Iterative Refinement of Knowledge Bases with Consistency Guarantees

Stephen F. Correl and Bruce W. Porter
Department of Computer Sciences
University of Texas at Austin
Taylor Hall 2.124
Austin, TX 78712-1188
{correl, porter} @cs.utexas.edu

Abstract

Natural kinds, such as the concepts *toy block* or *photosynthesis*, are ubiquitous in human reasoning yet lack definitional (i.e., individually necessary and jointly sufficient) properties as membership conditions. We want to represent a wide variety of assertions about these natural concepts with precision and logical consistency. Furthermore, we want to be able to verify that the meaning of each concept does not conflict with the constraints implied by its constituents. Classification-based approaches to these tasks have relied on definitional properties in order to deduce subsumption and thus detect inconsistency. These approaches are inapplicable for building representations of natural concepts. Our solution to these problems is twofold. First, we build KBs by iterative refinement; i.e., the KB is constructed through a sequence of editing operations. Second, we define practical, yet formal, properties of concept satisfiability and inheritance that each operator is guaranteed to preserve. When the user builds representations of concepts using the operators, guard procedures ensure that existing concepts are used in a manner consistent with their meaning. We also discuss the computational complexity of guarding consistency, and the characteristics of the system that we believe make these computations practical in an interactive setting.

1. Introduction

Building large, multi-functional knowledge bases (KBs) to support reasoning about real-world situations requires tools with particular strengths. Real-world concepts often lack definitional properties, although assertions about concepts and knowledge about inter-dependencies may be very rich. In order to express new concept definitions in terms of existing ones, some means is necessary to logically check for mismatched assertions. Knowledge solicited from domain experts (or added via translation from other sources) may introduce subtle inconsistencies due to changes in terminology or variations in the level of abstraction. In this setting, methods of guarding against ontological drift are required: the methods must be computationally tractable, they must be task-independent, and they must not rely on definitional properties of concepts. This problem can be viewed in terms of three challenges:

1. Provide assertion language constructs adequate to represent useful facts about concepts.
2. Provide constructs to capture the inter-dependent nature of concept definitions.
3. Ensure that each assertion does not logically conflict, in some useful sense, with other assertions in the KB.

Our approach to challenge (3) recognizes that detecting all inconsistencies in a real-world KB is impractical or impossible because of the level of expressiveness required and the need for scalability. General purpose theorem proving with a relatively expressive first-order logic-based representation language is so time-consuming as to be impractical during knowledge acquisition. Rather than use it, we define localized, limited consistency guarantees that will be enforced as the KB is constructed. These properties are formally defined, yet intuitively meaningful to the user. It must be computationally feasible to check them as construction proceeds.

These guarantees are enforced by means of primitive editing operations that modify the KB. For each edit, a guard computation checks that its effect will not violate consistency. Editing operations available to the user include the primitives as well as more advanced operations that are implemented entirely in terms of them. From the user's perspective, the guards ensure that references to existing concepts are consistent with their meanings. Since we can show that the guards maintain limited consistency across each editing step, we can show that each KB will cumulatively meet the consistency guarantees. Longer term, agenda-driven logical analysis of the KB can proceed in the background without delaying editing response to the user.

This approach is rigorous in terms of guarantees to the user, but practical in that it does not depend on classification by definitional properties. The approach also focuses computation on a question of immediate concern at knowledge acquisition time: is the newly added knowledge stated sensibly with respect to the concepts used to define it? We do not preclude the use of general classification applied to concept definitions, as an inference capability or a problem-solving tool; however, we focus on the task of maintaining the consistency of assertions during knowledge acquisition.

Our approach to challenge (2) is based on the observation that new concepts are naturally built in terms of existing concepts, by referring to their internal details, and by configuring these details to build the new definition. The editing operators allow the user to refer to parts of an existing concept in order to specialize them in their new use. For example, in the case of a *send* concept, we might know that the sender and receiver are specialized agents who are connected to the same network, and that the sender knows the receiver's address, and that the sent message is known to the sender before the *send* and to the receiver after the *send*. Handling this style of knowledge refinement naturally means allowing the user to add detail to any part of a concept description, including inherited parts and parts included through reference to related concepts, while preventing uses which would create conflicting assertions.

