# More Thoughts On Views
# Working Note 22

Peter Clark, John Thompson
Knowledge Systems
Mathematics and Computing Technology
Boeing
MS 7L66, PO Box 3707, Seattle, WA 98124

Ken Barker, James Fan,
Bruce Porter, Dan Tecuci, Peter Yeh
Computer Science Dept.
University of Texas
Austin, TX 78712

January 2001

**Abstract**

This working note provides some more discussion on the notion of representing "views" in a knowledge-base.

## 1 Introduction

This working note explores some thoughts about the notion of "views" (including the "modeled-as" relationship), which we have been discussing recently. It revisits – again – one of the constants in our research, namely the desire that a knowledge-base to be able to be selective about the knowledge it uses when reasoning, and the corresponding requirement for some sort of "packaging" of knowledge into units, which can then be switched on or off depending on the task at hand. It also touches on issues of analogy.

There seem to be two important, and perhaps distinct, notions of what a "view" is, which we now describe, and which we will try to relate in this note.

## 2 Two Notions of what a View Is

### 2.1 View = A Coherent Subset of Facts

When we read various passages of text, the author typically describes only certain facts about the objects of interest, and ignores others. In aerospace, for example, there are many annotated diagrams of aircraft showing their parts, but the parts listed and discussed differ: one diagram might show only the parts related to the hydraulic controls; another might show the flight control surfaces; another might show those relevant to passenger seating; another might show parts in the cockpit. Each of these diagrams is associated with narrative describing some facts about the aircraft. We can think of each of these presentations as describing a "viewpoint" on the airplane, exposing some facts and ignoring others, depending on the task at hand. If a user was asked "What are the parts of an airplane?", he/she would produce a different answer depending on the context in which the question was posed, reflecting these different viewpoints (And, interestingly, would be unsure how to answer if the question was posed without any surrounding context).

Similarly, in molecular biology, the Alberts book [1] provides several descriptions of "what is in a cell", but each list is different, showing just a subset of the entire list of parts, depending on the context in which that description is being given. On page 1, a cell is described as "a small membrane-bounded unit filled with a concentrated aqueous solution of chemicals". Later, cells are described in terms of their machinery for protein production; later in terms of machinery for DNA replication; later in terms of structural properties; later in terms of parts relevant to them "eating" organic molecules in order to acquire energy. Again, this illustrates different views of a cell being used for different expository purposes.

This "subset of facts" approach is not limited to static, physical properties of the objects of interest. We can also describe event sequences by including or ignoring various subevents. For example, in Distributed Computing (from the old DCE project [2]), we might describe how a client connects to a server just using subevents related to computer security, or just using subevents related to how actual data is transferred, or both. In Cell Biology, we might describe how a virus invades a cell just using subevents related to how the virus breaks into the cell, and/or how it takes control of the cell.

We will refer to this as the "subset of facts" notion of a view, and is roughly similar to the notion of views in a database, where a subset of rows and columns is selected from a "mother of all tables". However, there is one important difference: In a database, the master table is the core knowledge, and the views are derivatives; In our work, we'd like the *views* to be the core knowledge, and the (various) integrations of those views to be the derivatives. This is a critical distinction which the reader should keep in mind.

Also note that the subset of facts described in text books is not a random subset, but is somehow "coherent", i.e., constitutes a complete and non-redundant set of facts required for describing some particular structure or phenomenon. We will return to this idea of coherence shortly.

### 2.2 View = An Abstraction

A second (and probably related) notion of view is that of a mapping between some domain concept (e.g. a biological cell) and an abstraction (e.g. a container). We say "a cell can be *viewed as* (modeled as) a container", and then provide/have the system infer some mappings, e.g., "...in which the outer membrane is the container-wall, the cytoplasm is the medium filling the container cavity when 'empty',...". Many such abstractions like this exist: For example, in Alberts, a cell is also viewed as (modeled as) a *biological entity*, a *producer* (of energy), a *vehicle* (for genetic information), a *building block*, a *catalyst*, an *organism*, an *agent*, a *consumer*, a *structure*, a *factory*, a *shelter*, a *member of a community*, a *host*, a *living entity*, and a *messenger*, to mention jut a few.

