The Representation of Actions in KM and Cyc

Aarati Parmar
Department of Computer Science,
Gates Building, 2A Wing
Stanford University, Stanford, CA 94305-9020, USA
aarati@cs.stanford.edu

Technical Report FRG-1
May 7, 2001

Abstract

In this article we explore the representations of actions in the systems Cyc and KM, in sections 1 and 2. In Section 3, we compare a selected set of actions in KM against those in Cyc. For simplicity, KM terms are set in `this typeface`, while Cyc ones are in this one. Section 4 gives a summary of the differences between the two, along with a table comparing selected actions. The essential difference is that while Cyc can teach us much about actions and properties of them, KM can actually *simulate* these actions. KM can support the rich ontology Cyc does; there is only the matter of coding the facts.

Contents

1 Representation of Actions in Cyc **2**
1.1 Action's Place in the Cyc Ontology 3
1.2 Properties of Actions in Cyc 5
1.3 Representing Preconditions 5
1.4 Representing Results of Actions 6
1.5 Cyc Summary 7

2 Representation of Actions in KM **8**
2.1 Situations in KM 8
2.2 Actions in KM 8
2.3 Simulating Actions in KM 9
2.4 KM Summary 11

3 Comparison of Action Representations 11
3.1 Events 12
3.2 Actions 12
3.3 `Break` 12
3.4 `Break-Contact` 12
3.5 `Create` 13
3.6 `Make-Accessible` 13
  3.6.1 `Release` 13
3.7 `Make-Contact` 13
3.8 `Make-Inaccessible` 13
  3.8.1 `Confine` 14
  3.8.2 `Be-Confined` 14
3.9 `Move` 14
  3.9.1 `Carry` 15
  3.9.2 `Enter` 15
  3.9.3 `Move-Out-Of` 15
3.10 `Remove` 15
3.11 `Repair` 16
3.12 `Transfer` 16
3.13 `Encode` 16
3.14 `Read` 16

4 Conclusions and Discussion 17

5 Index 20
5.1 Cyc Collections 21
5.2 Cyc Predicates 22
5.3 `KM` Terms 22
5.4 `KM` Slots 23

1 Representation of Actions in Cyc

This study of the actions in Cyc is based on version 12, patch 1.652 of the IKB.

Cyc makes no commitment to any formalism for representing change, such as situation calculus [McCarthy and Hayes, 1969], event calculus [Shanahan, 1999], or STRIPS [Fikes and Nilsson, 1971]. Actions are defined by their location in the Action hierarchy and what axioms and slots (predicates) apply to them.

For example, Cracking is a subclass of the actions classes SeparationEvent, IntrinsicStateChangeEvent, and PhysicalEvent. Through the type hierarchy, any instance of Cracking inherits any axioms applicable to these classes, along with those that asserted about the class itself. Some axioms include:

```
(implies (and (isa ?CRK Cracking)
              (objectOfStateChange ?CRK ?OBJ))
         (holdsIn ?CRK (isa ?OBJ SolidTangibleThing)))
                                                   [From Cracking]

(requiredArg1Pred SeparationEvent outputsRemaining)
                                                   [Inherited from
                                                  SeparationEvent]

(requiredArg1Pred IntrinsicStateChangeEvent objectOfStateChange)
                                                   [Inherited from
                                         IntrinsicStateChangeEvent]
```

The first axiom asserts that the object involved in a Cracking event is a SolidTangibleThing during the event. The second abbreviates the fact that for any instance SeparationEvent1 of SeparationEvent, there is an object SomethingExisting2 such that (outputsRemaining SeparationEvent1 SomethingExisting2).[1] Since Cracking is a subclass of SeparationEvent, this property also applies to it. The third axiom asserts that every instance of IntrinsicStateChangeEvent has a PartiallyTangible object which is its objectOfStateChange. Note that the object that is cracked in this case is not even specifically mentioned; only its existence is assured by the third axiom, and asserted to exist after the Cracking by the second axiom.

The rest of section 1 describes where Action fits in the Cyc ontology, properties of Action, and how preconditions and results of Actions are represented in Cyc. It ends with a short summary.

1.1 Action's Place in the Cyc Ontology

The class Action is a subclass of Event. Actions are Events that *must* have doers.[2] Special kinds (subsets) of Actions include AnimalActivity, Information-

---

[1] requiredArg1Pred is a special second-order predicate used to assert the existence of a set of values for an instance's slot. Formally (requiredArg1Pred Collection slotpredicate) means that for any instance i of Collection, there are terms t1, ..., tn such that (slotpredicate i t1 ... tn). slotpredicate has arity $n+1$ and the terms obey its type restrictions.

[2] This is expressed by the GAF (requiredArg1Pred Action doneBy).