Although it would be simpler to restrict concept references to lambda calculus-style calls (e.g. send(x, y, m)), concepts do not seem to fit this mold in the real world. This construct may be sufficiently expressive for asserting a collection of facts about who sent what to whom, but when the time comes to describe new concepts based on *send* (e.g. to describe a *secure-send*, or a *client-server-interaction*), our experience leads us to believe that a more referentially flexible approach is necessary. Our goal is to explicitly support composition without requiring interface specifications or classification criteria.

Our approach to challenge (1) is bounded by our solutions to (2) and (3). We intend to implement any useful construct for expressing an assertion, as bounded by our ability to check for logical conflicts in a practical time frame, and our need to provide useful methods of constructing new concepts. We currently use the KM2 language (see Section 3) to express assertions about concepts, although the overall approach to knowledge base construction may be workable with other representation languages. The consistency guarantees that we ensure as the KB is constructed are explained in Section 2, and several basic editing operations are presented in Section 4. We believe that the properties presented can be tested by guard procedures in a practical time frame, but we have not yet developed a proof of this conjecture. The question of the computational complexity of maintaining consistency across each edit is discussed in Section 5.

2. Consistency Guarantees

Within the framework described above, as the user performs editing operations to construct the KB, the system enforces various consistency guarantees. Described below are two consistency guarantees that apply to concept assertions: *implication inheritance* and *concept satisfiability*.

This paper refers to knowledge bases in two related senses: the software KB constructed by the user, and the corresponding, more formal notion of a KB as a collection of axioms and ground facts. Concept descriptions in the software KB can be translated into a collection of logical axioms (the translation is outlined below), and the consistency guarantees are defined in terms of these axioms. The software KB can also contain instances of concepts and relation links among instances, corresponding to ground facts in the more formal KB. Consistency of the KB with respect to instances (although an important issue) is not addressed in this paper.

Implication Inheritance

As concepts are defined within the ISA hierarchy, and as assertions are added to describe knowledge about the concepts, the *implication inheritance* guarantee ensures that assertions about a concept are stronger than assertions about any of its given generalizations. In other words, for concepts c_1 and c_2 with implications F_1 and F_2, respectively:

Given	$c_1(x) \Rightarrow F_1(x)$	(implications of c_1)
and	$c_2(x) \Rightarrow F_2(x)$	(implications of c_2)
and	$c_1(x) \Rightarrow c_2(x)$	(c_1 ISA c_2), then
enforce	$F_1(x) \Rightarrow F_2(x)$	(Implication Inheritance).

This guarantee ensures that ISA inheritance works intuitively with respect to concept implications. Effectively, this prevents the user from adding knowledge about the specialization c_1 that implies that instances of c_1 will have attributes that contradict their stated attributes as members of the generalization c_2. By contrast, in a classification-based language, this property is guaranteed *a priori* because of the if-and-only-if semantics of definitions, as a side effect of the exhaustive computation of entailment:

Given $c_1(x) \Leftrightarrow F_1(x)$ (definition of c_1)
and $c_2(x) \Leftrightarrow F_2(x)$ (definition of c_2),
then $[c_1(x) \Rightarrow c_2(x)] \Leftrightarrow [F_1(x) \Rightarrow F_2(x)]$ must hold.

In other words, when mathematically sufficient definitions of concepts are given by the user, the implication inheritance guarantee would follow as a tautology.

Concept Satisfiability

The *concept satisfiability* guarantee captures the idea that assertions about an individual concept should not be contradictory. To guarantee this, we must ensure that it is possible to build a model of the predicate logic interpretation of the set of assertions about each

