Using Views in a Knowledge-Base
Working Note 19

Revision 2

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

November 2000

Abstract

In this working note, we sketch out some current thoughts on the use of "views" in a knowledge-base. A view is an explicit representation of how a general concept can be applied to a domain-specific concept. By making views explicit, we can control which, and how, more general concepts can be used to represent a domain-specific concept. We also discuss the use of views to select alternative theories describing ("implementing") the general concept, in the spirit of compositional modeling.

1 Introduction

A conventional view of a knowledge-base is as follows: A knowledge-base contains knowledge distributed among a number of different classes. Each class denotes a different abstraction/generalization of objects in the world being represented, and knowledge from various generalizations is applied to an object through "isa" links in the KB. The complete representation of an object is the integration of knowledge from all these different generalizations.

However, we can also think of a knowledge-base in another way: A knowledge-base contains a number of different "models" or "components", each model being a small collection of axioms describing some abstract phenomenon, denoted by a class. These models or components have two parts: an interface, namely the set of slots and classes (predicates and types) which it manipulates (i.e. can provide answers to questions about), and an implementation, namely the axioms themselves which compute those answers. To represent a domain-specific concept, we have to specify three things:

1. Which models should be applied, e.g. we might want to model an airplane as a container for answering a "capacity" question.

2. How those models should be applied, ie. how the models' interfaces should be mapped onto the domain-specific concept. For example, when modeling the airplane as a container, we may want to consider the plane's fuselage to be the container wall.

3. Which implementation of the model to use. For example, we might want to use a simple set of axioms about a container which ignores consideration of shapes, or a more complex set of axioms which take shape into account.

In a traditional knowledge-base, these choices are implicit and hard-wired: "isa" links specify which generalizations (models) to apply; axioms specify how to compute slot-values for the general concept from slot-values on the domain-specific concept in a single, fixed way; and there is no notion of multiple implementations of a generalization. We can thus view a traditional KB as the fixed application of a fixed set of models in a fixed way, to represent domain-specific concepts.

Rather than this fixed approach, we would like to allow some flexibility in the choice of models and how they are applied to construct domain-specific representations. This is desirable for several reasons: First, it is inefficient to apply every model all the time. Second, we may want to apply the same model in different and possibly conflicting ways (for example, the answer to the question "Is the airplane full?" may depend on whether the airplane is viewed as a container of people, or a container of cargo). Third, in the spirit of compositional modeling, we may want to allow different implementations of the same model to be used for different tasks.

The first two of these three requirements – specifying which and how generalizations should be applied – is addressed through the use of "views", the topic of this working note. The idea of a view is to make explicit the connections between a domain-specific concept and its generalizations, and thus to allow those connections to be made selectively, depending on the task at hand.

The third requirement – to allow alternative implementations of the same generalization – is essentially the task of compositional modeling. We discuss how this might also be handled using views in Section 4.

As of writing (November 2000), the ideas in this working note are only partially implemented. The goal of this note is to set down our current thinking on this topic.

2 Important Concepts

2.1 Interface and Implementation

There are several useful notions from object-oriented programming, software reuse, and artificial intelligence which we make use of. First, using object-oriented terminology, we separate the "interface" and "implementation" of a generalization. A generalization's interface[1] is the set of slots and classes (predicates and types) which the axioms in the generalization manipulate. The implementation of the generalization are the axioms themselves.

2.2 Views

In order to apply a generalization's axioms to a domain-specific concept, we need to specify how the data required by those axioms – namely the truth-value of predicates in the generalization's interface – can be computed given the domain-specific concept. For example, if a container generalization uses the predicate `capacity`, and is applied to an airplane (say), then we need to state how `capacity` is defined in terms known facts

[1] Or perhaps this is closer to the notion of a theory's signature?

about airplanes (e.g., use the predicate `max-passengers`, say). In typical knowledge-bases, these connections are hard-wired through rules on slots or through the "subslot" hierarchy. However in our approach here, we will bundle these "connection" axioms into an explicit object called a view. A view is a reification of these connection axioms, and is somewhat similar to the notion of a "wrapper" in databases, and Novak's use of views in software reuse [1].

Note that our views are, in some ways, exactly the opposite of Acker's use of "viewpoints" in the Botany Knowledge-Base [2]. In the Botany Knowledge-Base, Acker's viewpoints extracted knowledge out of a target representation, while here we wish to *construct* that target representation *from* views.