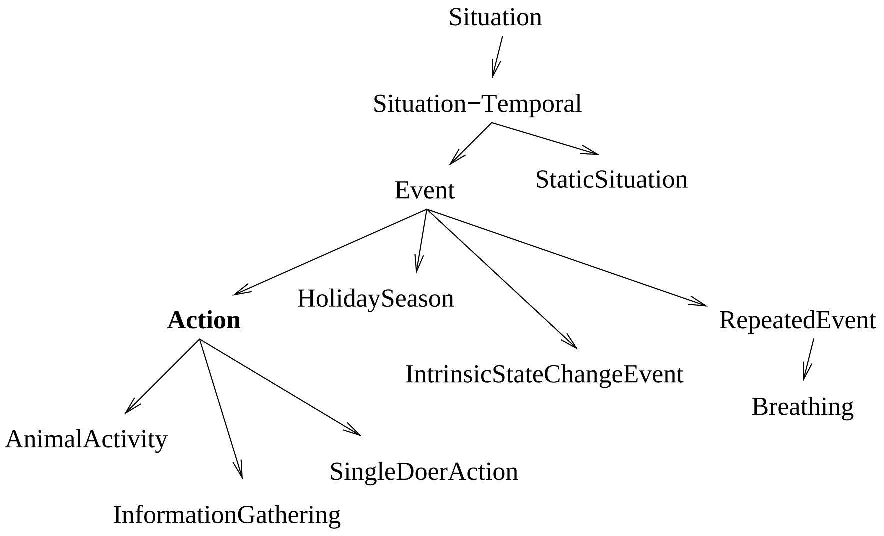

Figure 1: Action's Place in the Cyc Ontology.

Gathering, or SingleDoerAction. Note that these are not mutually exclusive categories.

More general information regarding Action is found in its superclass, Event. Other than Action, Event also includes subclasses:

1. HolidaySeason: events surrounding a particular holiday, such as Christmas,

2. IntrinsicStateChangeEvent: events where one of the participants in an event experiences an intrinsic change, and

3. RepeatedEvent: the class of events that are repeated in another event, such as Breathing.

At first it may seem unintuitive, but Event is a subclass of Situation. Situations are very generally described as states of the world. Situation-Temporal are the Situations with a temporal aspect (might depend on time). Event and StaticSituation are subclasses of Situation-Temporal. Any Event is a change in the state of the world. A StaticSituation on the other hand, is meant to be a time interval during which all relationships stay static. StaticSituations most closely fit our usual notion of situation.

1.2 Properties of Actions in Cyc

Properties of objects in Cyc are asserted through axioms or slots, and inherited through type hierarchies. Cyc has a rich ontology of slots for Events. The most general kind of slot is a Role. A Role is a relation between a Situation and an object. Role's subclass, ActorSlot is the collection of binary predicates which relate Events to the objects that are involved in them. Examples of ActorSlots include[3] bodilyDoer, damages, postActors and products. Other instances of ActorSlots include actors, doneby, prospectiveSeller, and inputs.

Some kinds of Actions are required to have values for their slots. On the other hand, certain slots only can apply to certain kinds of Action. The first constraint is accomplished through the second-order relation requiredArg1Pred. (requiredArg1Pred Buying buyer) asserts that every instance Buying1 of the action Buying must have an agent Agent2 such that (buyer Buying1 Agent2) holds. (requiredArg1Pred ActionOnObject objectActedOn) means that every instance of ActionOnObject has an associated object that is acted upon. Other relations in Cyc, such as relationAllExists, make similar assertions.

Restricting the application of some ActorSlots to certain types of events is straightforward as well. This is done by restricting the domain of the predicate to that event subclass. For example, inputs only applies to instances of CreationorDestructionEvent. prospectiveSeller only applies to actions of type CommercialActivity.

There are other predicates, besides instances of ActorSlots, which describe an action. Some include performanceLevel or skillRequired, which, using the instances of ScriptPerformanceAttribute, can talk about *how* an action is performed. Some of these ways include Agility, Dexterity, and Competence.

1.3 Representing Preconditions

Cyc has numerous means, using instances of ActorSlots, to describe an action. One of most important properties of an action, though, is its preconditions – what are the requirements for a successful action? Cyc has a rich array of predicates that could be used to represent this concept, but they are neither used systematically nor extensively. Most preconditions are specific notions such as resources required, or agents that play a certain role. In practice these requirements are expressed in terms of ActorSlots.

---

[3] Recall that all predicates in Cyc begin in lowercase, to be distinguished from terms, which are capitalized.

Four predicates that embody the traditional notion of preconditions are preconditionFor-PropSit, preconditionFor-Events, preconditionFor-Props, and preconditionFor-SitProp. However, none are used extensively to describe actions in the current version (KB 12, Patch 1.652) of Cyc. (preconditionFor-PropSit CycFormula1 Situation2) asserts that CycFormula1 is necessary in order for Situation2 to be possible. (preconditionFor-Events Event1 Event2) asserts that Event1 is a necessary condition for Event2. (preconditionFor-Props CycFormula1 CycFormula2) asserts that CycFormula1 is necessary for CycFormula2 to hold. (preconditionFor-SitProp Situation1 CycFormula2) is used to say that Situation1 must happen in order for CycFormula2 to hold.

Another predicate that could be used to represent all sorts of preconditions is (requiresForRole Situation1 Collection2 Role3), where the success of Situation1 requires there to be an object in Collection2 which plays the role Role3. For example, the Cyc formula

```
(implies (isa ?U WritingByHand)
         (requiresForRole ?U WritingImplement deviceUsed))
```

states that the WritingByHand action ?U requires an instance ?V of WritingImplement where (deviceUsed ?U ?V). In practice, this is also not used extensively in Cyc.

The predicate (preSituation Event1 StaticSituation2) is used to relate a situation StaticSituation2 that is true right before the occurrence of Event1. This pre-situation is used to assert salience of Event1 to StaticSituation2, a weaker type of precondition.