KM2 Syntax	Predicate Calculus Meaning
concept-specification :: $-$ *concept* **slot_filler**$_1$ ... **slot_filler**$_n$	$concept(x) \Rightarrow [\mathbf{slot\_filler}_1(x,x) \wedge \ldots \wedge \mathbf{slot\_filler}_n(x,x)]$
slot_filler :: $-$ $>$*relname* :0 **filler** $\mid$ $>$*relname* :0-1 **filler** $\mid$ $>$*relname* :1 **filler** $\mid$ $>$*relname* :1-n **filler** $\mid$ $>$*relname* :all **filler**	$\mathbf{slot\_filler}(x,s)$ :: $-$ $\neg\exists y[relname(x,y) \wedge \mathbf{filler}(y,s)]$ $\mid \ \forall y_1, y_2 \left\{ \left[\begin{array}{c} relname(x,y_1) \wedge \mathbf{filler}(y_1,s) \\ \wedge\ relname(x,y_2) \wedge \mathbf{filler}(y_2,s) \end{array} \right] \Rightarrow y_1 = y_2 \right\}$ $\mid \ \exists 1 y[relname(x,y) \wedge \mathbf{filler}(y,s)]$ $\mid \ \exists y[relname(x,y) \wedge \mathbf{filler}(y,s)]$ $\mid \ \forall y[\mathbf{filler}(y,s) \Rightarrow relname(x,y)]$
filler :: $-$ (*concept*) $\mid$ (*concept* **slot_filler**$_1$ ... **slot_filler**$_n$) $\mid$ **co-ref**	$\mathbf{filler}(x,s)$ :: $-$ $concept(x)$ $\mid \ concept(x) \wedge \mathbf{slot\_filler}_1(x,s) \wedge \ldots \wedge \mathbf{slot\_filler}_n(x,s)$ $\mid \ \mathbf{co\text{-}ref}(x,s)$
co-ref :: $-$ (self) $\mid$ (self $>$*relname concept*) $\mid$ (self $>$*relname*$_1$ *concept*$_1$ ... $>$*relname*$_n$ *concept*$_n$)	$\mathbf{co\text{-}ref}(x,s)$ :: $-$ $(x = s)$ $\mid \ (relname(s,x) \wedge concept(x))$ $\mid \ \exists y_1 \ldots y_{n-1} \left[\begin{array}{l} relname_1(s,y_1) \wedge concept_1(y_1) \wedge \ldots \\ \wedge\ relname_n(y_{n-1},x) \wedge concept_n(x) \end{array} \right]$

Figure 1: Simplified syntax and corresponding predicate calculus interpretation for KM2. A grammar for KM2 is given on the left. For any parsing of a KM2 expression, the corresponding rows on the right hand side show how logical terms are connected together to form the predicate logic interpretation. Non-terminals are shown in bold, terminal symbols are shown in plain text, and italicized names designate concept or relation names in the KB. (Note: The **slot_filler** non-terminal refers to a slot, its quantification, and its filler. The notation $\exists 1 y f(y)$ stands for "there exists a unique y such that $f(y)$"; i.e., $\exists y f(y) \wedge \forall z f(z) \Rightarrow y = z$.)

concept. In other words, for any concept c with implication formula F,

Given $c(x) \Rightarrow F(x)$	(implications of c), then
ensure $F(x)$ is satisfiable	(Concept Satisfiability).

The guarantee says that given the axioms corresponding to the implications of c, it must be possible to construct a collection of instances and relation links for which the implications hold. (It does not say that every KB must contain such a collection.)

Furthermore, since each collection of implications for each concept is separately quantified, the property does not require that some model exists which at once satisfies all the implications of all the concepts in the KB. Therefore, the effect is limited and, in a sense, local to each concept. We do *not* want to guarantee that no contradiction can ever be derived from a KB, since it is likely this would constrain the expressiveness of the representation language and require an impractical amount of computation to enforce.

By contrast, in a complete classification-based language, the system will discover if a concept is unsatisfiable in the process of computing subsumption. After this computation, the user may choose to retract it, then assert a modified definition.

Before explaining how our two guarantees work together to enforce limited and useful consistency among assertions inherited through the ISA hierarchy, we must define the assertion language.

3. Representation Language

We use a frame-slot-filler language called KM2 to present a linear, text-based form of assertions about concepts. The actual knowledge base structure (which is not described in this paper) is organized as a graph that captures the dependencies among the assertions in the system. KM2 is useful because it allows us to compactly describe assertions about a concept, it has a direct translation into predicate logic, and it can be easily presented with plain text. Figure 1 presents a simplified syntax of KM2 and summarizes a grammar for translating it into predicate logic. The language includes slot quantifications for "there does not exist," "there exists zero or one," "there exists exactly one," "there exists one or more," and "for all fillers there exists a relation."

For example, Figure 2 shows a KM2 concept description of *time* which states that every time is followed by some other time, and that this *followed-by* relation is antireflexive and transitive. (The second and third terms within the outermost brackets can be simplified to the standard expressions of antireflexivity and transitivity, respectively. We left them unsimplified to correspond to the KM2 grammar.)