According to this notion of a view, from our recent discussions, a view is an abstraction which is not normally applied to an object, i.e., the axioms contained in the abstraction are "invisible" to the object until the view is applied by some mechanism. The container theory, for example, includes axioms for computing the volume (say) of the medium inside the container's 'empty' space, but these axioms would normally not be applicable to a biological cell unless the container viewpoint had been explicitly applied.

We have to be a bit careful here on two counts. First, this notion of "view" starts to look suspiciously like analogical reasoning. (Some of the examples of views of a cell mentioned earlier are clearly analogies, e.g., a cell as a factory). So we need to be clear as to whether this notion of view is simply an "isa" link which can be switched on or off, or whether it requires more in terms of working out which properties of the abstraction

should transfer to the object of interest. Second, some of the abstractions of cell listed above are not really descriptions of what a cell *is*, but refer to a cell in the context of it performing a particular activity. For example, saying "a cell can be viewed as a producer (of energy)" is really just a short-hand for saying "consider the set of subevents in which a cell produces energy", not requiring any special viewpoint mechanism at all.

### 2.3 Relationship between the Two

One point of contact between these two notions is through the idea of "coherence". A subset of facts is coherent precisely because those facts fully instantiate some abstract model, and there are no extra facts left over. For example, a set of facts listing hydraulic components of the aircraft is coherent because they fully instantiate a "fluid system" abstraction in the KB, while a diagram of an airplane showing (say) a pump, passenger seat 23C, and the left actuator would be considered an incoherent view. Similarly, a set of facts given about a cell in the "Protein Synthesis" section of Alberts is coherent precisely because they instantiate a "chemical synthesis" abstraction in the KB.

A "subset of facts" view could be described as asserting: "here are some facts instantiating an abstraction", while the "abstraction" view could be described as asserting: "here is an abstraction; go find the facts which instantiate it.".

## 3 Mechanisms for Views

### 3.1 Intuitions about Views

Introspectively:

- reading a passage of text brings some facts at the forefront in the reader's mind, other facts are left behind.
- those facts seem to come in coherent "packets"
- various linguistic cues suggest when new packets should be added to the reader's current mental picture of what is being described
- Packets must integrate together, so we can work with multiple packets. (i.e., We want to do more than just pick and work with one solitary packet).
- new packets must integrate with old. (It's not quite as simple as "pick a packet")

Questions are answered with respect to those "active" packets. For example, "What are the parts of a cell?" is a question asked in context, and will produce different depending on what that context is. Without context, it is a highly unnatural question (Consider, one would probably not expect to be asked to "List the parts of a cell." as the first question in an exam). Similarly, open-ended questions such as "Describe...", "What would happen if...", assumes some bounding of the facts which are to be considered. We would like to account for this phenomenon in our knowledge-base.

### 3.2 Do we need Views?

As an interesting note, if all the reasoning which a KB did was backward chaining, and we did not want it to answer "open-ended" questions like this, then there may not be such

an imperative to account for selectively using views. Backward-chaining "automatically" ignores axioms which are irrelevant to a subgoal, so (given suitable indexing), it does not really matter whether there are a 100 or 1,000,000 irrelevant axioms in the KB – backward-chaining will ignore them anyway, and so there is no need to be selective about which axioms are "visible". However, this view breaks down when other modes of reasoning are brought in, at least three of which occur in KM: Constraint checking (Do we check all constraints?), open-ended simulation (Which actions do we consider when answering "what might happen if...?" questions), and automatic classification (Which concepts should we exert effort trying to classify instances under?). It is easily possible to bring the system to its knees without imposing any constraint on these tasks. Similarly for assessing the "coherence" of a representation through some sort of forward-chaining, in the style of KI [3] (something KM currently does not do), again some constraint on this activity is needed. Finally, if we wish the KB to provide context-specific answers to questions (e.g., parts of an X, subevents of X) without pre-enumerating all the possible interesting breakdowns, or we wish to have the KB deal with alternative, conflicting viewpoints, then some additional mechanism is necessary.

### 3.3 Criterea for Triggering a View

When should a view be invoked? Generally, the terminology in the query (or internal subgoal) implies the views to use. For example, if I ask "How does the virus *break in* to the cell?", the phase "*break in*" implies some invasion or attacking viewpoint is to be used. That is, the request for information of a particular type causes a search for viewpoints which can supply that type of information.