As mentioned above, ActorSlots themselves embody many specific preconditions. The ActorSlot inputs is used to state which objects are used (and changed) in CreationorDestructionEvent events. inputs has more specific predicates, including inputsCommitted and inputsDestroyed. instrument-Generic is a slot for the general category of objects that are used to facilitate an action.

1.4 Representing Results of Actions

The predicate (eventOutcomes Event1 Situation-Temporal2) asserts that Situation-Temporal is the result of Event. More specific subpredicates include postSituation, causes-EventEvent, postEvents, and inReactionTo. (postSituation Event1 StaticSituation2) is the closest notion of StaticSituation2 being the *result* of Event1.

(causes-EventEvent Event1 Event2) codifies causation between events Event1 and Event2.[4] Some other related predicates include causes-PropProp

---

[4] (causedBy Event1 Event2) appears to be an archaic converse relation.

and causes-SitProp, both subpredicates of causes-ThingProp, the most general notion of causation. causes-PropProp asserts that one Cyc formula causes another, a notion stronger than implication. causes-SitProp is a little more useful as it asserts that a situation causes a formula to become true.

There are looser notions of causation between collections of situations. (causes-SitSitType Situation-TemporalInstance1 Situation2) means that Situation-TemporalInstance1 causes an instance of Situation2, to occur. (causes-SitTypeSitType Situation-Temporal1 Situation-Temporal2) says that an instance of Situation-Temporal1 will cause an instance of Situation-Temporal2 to come about.

(postEvents Event1 Event2) orders the events Event1 and Event2, implying some sort of relevance between the two. Cyc also has functions (STIB TemporalThing1) and (STIF TemporalThing1) to return some TimeInterval shortly before/following TemporalThing1.

(inReactionTo Action1 Situation-Temporal2) is meant to be the weakest form of response, where Action1 is performed in response to Situation-Temporal2. This relation is useful in event narration.

Some action-specific predicates include resultantMentalObjects, used to relate how an agent feels after experiencing a mental object. fromState and toState are predicates used in conjunction with the Action ChangingDeviceState to talk about preceding and post states.

1.5 Cyc Summary

Cyc does not follow the traditional route of action representation by elaborating the sets of fluents which change. Rather, actions are organized into hierarchies, which themselves contain (and inherit) relevant axioms and ActorSlots which [must] apply to certain types of actions. Thus in an instance of the event DryingSomething, there is an instance of a LiquidTangibleThing that permeates the object to be dried beforehand, and does not permeate it afterward. Being a subclass of IntrinsicStateChangeEvent, the drying action *must* have an object that is acted upon.

Properties are ascribed to actions by means of a rich set of ActorSlots, whose instances are predicates relating Events to objects. These are unlimited in expressivity. There are no formal notions of precondition and result, although there are relations that can express what is required. The bulk of preconditions and result appear to be formalized in terms of ActorSlots.

2 Representation of Actions in KM

`KM` [Clark and Porter, 1998] is a frame-based representation language with first-order logic semantics. It has built-in support for reasoning about change. Most of the information below was gleaned from [Clark and Porter, 2000], and the Component Library (v1.0) at `http://www.cs.utexas.edu/users/mfkb/RKF/tree/`. The version of `KM` evaluated is 1.4.3.10.

2.1 Situations in `KM`

In order to talk about actions in `KM`, situations should be described first. As usual, a `Situation` describes the state of the world at a moment in time.

`KM` treats situations much as contexts [McCarthy and Buvač, 1994], where facts (or technically, fluents) are contained within a situation. Situations can be organized hierarchically, having a super- and sub- situations. One can "pop" in and out of situations just as one would with contexts. Facts within a situation are viewable from its subsituations, but not from its supersituations. `KM` has one global situation `*Global`, which is the supersituation of all situations. `*Global` intuitively is the situation containing all timeless truths. Thus, one must be careful what to assert in the global situation (such as information that may be time variant), because it is accessible to every other situation. Situations can be viewed, and quantified over as objects.

Situations require *fluents*. `KM` defines fluents (`*Fluent`) as slots whose values depend on the situation. An *inertial fluent* (`*Inertial-Fluent`) is a fluent whose value persists from one situation to the next.[5] A *non-fluent* (`*Non-Fluent`) is a slot whose value is not situation dependent. Every slot is described by one of these three possibilities.

2.2 Actions in `KM`

`Actions` in `KM` are point-like, in that only the situations before and after the action are modeled.[6] As in Cyc, an `Action` is also a subclass of `Event`.

`KM` uses slots to describe aspects of actions. It implements the STRIPS representation of actions by means of the following four slots:

---

[5] `*Fluent` slots and `*Inertial-Fluent` slots are disjoint – a slot that is a `*Fluent` can vary between situations, but will *not* persist.

[6] `KM` can extend to actions with duration, where the situation during the action is also represented. This is discussed in the summary section 2.4.

1. `pcs-list` (positive preconditions): a list of ground literals, also known as *propositions*[7] which must be true for the action to occur.
2. `ncs-list` (negative preconditions): a list of propositions which must be false.
3. `add-list`: the propositions that become true after the action.
4. `del-list`: the propositions which become false after the action.

Other than the above slots, actions in `KM` have other property-defining slots, such as `object` and `instrument`. As `KM` is frame-based, any action class will inherit slots (and their values) from its superclasses. Hence the action class `Break` inherits from `Action` the preconditions (`pcs-list`) that the object acted upon is accessible.