We emphasize that the contribution described in this paper is in our approach to building KBs rather than in the detail of specific language features. The particular constructs currently in KM2 are there because we can perform consistency testing on them in an interactive time frame during KB refinement, and because

KM2: time
>followed-by :1-n (time)
>followed-by :0 (self)
>followed-by :all (self >followed-by time >followed-by time)

P.C.: time(x) $\Rightarrow$

$$\left\{ \begin{array}{c} \exists y(\text{followed-by}(x,y) \wedge \text{time}(y)) \wedge \neg\exists y(\text{followed-by}(x,y) \wedge (y = x)) \\ \wedge \forall y \left[\exists z \left(\begin{array}{c} \text{followed-by}(x,z) \wedge \text{time}(z) \\ \wedge\ \text{followed-by}(z,y) \wedge \text{time}(y) \end{array} \right) \Rightarrow \text{followed-by}(x,y) \right] \end{array} \right\}$$

Figure 2: Every instance of time is followed by one or more other instances. The *followed-by* relation is antireflexive and transitive over time.

in our experience they are useful in representing assertions. Our approach could certainly be adapted to other languages. The more interesting language issue (and indeed we are working on it) is in how much expressiveness can be supported while keeping the consistency testing problem tractable.

4. Refinement Operators

Refinement operators must be used for all modifications to the KB, and they are the means by which the user defines new concepts in terms of existing ones. Each operator has a guard procedure which ensures that the guarantees described in Section 2 are not violated. Extensive automated editing assistance can be provided within the user interface by combining primitive operations in different ways. A variety of interface paradigms, such as "drag and drop," or graphical KB editing could be implemented. Since guarantees are proven for the primitive operations, the guarantees will be preserved by composite operations that are built from primitives.

Conceivable editing primitives include any possible function over the knowledge representation grammar that underlies the KB. However, for practical reasons we want the guard computations to have a worst case complexity that is acceptable in an interactive setting; e.g., a function that is polynomial in the number of concepts defined in the KB. This tends to favor local edits versus sweeping modifications which cut across many definitions in the KB. (Sweeping modifications could perhaps be implemented as overnight tasks.)

Three examples of primitive operations are given below. Although they can be used to represent sophisticated knowledge, they are somewhat limiting in the order in which they must be applied to build a given KB. (Of course, other primitives could be created to provide more flexibility.) Note that regardless of how we describe the KB, as long as it can be reconstructed from these primitives, we are assured that the desired guarantees hold.

Add-slot We can add knowledge about a new slot to the base concept or to one of the concepts used in its definition.

Co-reference restriction We can assert that a slot filler is defined in terms of some number of instances described elsewhere in the concept implication.

New-concept We can add to the KB a new concept which initially inherits all its properties from a collection of immediate generalizations.

These basic operations are defined more precisely below.

Add-slot

This operation takes as parameters the site within a concept definition where the new slot is to be attached, and the new slot's quantification and filler. The site can be either the base concept itself, or one of the fillers of slots used in its definition. For example, starting from a *data-transfer* concept with no slots, we can perform the following add-slot operations to produce the concept definition shown in Figure 3:

1. Add-slots at data-transfer: every data-transfer has exactly one *from* location and one *to* location, exactly one *before* time and one *after* time, and some *contents*, which is data.
2. Add-slot at (data-transfer >from location): every such location *contains* data.
3. Add-slot at (data-transfer >to location): every such location *contains* data.

A concept is implicitly specialized when it is mentioned as a slot filler; the filler is a *dependent concept* because its definition depends on the existence of the related instance of the base concept. For example, data-transfer in Figure 3 implicitly defines dependent concepts we might call location1, data1, location2, data2, time1, time2, and data3; these are defined as "the location which is the *from* location of a data-transfer," "the data which is contained in the *from* location of a data-transfer," and so on. We refer to dependent concepts by specifying a *path* of relation and concept names leading from the root concept to the dependent concept; e.g. the data referred to in (3) above

KM2: data-transfer
>from :1 (location >contains :1-n (data))
>to :1 (location >contains :1-n (data))
>before :1 (time)
>after :1 (time)
>contents :1-n (data)

P.C.: data-transfer(x) $\Rightarrow$

$$\left\{ \begin{array}{c} \exists 1y \left[\begin{array}{c} \mathrm{from}(x,y) \wedge \mathrm{location}(y) \wedge \\ \exists z(\mathrm{contains}(y,z) \wedge \mathrm{data}(z)) \end{array} \right] \wedge \exists 1y \left[\begin{array}{c} \mathrm{to}(x,y) \wedge \mathrm{location}(y) \wedge \\ \exists z(\mathrm{contains}(y,z) \wedge \mathrm{data}(z)) \end{array} \right] \\ \wedge\ \exists 1y(\mathrm{before}(x,y) \wedge \mathrm{time}(y)) \wedge \exists 1y(\mathrm{after}(x,y) \wedge \mathrm{time}(y)) \\ \wedge \exists y(\mathrm{contents}(x,y) \wedge \mathrm{data}(y)) \end{array} \right\}$$

Figure 3: Every data transfer has exactly one *from* and one *to* location (each of which contains data), and exactly one *before* and one *after* time, and some *contents*, which is data.

would be (data-transfer >to location >contains data). Dependent concepts may be nested in this way to arbitrary (finite) depths in order to create far-ranging assertions regarding the base concept.