2.3 Standardized Interfaces

Drawing from the literature on software reuse and compositional modeling, it may also be desirable to have multiple implementations available of the same concept. For example, one representation (implementation) of a container may ignore possible expansion of the container, or the requirement that objects can fit through it's portals, while a more complex representation would take these into account.

To allow these representations to be interchangable, it is important that they are all written using the same interface – that is, the predicates and classes they manipulate are the same (or one a superset of the other), so that we can change the implementation (representation) without having to necessarily change the "connections" between the representation and the object being modeled, i.e., without changing the views.

2.4 Elaboration Tolerance

A third important requirement is that concept representations be "elaboration tolerant" [3]. An elaboration-tolerant representation is one where we can add conceptual detail by a purely additive process of including more axioms, rather than having to modify existing axioms. Similarly, we should be able to remove detail by simply removing axioms. Our generalizations clearly must be elaboration tolerant if we are going to allow them to be switched in and out – in other words, the representation of an object shouldn't "break" if we remove one of its generalizations. This requires that the implementation of one generalization should not depend on the implementation of other generalizations. This can be a somewhat difficult task to achieve, and may require some thought on representational style. In particular, it may require more use of "find-or-create" constructs (e.g., the `:forc` command in Algernon [4], or the `the+` command in KM [5]), so that a path/role chain's failure to "find" an object (because a generalization is not included) triggers a "create" operation to create that object. This requirement for elaboration tolerance has also been identified in software reuse, e.g., the notion of "extension by addition" [6].

2.5 Automatic View Application

Finally, where one generalization does necessarily depend on another, this approach requires ways in which generalization/view application can be automatically triggered at inference time. For example, if one axiom requires finding `(the Enclosure parts of (a Cell))`, the reference to the concept `Encloure`, part of the `Container`'s interface/signature, should inform the inference engine that the `Container` generalization is being appealed to, and thus that it should be included.

3 Specifying Views

3.1 Syntax

A `View` contains a name, the concept to model (the "base concept"), the model to use (the general concept), and (optionally) two sets of mappings:

slot-mapping: A mapping stating how the slots in the generalization are to be computed from slots in the base concept, and

class-mapping: A mapping stating how objects in the base concept map into classes in the generalization.

The slot mappings consist of a set of pairs, each pair being a slot (in the generalization's interface) and an expression for computing it. The class mappings also consist of a set of pairs, each pair being a class (in the generalization's interface) and an expression describing which objects in the base concept are in that class. These expressions thus describe the connections between the generalization and the base concept.

As an example, consider a view of an airplane as a container of people, in which the `Fuselage` is the `Enclosure` of the container, and the `Fuselage-Door` is the `Portal-Covering` of that container:

```
;;; "The view of an airplane as a container of people."
(a View with
  (name ("an airplane as a container of people"))
  (of-a (Airplane))
  (as-a (Container))
  (slot-mapping ((:pair capacity '(the seating-capacity of Self))
                 (:pair occuped-volume
                        '(the number of (the Person contents of Self)))))
  (class-mapping ((:pair Enclosure '(the Fuselage parts of Self))
                  (:pair Portal-Covering '(the Fuselage-Door parts of Self)))))
```

If the `View` is applied to a particular airplane, e.g. `_Airplane01`, whose parts include `_Fuselage01`, `_Fuselage-Door01`, then it will be automatically "compiled" to produce the following:

```
(_Airplane01 has
  (instance-of (Container))
  (capacity ((the seating-capacity of Self)))
  (occupied-volume ((the number of (the Person contents of Self)))))

(_Fuselage01 has (instance-of (Enclosure)))
(_Fuselage-Door01 has (instance-of (Portal-Covering)))
```

This compiled form is exactly what the knowledge engineer would have entered manually, had he/she wanted to "hard-wire" this connection between `_Airplane01` and a container. Thus, a view is a bit like (the specification of) a macro, which expands to a small set of "connection" axioms. However, by reifying the notion of a view itself, we can control which views are applied and when during reasoning.

3.2 Useful Views

If we reify views, we can now list "useful" views which the user or system may which to apply to solve specific problems, e.g.:

```
(every Airplane has
  (useful-views (
    (a View with
      (name ("an airplane as a container of people"))
      (as-a (Container))
      (slot-mapping ((:pair capacity '(the seating-capacity of Self))
                     (:pair occuped-volume
                        '(the number of (the Person contents of Self))))))
    (a View with
       (name ("an airplane as a container of cargo"))
       ...))))
```

A mechanism would then be needed to select which views to use for particular problems, or query the user which views to use.