While an action's operations in `KM` are governed by STRIPS lists, which are conjunctions of literals, `KM` is more expressive than STRIPS because the *generation* of these lists can involve quantification and if-statements. For example, the `pcs-list` for `Move` includes the formula:

```
(if (has-value (the source of Self))
             then
             (forall (the object of Self)
               (:triple
                 It
                 location
                 (the source of Self)))),
```

which states that if the action in question has a value for its `source` slot, then every object moved in the action must be in the same location as this source. When evaluated, this formula will reduce to a set of literals as required by STRIPS semantics. Note how this treatment increases expressivity, not only with the use of quantification and if-statements, but through the use of the predicate `has-value`, which actually checks to see if a slot has a value.

2.3 Simulating Actions in `KM`

Unlike Cyc, `KM` can simulate the application of an action in a situation. It temporally projects facts to create the resulting situation, and also computes

---

[7] *Propositions* are of the form `(frame slot value)`, which asserts that the `slot` of `frame` has value `value`.

ramifications of actions. `KM`'s simulation is non-monotonic in the way it applies preconditions, and performs temporal projection.

The situation resulting from an action is related by the slot `(next-situation s s' a)`, where $s' = result(a, s)$. `KM` also provides some equivalences:

$$\begin{array}{lcl} \texttt{(next-situation s s' a)} & \Longleftrightarrow & \texttt{(prev-situation s' s a)} \\ & \Longleftrightarrow & \texttt{(before-situation a s s')} \\ & \Longleftrightarrow & \texttt{(after-situation a s' s)} \end{array}$$

These different ternary relations equivalent to `next-situation` facilitate syntactic indexing operations. Just as in traditional situation calculus, alternative situations, resulting from different actions in the same situation, are representable. Thus `KM` can represent alternate histories.

The command `(do act1)` simulates the action `act1` in the current situation. It works by first "asserting" that the positive preconditions (`pcs-list`) hold in the current situation. By "asserting" we mean that the preconditions are checked in the current situation. If they do not hold, but it is consistent to assume that they are true, then they are actually set as true in that situation. (This is the first use of non-monotonicity in `KM`.) `KM` also similarly "asserts" that the negative preconditions in `ncs-list` do not hold. If both sets of preconditions are assertable, then a new `next-situation` is created where the propositions of the `add-list` hold, and where those of `del-list` are not allowed to hold.[8] If the preconditions were not assertable, the `nil` action is applied, and the next situation generated is based on that.[9]

`KM` has a built-in mechanism for solving the frame problem (the problem of succinctly expressing and carrying over all facts that do not change). When temporally projecting an action from `s` to `s'`, `KM` asserts the propositions in `add-list` and `del-list` as explained above. Then, all other propositions in slots that are `*Inertial-Fluents`, and hold in `s`, are carried over to `s'`, *as long as they are consistent with* `s'`. Non-inertial slot fluents are not carried over, but are recomputed according to any relevant axioms in `s'`. This is the second use of non-monotonic reasoning by `KM`.

One can see that the inertial/non-inertial distinction ensures that ram-

---

[8] In fact, a constraint is enacted on the relevant slot, specifically disallowing that situation from taking on the `del-list` value. Hence neither direct effects nor ramifications can assert a value that is supposed to have been deleted.

[9] `is-possible` can be used to test, rather than assert, the preconditions of an action in a situation. This is useful for tasks such as planning.

ifications are treated properly, to a first order.[10] Any ramification must be in a slot that is non-inertial, since its value is derived from other fluents that could change, rather than inertia. This solution is adequate, since it deals with what are essentially conjunctions of literals.[11]

2.4 `KM` Summary

`KM` follows the traditional situation calculus style of representing change, with an action linking one situation to a resulting situation. By explicitly representing situations, `KM` has the power to inspect action histories.

Actions in `KM` have slots for STRIPS-like lists, represented as statements which evaluate to lists of conjunctions of propositions. `KM` distinguishes between inertial and non-inertial fluents, and uses this information to perform temporal projection. `KM` is non-monotonic in the way it asserts action preconditions, and performs this temporal projection. Actions in `KM` have other slots as well, to add as much description as necessary. All of these slot values are subject to inheritance.

[Clark and Porter, 2000] demonstrates how `KM` could treat actions as ongoing processes rather than point events. A new type of situation, a "during-situation," is added, which represents the period of time the action is taking place. Ongoing fluents could be tagged as occurring in this situation. This "during-situation" could be broken down into subsituations. This has not been implemented but seems straightforward.

3 Comparison of Action Representations

In this section, we compare how specific actions are formalized in `KM` and Cyc. We first compare the general categories, Event, and Action, and then move on to selected actions.

For the selected actions, we compare the representation of selected actions from `KM` against the closest equivalent one in Cyc. $[t]$ is the time it took the author to find the closest Cyc expression. A $*$ indicates the number is a lower bound. The Wordnet facility in Cyc was not used in the search.

---

[10] This treatment depends on there being exactly two disjoint classes of fluents, those that are exactly the direct effects of actions, and those whose truth is always derived from these direct effects. Furthermore, the partition between these two classes of fluent is permanent. [Thielscher, 1996] demonstrates that this approach is not always viable. In short, it is possible to have a fluent which fits in both categories.