We have to be a bit careful here. If we say a view is used "whenever it can supply requested information" then we may as well get rid of some special view mechanism and just throw everything into the KB; After all, a standard inference engine just uses an axiom "whenever it's needed" already. It's perhaps useful to consider the complementary question, "When should a view *not* be invoked?", and look for a non-trivial answer.

Rather than use a view "whenever it supplies useful information", a more restricted criteria for view application is that, given some query Q, if the currently "active" views can already supply a non-nil answer, then *don't* invoke additional views (even if they can supply additional answers). Only if no active view can supply an answer would the KB then look for additional views which could be made active (and then does the system use one of these? all of these? ask the user?). The intuition here is that if the KB is asked "What are the parts of a cell?", and the currently active views already can produce an answer, then the KB doesn't need to pull in every other view which has something to say about parts of a cell – instead, it will produce an answer which is in the context of its current reasoning. This idea probably seems heretical to a logician, as it means that the answer to a query depends on the context in which it was asked. But this is precisely the phenomenon which we are trying to account for, and which we have argued is important for large-scale, multifunctional KBs to be viable.

## 4 Elaboration Tolerance

If we are going to allow views to be either used or ignored, this poses a technical requirement that each view is *elaboration tolerant* [4], That is, that it can correctly combine with other views, and (more importantly), that a view can be removed from a composite

representation without breaking it.

A serious type of breaking which occurs with non-elaboration-tolerant representations is when an axiom in one view/component refers to an object whose existence is declared in another, and then the latter view/component is removed, resulting in a reference to a now non-existent object. For example, in the "purchase" view of a restaurant visit script, we might have an axiom stating that "the diner pays before the eating occurs", "the eating" referring to some event declared in the dining viewpoint. But if the dining viewpoint is hidden, then there is no referent for "the eating". We need to make sure this is not problematic, by ensuring reasoning can still proceed in this circumstance. For example, the existence of "the eating" also needs to be declared in the "purchase" view, or some mechanism is needed to ensure that a request for "the eating" causes a search for other viewpoints which declare its existence.

A related problem for elaboration tolerance is the issue of coreference – if the same object is declared to exist in multiple viewpoints, how is the reasoner to know these declarations all refer to the same object? they are intended to be the same? There are several ways of ensuring this in KM:

- For single-valued slots, multiple declarations of the slot-filler will be automatically considered coreferential by KM and unified.
- Use of the "find-or-create" command in KM (`the+`), which will first search for an object, and only if it is not there will it create it.
- Use of KM's heuristic unification. Given multiple declarations for the fillers of a multi-valued slot, e.g., declarations in different views, KM heuristically tries to determine coreference between those value-sets using its set unification operator (`&&`). Additional guidance about coreference can be provided by the user using KM's instance tagging mechanism (i.e., with expressions of the form `(a` *class* `called "`*name*`")`).

# 5 Mechanisms in KM for Bounding Theories

KM has three possible mechanisms for bounding a theory, and switching it in and out:

1. The `(in-theory ...)` ... `(end-theory)` mechanism (see KM's release notes). Here axioms bounded by the declaration are internally stored in a different memory partition, which can then (manually) be made visible or invisible. There is currently no internal KM mechanism for automatically deciding when to use a theory.
2. The "views" mechanism. Currently, a view is implemented as a mapping between a concept and a generalization, which is only applied when that concept is coerced to be a subclass of that generalization. Coercion can occur when a `must-be-a` constraint is imposed on an instance, or when the domain or range constraints of a slot are imposed on an instance. For example, if the user queries for `(the contents of (a Cell))`, and the domain of the slot `contents` is `Container`, then the cell will be coerced to be a container, and any view declaring how a cell can be viewed as a container will be applied.
3. The "prototype" mechanism. A prototype bounds a set of "forall-exists" assertions about an object. Prototypes are currently always used, although it would be straightforward to prevent certain prototypes being used under certain conditions.

We provide some illustrative KM code for these in the appendices of this working note.

# 6 Open Issues

We now list some additional open issues which come to mind.