3.3 Simple Views

For slots which aren't mentioned in the `slot-mapping`, they are considered to correspond to themselves (i.e., they transfer through without being changed). In the simplest case, there are no slot transformations, for example we might write:

```
(every Cell has
  (useful-views (
    (a View with
       (as-a (Container))))))

(every Room has
  (useful-views (
    (a View with
       (as-a (Container))))))
```

In other words, it's sometimes useful to consider a (biological) `Cell` or a `Room` as a `Container` (without any slot transformations). The system or user may then decide to actually select and apply this view to solve a particular problem (through some mechanism not specified here). This would involve asserting the "connection axioms" that the view denotes, as described earlier. We might list the views actually being applied on a `views-in-use` slot.

We can now see what the "isa" (superclass) links in a KB mean in terms of view application. If we say directly in the KB that `Room` is a subclass of `Container`, this is equivalent to always applying the `Container` view on every instance of `Room`. In other words, the "isa" link is simply the permanent, hard-wired application of a simple view in the KB.

3.4 The Bioremediation Example

This is an earlier representation of `Bioremediation`, taken from our earlier AAAI paper [7] and implemented by "hard-wiring" its abstractions as three, overlapping superclasses:

```
(Bioremediation has (superclasses (Conversion Treatment Digestion)))

(every Bioremediation has
  (raw-materials ((Self patient)))
  (eater ((Self theme)))
```

```
(food ((Self patient)))
(theme ((a Microbes)))
(patient ((a Oil)))
(product ((a Fertilizer))))
```

Here is the same thing, but implemented using Views. In the below, by writing "(viewed-as ((the useful-views of Self)))", KM will essentially compile the below into the above for each instance. But now we can "switch out" components by restricting the `viewed-as` to be a subset of the useful views, or even add in new unanticipated views.

```
;;; Switchable components.
(every Bioremediation has
  (theme ((a Microbes)))
  (patient ((a Oil)))
  (product ((a Fertilizer)))
  (useful-views (
    (a View with
      (as-a (Conversion))
      (slot-mappings (
        (:pair raw-materials '(the patient of Self)))))
    (a View with
      (as-a (Digestion))
      (slot-mappings (
        (:pair food '(the patient of Self))
        (:pair eater '(the theme of Self)))))
    (a View with
      (as-a (Treatment)))))
  (viewed-as ((the useful-views of Self))))
```

The original "hard-wired" implementation can thus be seen as a "compiled" version of the latter representation.

4 Compositional Modeling as View Application

So far we have mentioned two uses of views: First, we may wish to ignore some generalizations for a particular task, for comprehensibility and efficiency. This can be achieved with views by excluding certain views for certain tasks. Second, we may wish to apply a particular generalization (e.g. container) in alternative ways (e.g., an airplane as a container of people, or as a container of cargo). Again, views can specify these alternative applications and a view selection mechanism can determine the appropriate one to use.

A third use of views is when we have alternative "implementations" (axiom sets) for the same concept, and we wish to select and use just one of these implementations. This is the key idea behind both compositional modeling [8], and composition in object-oriented programming [9] and software reuse [10].

We can understand compositional modeling in two alternative ways:

1. The automatic selection of views to answer a question.

2. The automatic selection of an implementation of a view.

We present both here, being unsure as the the most appropriate perspective to take.

4.1 Compositional Modeling as View Selection

Consider a simple example taken from Levy [8], in which he wanted to handle multiple, conflicting representations for computing the current output voltage of a battery. One simple model of a battery is that its voltage is constant. An alternative model is that its voltage is a function of room temperature. We might encode these two representations in KM as follows:

```
;;; Model 1: battery has constant voltage.
(every Constant-Voltage-Battery has
  (voltage ((the nominal-voltage of Self))))

;;; Model 2: battery has temperature-sensitive voltage V = max(V0 - kT, 0)
(every Temperature-Sensitive-Voltage-Battery has
  (temperature ((a Number)))
  (voltage (
    (the max of
       (:set ((the nominal-voltage of Self) -
              (the temperature of Self)*(the voltage-decay-constant of Self))
             0.0)))))
```

We then describe how these two models denote different (conflicting) views of a battery:

```
;;; Specify the different views of the battery itself
(every AA-Battery has
  (nominal-voltage (1.5))                        ; volts
  (voltage-decay-constant (0.01))                ; volts/degree
  (useful-views (
    (a View with
      (as-a (Constant-Voltage-Battery))
      (assumptions (*Temperature-Irrelevant)))
    (a View with
      (as-a (Temperature-Sensitive-Voltage-Battery))
      (assumptions (*Temperature-Relevant))))))
```

The model selection task is then to select which view to use to answer a question. Note that the views above have an `assumptions` slot, indicating the assumptions they make. The model selection algorithm must ensure views with conflicting assumptions are not used together.