[11] [Clark and Porter, 2000] correctly notes that `KM` does not properly handle disjunctive ramifications, so disjunctive effects are specifically disallowed.

3.1 Events

Both Cyc and `KM` treat Events as objects with a temporal aspect. In `KM` events have (non-required) slots: **`subevents`**, **`time`**, **`agent`**, **`beneficiary`**, **`donor`**, **`instrument`**, **`object`**, **`recipient`**, and **`result`**. `KM` requires that all of these slot with values must be cotemporal and cospatial. The agent in question must be able to perform the event, and any instruments used in the event cannot be broken.

Cyc requires Events to have an associated actor and subevent. The slot actors contains most of the distinctions `KM` makes for its slots, along with many others. These slots include: socialParticipants, intendedBeneficiary, instrument-Generic, target, and outputs.

3.2 Actions

In `KM`, **`Event`** has subclasses `Action` as well as **`CompoundAction`**. A **`CompoundAction`** is an object made up of multiple `Actions` (through **`subevents`**), none of which are dominant. `Action` inherits the facts pertinent to **`Event`**, along with a required **`agent`**, **`object`** and **`instrument`**. Cyc on the other hand, only requires an additional doer. Compound actions are described using its subEvents relation.

3.3 `Break`

A **`Break`** puts its object in a **`Be-Broken`** state, where it can no longer serve its function. [3 min*]: IncurringDamage (A type of PhysicalEvent and thus Event) is the closest term. The main fact about IncurringDamage, similar to `KM` is it damages its object. However, no relation is made in Cyc between the object's being damaged and its usability.

3.4 `Break-Contact`

`Break-Contact` moves two objects from a **`Be-Touching`** state to one where they are not. [2 min*]: Separation-Complete is a similar Cyc event, except that it involves one object being broken into two separate pieces, rather than two objects becoming undetached. [5 min]: Both RemovingSomething and its subclass RemovingSomethingByMovingIt remove an object out of a configuration, which can be used to represent the notion in **`Break-Contact`**. Both **`Break-Contact`** and RemovingSomething have slots for the objects being worked upon.

3.5 `Create`

In a `Create`, an object (which must already "exist" in the sense of a `KM` object) is given a physical, present-time existence, along with information about who made it, and how. [1 min*]: CreationEvent also requires an associated object to be created through the formula (requiredArg1Pred CreationEvent outputsCreated).

3.6 `Make-Accessible`

`Make-Accessible` makes an object be accessible at a destination, through actions such as `Expose`, `Unblock`, or `Unobstruct`, so that other actions can apply to the object. In essence, "undoes" inaccessibility. [10 min*]: No known Cyc counterpart. The closest idea is UnblockingTraffic, which opens a path of transportation.

3.6.1 `Release`

`Release` is a subclass of `Make-Accessible`: `Make-Accessible` → `Unobstruct` → `Release`. In a `Release`, an object is moved out of its enclosure. [14 min]:As mentioned above, `Make-Accessible` has no known counterpart in Cyc, so the closest terms are: ArrangingObjects, and RemovingSomething. [6 min]: TransferOut is used in Cyc as one of the superclasses of the instance TalibanReleaseTruckDrivers. TransferringPossession may also be relevant, as it relates the change of rights associated with an object, similar to a `Release`. None of these however have the notion of removing something out of a confining enclosure.

3.7 `Make-Contact`

`Make-Contact` is the opposite of `Break-Contact`, as it puts two objects in contact with each other, or one object in contact with a `destination`. [2 min*]: ConnectingTogether is the closest Cyc notion, except that it involves a third object, a Connector, which connects the other two (or more) objects. However, a Connector can be a part of one of the two objects being connected.

3.8 `Make-Inaccessible`

`Make-Inaccessible` is the opposite of `Make-Accessible`. Here, an agent causes an object to be inaccessible to a certain destination. Subclasses include `Block`, `Conceal` and `Obstruct`. [2 min]: In Cyc the closest action is BlockingTraffic, which blocks access to a pathway.

3.8.1 `Confine`

`Confine` is a subclass of `Make-Inaccessible`: `Make-Inaccessible` → `Obstruct` → `Confine`. It inherits one main fact from `Obstruct`, where in the result the object is in `Be-Obstructed` state. `Confine` includes an enclosure object to which the given object is to `Be-Confined` to. [4 min]: ControllingSomething is the closest action, in that an object is controlled by an agent. No enclosure-related information was found.

3.8.2 `Be-Confined`

The state `Be-Confined` has the subclass hierarchy `State` → `Be-Inaccessible` → `Be-Obstructed`→ `Be-Confined`. Every `Be-Inaccessible` state has an associated object which is inaccessible, from the destination if provided. `Be-Obstructed` adds the constraint that the object is also obstructed. Finally, `Be-Confined` narrows the distinction further by requiring an object and enclosure, so that the object is confined to the enclosure at the location, and cannot `Move-Out-Of` it. [2 min]: StaticSituation is Cyc's `State`. PhysicalContactSituation, along with the abovementioned ControllingSomething, are the closest terms. PhysicalContactSituation requires its associated objects to be in contact with each other.

3.9 `Move`

A `Move` changes the location of an object. This action has some important requirements:

1. The object must be at the source location (if specified).
2. If the object is held by an agent, that agent must be the one doing the moving.
3. The object cannot be restrained, nor can its path (if specified) be blocked.