## 6.1 Computing How a Generalization Maps to an Object

In our notion of views as an abstraction, we've assumed the user *specifies* how domain objects map to objects in the abstraction. ("The cell membrane is the container wall; the cytoplasm is the container's cavity's medium; ..."). However, we'd also like the system to *infer* (or perhaps even guess) what this mapping should be.

This can partly be achieved by providing definitions for the objects in the abstraction, for example: The container wall is the part surrounding all the other parts; the portal is a 2D surface intersecting the container wall; etc. Given such definitions, and the information that (say) a cell should be viewed as a container, domain objects satisfying these definitions can be identified (by KM) and thus automatically be assumed to fill those roles. Conversely, if a user-supplied view *states* which domain objects fill the abstract roles, then the facts in the definitions will hold for those domain objects. For example, given the container wall is defined as the part surrounding all the other parts, and the cell membrane surrounds all the other parts of a cell, KM could infer the cell membrane must therefore be the container wall; conversely, given (from the user in the view declaration) that the cell membrane *is* the container wall, KM can conclude that the cell membrane must therefore surround all the other parts of the cell. Inference can thus proceed in either direction, depending whether the mapping, or facts implying the mapping, are supplied.

## 6.2 Analogical Reasoning

We haven't accounted for the situation where not all the abstraction's properties "make sense" for the object to which it is being applied.

## 6.3 Multiple, Alternative Theories / Views

We haven't accounted for the phenomenon when there are multiple ways of viewing the same object, and we want to work with just one of them.

For example, DNA can be thought of as just a linear object with regions in, or as a pair of strings, or as a chain of nucleotide pairs. We may wish to view DNA at different levels of granularity for different tasks.

# References


[1] B. Alberts, D. Bray, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Essential Cell Biology*. Garland, NY, 1998.

[2] Peter Clark and Bruce Porter. The DCE help-desk assistant project (working note 8). (http://www.cs.utexas.edu/users/pclark/working_notes/), 1996.

[3] Ken Murray and Bruce Porter. Controlling search for the consequences of new information during knowledge integration. In *Proc. 6th Int. Workshop on Machine Learning*, 1989.

[4] John McCarthy. Elaboration tolerance. In *Proc 4th Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense '98)*, 1998.

## Appendix A1: Views Implemented using Theories

Here is to put the two "theories" about `Invade` and `Deliver` in different partitions, which can then be made visible or not. The disgusting KM commands `[1]` ensure that the appropriate coreferences and relations hold when both theories are visible simultaneously.

```
(reset-kb)
(*Invasion has (instance-of (Theory)))
(*Delivery has (instance-of (Theory)))

(in-theory *Invasion)

(every Invade has
  (subevents (
    (a Arrive called "a1")
    (a Break called "b1")
    (a Enter called "e1"))))

(end-theory)

(in-theory *Delivery)

(every Deliver has
  (subevents (
    (a Arrive called "a2")
    (a Give called "g2"))))

(end-theory)

(Virus-Attack has (superclasses (Invade Deliver)))

;;; [1] This is a disgusting way of embedding coreference statements in the KB:
(every Virus-Attack has
  (subevents (
      (   ((the Arrive subevents of Self) called "a1")  ; [1]
       == ((the Arrive subevents of Self) called "a2") )
      (if   (has-value ((the Enter subevents of Self) called "e1")) ; [1]
       then (((the Enter subevents of Self) called "e1") has
            (before (((the Give subevents of Self) called "g2")))))
      )))


#|
KM> (the subevents of (a Virus-Attack))
NIL

KM> (see-theory *Invasion)
KM> (the subevents of (a Virus-Attack))
(_Arrive3 _Enter5 _Break4)

KM> (hide-theory *Invasion)
KM> (see-theory *Delivery)
KM> (the subevents of (a Virus-Attack))
(_Arrive7 _Give8)
```

```
KM> (see-theory *Invasion)
(*Invasion *Delivery)

KM> (the subevents of (a Virus-Attack))
(_Give13 _Arrive10 _Break11 _Enter12)

KM> (the before of (the Enter subevents of (thelast Virus-Attack)))
(_Give13)
|#
```