4.2 Compositional Modeling as Implementation Selection

An alternative interpretation is to consider compositional modeling as *implementation selection*. To encapsulate an implementation, we introduce the notion of a `Theory`. Axioms in a theory may be visible or invisible to the reasoning engine, depending whether the theory is currently "in use". Syntactically, the start of a theory is specified by the KM command `(in-theory ...)`, and the end specified by an `(end-theory)` directive. Implementationally, theories use much of the same machinery as for handling situations. Again, theories are labeled according to the assumptions they make. The implementation would thus look:

```
;;; Theory 1 - simple theory
(*Theory1 has
  (name ("constant voltage battery"))
```

```
  (assumptions (*Temperature-Irrelevant)))

(in-theory *Theory1)

(every Battery has
  (voltage ((the nominal-voltage of Self))))

(end-theory)

;;; Theory 2 - more complex theory
(*Theory2 has
  (name ("variable voltage battery"))
  (assumptions (*Temperature-Relevant)))

(in-theory *Theory2)

(every Battery has
  (temperature ((a Number)))
  (voltage (
    (the max of
       (:set ((the nominal-voltage of Self) -
              (the temperature of Self)*(the voltage-decay-constant of Self))
             0.0)))))

(end-theory)
```

We thus describe an `AA-Battery` simply by:

```
(AA-Battery has (superclasses (Battery)))

(every AA-Battery has
  (nominal-voltage (1.5))                    ; volts
  (voltage-decay-constant (0.01))            ; volts/degree
```

Again, each implementation of `Battery` must use the same interface in order that they can be interchanged. The model selection task is then to select the best "implementation" of `Battery` which can answer the question. Some KM directive like `(use-theory ...)` would be used to state (but not select) which theory to use.

4.3 Compositional Modeling, Object-Oriented Style

We finally mention an object-oriented perspective on compositional modeling. In this perspective, the class `Battery` is an abstract class, namely a class which only specifies an interface, and has no implementation specified. Different implementations of `Battery` are then standard subclasses of battery. The model selection task, then, is to select the appropriate subclass of `Battery` to use for answering a particular query about a particular battery. This perspective still has some wrinkles in it, however. In particular, how does the KB distinguish subclasses denoting different implementations of the *same* concept from subclasses denoting different subtypes of that concept? It is not completely clear that this perspective is a useful one for the task of automatic theory selection, but we mention it nevertheless for interest and possible relevance.

5 Summary

We have presented some initial ideas on using views in this Working Note. In particular, we wish to control which, and how, generalizations are applied to a domain-specific concept, and also control which how those generalizations are implemented (i.e., what axiom set to associate with those generalizations). An explicit notion of views allows all three of these requirements to be met, and suggests how the modularization of knowledge in a KB can be exploited not just at KB-authoring time, but also at run-time.

References

[1] Gordon S. Novak. Composing reusable software components through views. In *Proc 9th Knowledge-Based Software Engineering Conference (KBSE'94)*, pages 39–47. IEEE, Sept 1994.

[2] Liane Acker and Bruce Porter. Extracting viewpoints from knowledge bases. In *AAAI'94*, pages 547–552. AAAI Press, 1994.

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

[4] J. M. Crawford and B. J. Kuipers. Algernon – a tractable system for knowledge-representation. *SIGART Bulletin*, 2(3):35–44, June 1991.

[5] Peter Clark and Bruce Porter. KM – the knowledge machine: Users manual. Technical report, AI Lab, Univ Texas at Austin, 1999. (http://www.cs.utexas.edu/users/mfkb/km.html).

[6] William Harrison and Harold Ossher. Extension-by-addition: Building extensible software. Research Report RC 16127 (#71662) 9/25/90, IBM Research Division, TJ Watson Research, NY, 1990.

[7] Peter Clark and Bruce Porter. Building concept representations from reusable components. In *AAAI-97*, pages 369–376, CA, 1997. AAAI. (http://www.cs.utexas.edu/users/pclark/papers).

[8] Alon Y. Levy. Irrelevance reasoning in knowledge-based systems. Tech report STAN-CS-93-1482 (also KSL-93-58), Dept CS, Stanford Univ., CA, July 1993. (Chapter 7).

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[10] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings ACM SIGSOFT'93 (Symposium on the Foundations of Software Engineering)*, Dec 1993.