At the end of the action, the object should be at its destination, if specified.

[1 min]: Cyc would call this a MovementEvent. It has an associated moved object, transferredThing. [5 min]: Translocation might be a closer match, as it requires having a toLocation. Both `KM` and Cyc have rich ontologies describing movement:

3.9.1 `Carry`

`Carry`, a subclass of `Move`, is defined as a concurrent `Locomotion` of an agent while `Holding` an object. [1 min]: TransportationEvent would be the exact analog, with required transporter and transportees slots. TransportationEvent has many different subclasses parameterized by who the transporter is, through the function (TransportViaFn ?x). There are also specific subclasses such as TransportingPeople or UnderwaterTransportation.

3.9.2 `Enter`

`Enter` is a subclass of `Move` through the hierarchy `Move` → `Move-To` → `Move-Into` → `Enter`. From `Move-To`, `Enter` gets a required `destination` slot. From `Move-Into` it inherits a `the-enclosure` slot. Since `Move-Into` is all about moving inside another object, it requires that any of the portals of the enclosure, along the object's route not `Be-Closed`. Also the object cannot be shut out from the enclosure. `Enter` in simply any `Move-Into` that is also a `ReflexiveCliche`. `ReflexiveCliche` is an intriguing class of actions where the agent is the object. [3 min]: EncasingSomething might be the associated term, except that it is not fleshed out enough. [1 min]: In GuidingAMovingObject the reflexive cliche is not inherently satisfied (but could be). [4 min]: TransferIn is also related, where a transferredThing is at toGeneric.

3.9.3 `Move-Out-Of`

`Move-Out-Of` inherits from: `Move` → `Move-From` → `Move-Out-Of`. `Move-From` has a required source slot. `Move-Out-Of` adds a slot for an object and enclosure. This action depends on, like `Move-Into`, the portals of the enclosure being open and the object not being confined to the enclosure. [4 min]: LeavingAPlace (superclass TransferOut) and GuidingAMovingObject have some of the flavor of moving out of something, but without the notion of an enclosure.

3.10 `Remove`

In a `Remove` a part is removed from its source, negating the associated "part-of" relation. [1 min]: RemovingSomething is a bit more general; the part doesn't have to be a piece of the source, but can be a completely independent object.

3.11 `Repair`

Every object of the `Repair` must `Be-Broken` but can't `Be-Ruined`. [1 min]: SimpleRepairing is the analogue, in which we have the required slot objectTakenCareOf. Other related actions include TakingCareOfSomething and DiagnosingAndRepairingSomething. The difference between something being broken but not ruined is not made in Cyc.

3.12 `Transfer`

A **`Transfer`** has an object, and can have a donor and recipient. The donor, if specified, must possess the object beforehand. Afterwards, the recipient possesses the object instead. [1 min]: TransferringOwnership is the same notion of an object changing hands. TransferringPossession is its superclass, dealing with abstract rights of the fromPossessor and toPossessor which are altered in the event. Cyc has a rich set of subclasses of such actions, including Stealing-Generic, BorrowingSomething, and SaleByCreditCard.

3.13 `Encode`

This action has not yet been defined in `KM`. [1 min]: Cyc has Encoding, which is a collection of actions where data in a InformationBearingThing is compressed from a more natural format. Its superclass is IBTRecoding, which are the set of events where an InformationBearingThing (IBT) is copied to another format, so it inherits the requirements that AccessingAnIBT and IBTGeneration-Original are subevents. Other interesting superclasses of Encoding include IBTGeneration and InformationTransferEvent, through which the inherited slots include infoTransferred, informationOrigin, and informationDestination.

3.14 `Read`

`Read` also has not been defined in `KM`. [1 min]: If `Read` is meant to be the opposite of `Encode`, Decoding is the relevant Cyc term, though it does not appear to be fleshed out at all. (All it does is inherit facts by virtue of being a subclass of IBTRecoding.) [1 min]: There is also Reading, a subclass of AccessingAnIBT, where the informationOrigin is a type of TextualMaterial.

4 Conclusions and Discussion

Cyc and `KM` have somewhat orthogonal means of representing change. Cyc's approach is much more descriptive and hierarchical, while `KM`'s is much more functional. Cyc can teach us much about actions and properties of them, but `KM` can actually *simulate* these actions. We note that `KM` does have the capability to support a similar descriptive database of actions, but simply has not been around long enough to accumulate the ontology Cyc has. Note that `KM` is more powerful than typical frame-based languages as it allows rules, constraints, and multiple values for slots.

In terms of the situation-action dichotomy, Cyc appears to have little fundamental support for casting actions as objects of change between states of the world. On the other hand, `KM` keeps track of all situations in every asserted action sequence, so that it can check facts about any situation. Note that this was not possible in STRIPS, which only kept track of the current state. It can also store multiple possible histories. `KM` goes on even further to provide solutions for the frame and ramification problems.

Both knowledge bases are hierarchical and thus implement inheritance. Cyc's notion of inheritance is through sub-typing of concepts, so that axioms apply not only to one class, but all its subclasses. Instances of a class in `KM` inherit slots and values from superclasses. For the casual browser, it is difficult to discern all the properties of an action in either formalism, since most facts are inherited from superclasses. Properties of an action are scattered up and down the hierarchy chain.

