentityToSubscenes
          (repeatUnordered DNARNATranscription Many-Quant) RNAPolymerase)).
F: (unknownSentence (inheritIdentityToSubscenes
          (repeatUnordered DNARNATranscription Many-Quant) MessengerRNA)).
```

In this case, the repetition of *DNARNATranscription* does not have the *inheritIdentityToSubscenes* property for any of its object-types. Therefore, many *MessengerRNA* are involved in the repeated process as *outputs*, each repetition producing one molecule (by the RNA transcription model). New properties of the *MRNATranscription-Early* process can be derived as a result, and it becomes possible to explore the structure and content of the transcription model.

The three perspectives on the process: temporal order, participants, and conditions are useful views to distinguish in process modelling. The relation of the perspectives through the types of participants that are involved in each process is the feature which unifies the views.

5 Related Work

The Process ontology is closely related to the Process Specification Language, in terms of both the intended area of application and formal approach. PSL identifies four classes which form a partition: Activity, ActivityOccurrence, Object and Timepoint. An ActivityOccurrence has a type (*occurenceOf*), can be related to a timepoint, and can have an object associated with it. The PSL Core is an instance-level theory of activity which can be mapped to the equivalent event/object relations in Cyc (indeed we have specified four PSL theories in CycL). While the predicates differ, equivalent instance-level models of events can be created.

PSL has several theories which the Process ontology currently lacks. However, PSL does not provide a well defined set of type-level relations for subevent ordering, participants or conditions (fluents in PSL)[4]. Modelling using the core PSL theories must be performed primarily at the event instance level. PSL does define a theory of junctions in processes, i.e. it is possible to define or-splits and and-splits. This theory is expressed in the way we advocate: a property of an activity-type is defined in terms of constraints at the activity occurrence level. Therefore, we can import junctions theory, after re-expressing it in Cyc terms.

[4] Many of the type-level relations have no axioms and only a textual definition.

DAML-S [4] contains a process ontology which contains many concepts found in PSL, and in the ontology presented here. For example, types of process include atomic, simple and composite, process parameters (properties) include inputs, outputs and participants and these correspond to roles in our ontology. DAML-S also has preconditions and effects, sequence, split+join and repetition. However, DAML-S currently only defines the names of collections or properties so no detailed comparison is possible.

IDEF3[5] is a process modelling methodology which primarily diagrammatic. Processes, process products, and their connections are represented by a convention of boxes and arrows. The visual presentation is important in the modelling process, and in explaining the model to the managers and employees in an organisation. IDEF3 models have several features which inform the interpretation of the diagrams, including and/or/xor junctions. Processes may have attributes such as triggers which are not shown in the diagram but are documented elsewhere. None of these features have formal semantics. The informality allows a single model to describe many complex phenomena, such as repetition, splits in the flow of processes (junctions) and synchronisation, in a relatively intuitive way. Naturally, IDEF3 models cannot be processed by machine without creating an interpretation (explicitly or implicitly). Defining an underlying formal semantics for informal modelling techniques allows the consistency and integrity of the models to be maintained [3]. These practical benefits are important justifications for formalisation in a business context.

The dominance of visual methods in process modelling is notable. While the current Cyc GUI requires the user to fill-out a form with the names of event-types and other classes, the same information could be obtained while the user labels a process in a drag-and-drop panel. As is the case in the form-based interface, the user need not be aware of the underlying type-level relations that are being asserted: We consider the three views of a process illustrated in Figure 3 to be the essence of such an interface.

6 Conclusions

Through pursuing the type-level approach, we have found there to be a relatively small set of useful abstractions of processes. Consequently, the combination of a type-level vocabulary with the associated knowledge-elicitation rules is a powerful technique for knowledge acquisition. An advantage of the Process vocabulary is the ability to specify processes in an incremental fashion, as properties of Scripts and Scenes are broken down to manageable fragments. The alternative is to author complex rules at the event-instance level, a task which, when given to domain experts, certainly requires them to become much more familiar with the logical encoding - in addition to the semantics of the logical terms.

Supporting the user by providing a visualisation of the formalised process will assist the modelling task. This may be form-based or use direct manipulation. The tool developers task of providing the interface is simplified by the type-level

characterisation of the event models, as diagrammatic views of arbitrary logical formula need not be created.

In conclusion, we believe type-level abstractions of processes to be valuable in terms of the semantics they provide, and in terms of supporting the knowledge authoring task directly, and through easing tool design.

Acknowledgements

This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) under subcontract 00-C-0160-01 with Cycorp on BAA99-35. The U.S. Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of DARPA, Rome Laboratory or the U.S. Government.

References

1. Alberts, B., Bray, D., Johnson, A., Lewis, J., Raff, M., Roberts, K., and Walter, P. *Essential Cell Biology* Garland Publishing, London, 1998.
2. Aitken S. Process Planning in Cyc: From Scripts and Scenes to Constraints. *Proc. Twentieth Workshop of the UK Planning SIG, PLANSIG 2001*, Ed. Levine, J., December 2001, pp. 257-260.
3. Chen-Burger, Y. and Robertson, D. Formal Support for an Informal Business Modeling Method. *Proc. SEKE 1998.*
4. DAML Web Service Ontology `http://www.daml.org/services/`
5. *IDEF3 Method Report*, KBSI Inc. `http://www.idef.com/idef3.html`
6. Lenat, D.B. and Guha, R.V. *Building large knowledge-based systems. Representation and inference in the Cyc project.* Addison-Wesley, Reading, Massachusetts, 1990.
7. Lenat, D.B. *Leveraging Cyc for HPKB Intermediate-level Knowledge and Efficient Reasoning* `http://www.cyc.com/hpkb/proposal-summary-hpkb.html`
8. Rapid Knowledge Formation Project `http://reliant.teknowledge.com/RKF/`
9. Panton, K., Miraglia, P., Salay, N., Kahlert, R.C., Baxter, D., and Reagan, R. Knowledge Formation and Dialogue using the KRAKEN Toolset. *Proc. IAAI-02 in press.*
10. *RKF EKCP Specification Molecular Cell Biology Spatio-temporal Dynamics for Advanced Genome Annotation and Exploitation.* Version 2.01, IET Inc., Rosslyn, Virginia, 2001.
11. Schlenoff, C., Gruninger, M., Tissot, F., Valois, J., Lubell, J., and Lee, J., *The Process Specification Language (PSL) Overview and Version 1.0 Specification.* NIST Report (NISTIR) 6459, Jan. 1999.