## Appendix A2: Views Implemented using `views` ("model-as") slot

Here, rather than switching the *axioms* for `Invade` and `Deliver` in and out of the KB, we switch the *connection* between `Virus-Attack` and these two generalizations in and out. This doesn't really do justice to the notion of a view here: Simply stating `(as-a (Invade))` is vacuous unless some other slot-mappings are also provided.

```
(reset-kb)

(every Invade has
  (subevents (
    (a Arrive called "a1")
    (a Break called "b1")
    (a Enter called "e1"))))

(every Deliver has
  (subevents (
    (a Arrive called "a2")
    (a Give called "g2"))))

(every Virus-Attack has
  (useful-views ( ; i.e., model-as
    (as-a (Invade)) ; would normally also spell out the slot mappings too
    (as-a (Deliver)))))

;;; [1] This is a disgusting way of embedding coreference statements in the KB:
(every Virus-Attack has
  (subevents (
      (   ((the Arrive subevents of Self) called "a1")  ; [1]
       == ((the Arrive subevents of Self) called "a2") )
      (if   (has-value ((the Enter subevents of Self) called "e1")) ; [1]
       then (((the Enter subevents of Self) called "e1") has
            (before (((the Give subevents of Self) called "g2")))))
      )))

#|
KM> (the subevents of (a Virus-Attack))
NIL

;;; & (a Invade) "provokes" the view of the Virus-Attack as an Invade.
;;; We normally would expect this provocation to be done indirectly by the
;;; user (e.g., he/she requests the value of some slot specific to Invade),
```

```
;;; rather than explicitly as is done here.
KM> (the subevents of ((a Virus-Attack) & (a Invade)))
(_Arrive3 _Enter5 _Break4)

KM> (the subevents of ((a Virus-Attack) & (a Deliver)))
(_Arrive7 _Give8)

KM> (the subevents of ((a Virus-Attack) & (a Invade) & (a Deliver)))
(_Give13 _Arrive10 _Break11 _Enter12)

KM> (the before of (the Enter subevents of (thelast Virus-Attack)))
(_Give13)
|#
```

## Appendix A3: Views Implemented using "prototypes"

Very similar to the previous version, here using user interaction to select which prototypes to use. Note we could have several prototypes for the same class, each describing different facets of it.

```
(reset-kb)

(a-prototype Invade)

((the Invade) has
  (subevents (
    (a Arrive called "a1")
    (a Break called "b1")
    (a Enter called "e1"))))

(end-prototype)

(a-prototype Deliver)

((the Deliver) has
  (subevents (
    (a Arrive called "a2")
    (a Give called "g2"))))

(end-prototype)

(Virus-Attack has (superclasses (Invade Deliver)))

;;; [1] This is a disgusting way of embedding coreference statements in the KB:
(every Virus-Attack has
  (subevents (
      (   ((the Arrive subevents of Self) called "a1")  ; [1]
       == ((the Arrive subevents of Self) called "a2") )
      (if   (has-value ((the Enter subevents of Self) called "e1")) ; [1]
       then (((the Enter subevents of Self) called "e1") has
            (before (((the Give subevents of Self) called "g2")))))
      )))

;;; Now I add special code to make use of the prototypes *interactive*...
```

```
#|
KM> (the subevents of (a Virus-Attack))
Use prototype _ProtoInvade1 for _Virus-Attack9? n
Use prototype _ProtoDeliver5 for _Virus-Attack9? n
NIL

KM> (the subevents of (a Virus-Attack))
Use prototype _ProtoInvade1 for _Virus-Attack10? y
Use prototype _ProtoDeliver5 for _Virus-Attack10? n
(_Arrive13 _Enter11 _Break12)

KM> (the subevents of (a Virus-Attack))
Use prototype _ProtoInvade1 for _Virus-Attack15? n
Use prototype _ProtoDeliver5 for _Virus-Attack15? y
(_Arrive17 _Give16)

KM> (the subevents of (a Virus-Attack))
Use prototype _ProtoInvade1 for _Virus-Attack24? y
Use prototype _ProtoDeliver5 for _Virus-Attack24? y
(_Arrive27 _Enter25 _Break26 _Give29)

KM> (the before of (the Enter subevents of (thelast Virus-Attack)))
(_Give29)
|#
```