Both Cyc and `KM` use non-monotonicity. `KM` uses non-monotonic reasoning to apply actions and temporally project fluents, while Cyc uses second-order predicates such as (minimize CycFormula) or (minimizeExtent predicate), to apply the Closed World Assumption to the extent of the given predicate, mainly instances of ActorSlot. `KM` has a similar use of negation-as-failure, where value-less expressions are treated as false.

Following is a summary of the selected actions and states that were compared. We include each `KM` action, the Cyc equivalent, and the time it took to find the equivalent Cyc action. If the time is starred it is only a lower bound. We also include a subjective notion of "Quality of Match" on a scale from 1 (poor) to 10 (perfect), along with some notes.

KM action	Cyc equivalent	Time (min)	Quality of Match	Notes
`Break`	IncurringDamage	3*	7	Cyc makes no relation between damaged and usability as `KM` does.
`Break-Contact`	Separation-Complete	2*	5	`Break-Contact` involves two objects; Separation-Complete breaks an object into two pieces.
	RemovingSomething	5	6	RemovingSomething is a bit closer in that one object is moved out of a configuration.
`Create`	CreationEvent	1*	10	Same meaning.
`Make-Accessible`	UnblockingTraffic	10*	1	No known match; UnblockingTraffic is the closest concept.
`-Release`	ArrangingObjects	14	2	No notion of enclosure.
	RemovingSomething	14	4	Does have idea of being moved out.
	TransferOut	6	3	Has general notion of moving something out or away.
	TransferringPossession	6	2	Can at least relate the change in rights.
`Make-Contact`	ConnectingTogether	2*	10	ConnectingTogether is a little more general in that a third object connects the other two.
`Make-Inaccessible`	BlockingTraffic	2	1	No equivalent Cyc term.
`-Confine`	ControllingSomething	4	3	Same in that the object is controlled by some agent. Nothing about enclosures.
`-Be-Confined`	PhysicalContactSituation	2	2	In a PhysicalContactSituation objects must be touching each other so it could be used to model confinement.

KM action	Cyc equivalent	Time (min)	Quality of Match	Notes
`Move`	MovementEvent	1	9	Match, except for `KM` preconditions.
	Translocation	5	9	Like a MovementEvent, but requires a toLocation slot.
`-Carry`	TransportationEvent	1	10	Cyc refines TransportationEvent to include mode of transport, or vehicle involved.
`-Enter`	EncasingSomething	3	3	Not fleshed out enough.
	GuidingAMovingObject	1	6	Need to set deliberateActors = objectControlled.
	TransferIn	4	5	Right notion, without idea of going inside something else.
`-Move-Out-Of`	LeavingAPlace, GuidingAMovingObject	4	5	Moving out of something, but without notion of enclosure.
`Remove`	RemovingSomething	1	10	RemovingSomething is more general in that the removed object does not have to be a part of the source.
`Repair`	SimpleRepairing, TakingCareOfSomething, DiagnosingAndRepairingSomething	1	8	No difference is made between being fixable and permanently ruined like in `KM`.
`Transfer`	TransferringOwnership	1	10	TransferringPossession is even more general, and has a rich set of related actions.
`Encode`	Encoding	1	9	In an Encoding, data is compressed.
`Read`	Decoding	1	?	Not fleshed out, but the analog if `Read` = Opposite(`Encode`)
	Reading	1	10	The analog where `Read` = reading a book.

References

[Clark and Porter, 1998] Clark, P. and Porter, B. (1998). *KM (v1.3): Users Manual.* Knowledge-Based Systems Group, Univ. of Texas at Austin, Austin, Texas. Available at `http://www.cs.utexas.edu/users/mfkb/km.html`.

[Clark and Porter, 2000] Clark, P. and Porter, B. (2000). *KM (1.4): Situations Manual*[12].

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

[McCarthy and Buvač, 1994] McCarthy, J. and Buvač, S. (1994). Formalizing Context (Expanded Notes). Technical Note STAN-CS-TN-94-13, Stanford University.

[McCarthy and Hayes, 1969] McCarthy, J. and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence[13]. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press.

[Shanahan, 1999] Shanahan, M. (1999). The Event Calculus Explained[14]. In Wooldridge, M. J. and Veloso, M., editors, *Springer-Verlag Lecture Notes in Artificial Intelligence*, 1600, pages 409–430. Springer-Verlag.

[Thielscher, 1996] Thielscher, M. (1996). Ramification and Causality[15]. Technical Report TR-96-003, International Computer Science Institute, Berkeley. (A revised version appeared in the *Artificial Intelligence Journal*, 89(1–2):317–364, 1997).

5 Index

---

[12] http://www.cs.utexas.edu/users/mfkb/manuals/situations.ps.Z

[13] http://www-formal.stanford.edu/jmc/mcchay69.html

[14] http://www-ics.ee.ic.ac.uk/ mpsha/ECExplained.ps.Z

[15] ftp://ftp.icsi.berkeley.edu/pub/techreports/1996/tr-96-00 3.ps.gz

Index

5.1 Cyc Collections

AccessingAnIBT, 16
ActionOnObject, 5
Action, 2–5, 7
ActorSlot, 5–7, 17
Agility, 5
AnimalActivity, 3
ArrangingObjects, 13, 18
BlockingTraffic, 13, 18
BorrowingSomething, 16
Breathing, 4
Buying, 5
ChangingDeviceState, 7
CommercialActivity, 5
Competence, 5
ConnectingTogether, 13, 18
Connector, 13
ControllingSomething, 14, 18
Cracking, 2, 3
CreationEvent, 13, 18
CreationorDestructionEvent, 5, 6
Decoding, 16, 19
Dexterity, 5
DiagnosingAndRepairingSomething, 16, 19
DryingSomething, 7
EncasingSomething, 15, 19
Encoding, 16, 19
Event, 3–5, 7, 12
GuidingAMovingObject, 15, 19
HolidaySeason, 4
IBTGeneration-Original, 16
IBTGeneration, 16
IBTRecoding, 16
IncurringDamage, 12, 18
InformationBearingThing, 16
InformationGathering, 3
InformationTransferEvent, 16
IntrinsicStateChangeEvent, 2–4, 7
LeavingAPlace, 15, 19
LiquidTangibleThing, 7
MovementEvent, 14, 19
PartiallyTangible, 3
PhysicalContactSituation, 14, 18
PhysicalEvent, 2, 12
Reading, 16, 19
RemovingSomethingByMovingIt, 12
RemovingSomething, 12, 13, 15, 18, 19
RepeatedEvent, 4
Role, 5
SaleByCreditCard, 16
ScriptPerformanceAttribute, 5
Separation-Complete, 12, 18
SeparationEvent, 2, 3
SimpleRepairing, 16, 19
SingleDoerAction, 3
Situation-Temporal, 4
Situation, 4, 5
SolidTangibleThing, 3
StaticSituation, 4, 14
Stealing-Generic, 16
TakingCareOfSomething, 16, 19
TalibanReleaseTruckDrivers, 13
TextualMaterial, 16
TimeInterval, 7
TransferIn, 15, 19
TransferOut, 13, 15, 18
TransferringOwnership, 16, 19
TransferringPossession, 13, 16, 18, 19
Translocation, 14, 19
TransportationEvent, 15, 19

TransportingPeople, 15
UnblockingTraffic, 13, 18
UnderwaterTransportation, 15
WritingByHand, 6
WritingImplement, 6

5.2 Cyc Predicates

actors, 5, 12
bodilyDoer, 5
causes-EventEvent, 6
causes-PropProp, 6, 7
causes-SitProp, 7
causes-ThingProp, 7
damages, 5, 12
doneby, 5
fromPossessor, 16
fromState, 7
inReactionTo, 6
infoTransferred, 16
informationDestination, 16
informationOrigin, 16
inputsCommitted, 6
inputsDestroyed, 6
inputs, 5, 6
instrument-Generic, 6, 12
intendedBeneficiary, 12
objectOfStateChange, 3
objectTakenCareOf, 16
outputs, 12
performanceLevel, 5
postActors, 5
postEvents, 6
postSituation, 6
preconditionFor-Events, 6
preconditionFor-PropSit, 6
preconditionFor-Props, 6
preconditionFor-SitProp, 6
products, 5
prospectiveSeller, 5
relationAllExists, 5
requiredArg1Pred, 3, 5
resultantMentalObjects, 7
skillRequired, 5
socialParticipants, 12
subEvents, 12
target, 12
toGeneric, 15
toLocation, 14, 19
toPossessor, 16
toState, 7
transferredThing, 14, 15
transportees, 15
transporter, 15

5.3 `KM` Terms

`*Fluent`, 8
`*Global`, 8
`*Inertial-Fluent`, 8, 10
`*Non-Fluent`, 8
`Action`, 8, 9, 12
`Be-Broken`, 12, 16
`Be-Closed`, 15
`Be-Confined`, 14, 18
`Be-Inaccessible`, 14
`Be-Obstructed`, 14
`Be-Ruined`, 16
`Be-Touching`, 12
`Block`, 13
`Break-Contact`, 12, 13, 18
`Break`, 9, 12, 18
`Carry`, 15, 19
`CompoundAction`, 12
`Conceal`, 13
`Confine`, 14, 18
`Create`, 13, 18
`Encode`, 16, 19
`Enter`, 15, 19
`Event`, 8, 12

`Expose`, 13
`Hold`, 15
`Locomotion`, 15
`Make-Accessible`, 13, 18
`Make-Contact`, 13, 18
`Make-Inaccessible`, 13, 14, 18
`Move-From`, 15
`Move-Into`, 15
`Move-Out-Of`, 14, 15, 19
`Move-To`, 15
`Move`, 9, 14, 15, 19
`Obstruct`, 13, 14
`Read`, 16, 19
`ReflexiveCliche`, 15
`Release`, 13, 18
`Remove`, 15, 19
`Repair`, 16, 19
`Situation`, 8
`State`, 14
`Transfer`, 16, 19
`Unblock`, 13
`Unobstruct`, 13
`add-list`, 9, 10

5.4 `KM` Slots

`agent`, 12
`beneficiary`, 12
`del-list`, 9, 10
`destination`, 13, 15
`donor`, 12
`frame`, 9
`has-value`, 9
`instrument`, 9, 12
`is-possible`, 10
`ncs-list`, 9, 10
`next-situation`, 10
`object`, 9, 12
`pcs-list`, 9, 10
`recipient`, 12
`result`, 12
`slot`, 9
`source`, 9
`subevents`, 12
`subevent`, 12
`the-enclosure`, 15
`time`, 12
`value`, 9