Each such operation is guarded; e.g., if the concept definition for location (not shown) specified that every location contains exactly zero or one things (i.e. location >contains :0-1 (thing)), and data is a thing, then implication inheritance would be violated because the *contains* slots in our dependent definition can contain more than one instance of data, and hence more than one thing.

Co-reference Restriction

A co-reference restriction asserts that instances filling a slot are (1) described by a dependent concept occurring somewhere else in the KB, and (2) found by traversing a specified path of relation and concept types starting from the instance of the base concept. For example, we can add the following co-reference restrictions to the KB from Figure 3:

1. Co-reference restriction at (data-transfer >before time): must be in a *followed-by* relation to (data-transfer >after time).
2. Co-reference restrictions at (data-transfer >from location): must be in a *contains* relation to each (data-transfer >contents data), and in an *at-time* relation to (data-transfer >before time).
3. Co-reference restrictions at (data-transfer >to location): must be in a *contains* relation to each (data-transfer >contents data), and in an *at-time* relation to (data-transfer >after time).

The modified definition for data-transfer is shown in Figure 4.

Again, each operation is guarded. For example, suppose some specialization of data-transfer already existed whose *before* and *after* times were co-referent (e.g. null-transfer ... >after :all (self >before time)). Then operation (1) would have been rejected because it would imply the dependent concept (null-transfer >before time) would be in a *followed-by* relation to itself, while at the same time we know that *followed-by* is antireflexive over time (see Figure 2, second clause), violating concept satisfiability. Incidently, the antireflexive clause illustrates the use of negation with co-reference: it asserts that no time instance can be in a *followed-by* relation to itself.

5. Guard Computations

This section discusses the guard computations necessary before allowing primitive operations to proceed. First, note that the new-concept operation can be guarded using multiple applications of the guard procedures of the other two operations.[1] This can be done by treating the slots and fillers inherited from the new concept's generalizations as though they were added one by one.

We can enforce the implication inheritance guarantee by simply ensuring that every assertion at each site in the KB is propagated to every specialization of the site.[2] From the user's point of view this is simply an inheritance policy. The remaining problem is to check concept satisfiability at any site where a change occurs.

When a new slot and filler or co-reference restriction is added at a particular site, our ability to test for satisfiability hinges on restricting the effect of the change to a well-defined locality. We believe that it is sufficient to perform a "site test" (described below) at a limited number of sites within the KB. The number of sites that must be tested is a function of (1) the distance between the base concept and the site of the change (measured by the number of slots that must be traversed to go between them), and (2) the number of co-references in the KB that traverse through the change site. For examples of these metrics, refer

[1] It must also be verified that the new concept name is not a duplicate.

[2] Physical propagation is costly in space; the KB graph structure achieves this effect without copying.

KM2:

```
data-transfer
>from      :1 (location >contains :all (self >contents data)
                        >at-time  :all (self >before time))
>to        :1 (location >contains :all (self >contents data)
                        >at-time  :all (self >after time))
>before    :1 (time >followed-by :all (self >after time))
>after     :1 (time)
>contents  :1-n (data)
```

P.C.: data-transfer$(x) \Rightarrow$

$$\left\{ \begin{array}{c} \exists 1y \left[\begin{array}{c} \text{from}(x,y) \wedge \text{location}(y) \wedge \\ \forall z[(\text{contents}(x,z) \wedge \text{data}(z)) \Rightarrow \text{contains}(y,z)] \\ \wedge \forall z[(\text{before}(x,z) \wedge \text{time}(z)) \Rightarrow \text{at-time}(y,z)] \end{array} \right] \wedge \\ \exists 1y \left[\begin{array}{c} \text{to}(x,y) \wedge \text{location}(y) \wedge \\ \forall z(\text{contents}(x,z) \wedge \text{data}(z)) \Rightarrow \text{contains}(y,z)] \\ \wedge \forall z[(\text{after}(x,z) \wedge \text{time}(z)) \Rightarrow \text{at-time}(y,z)] \end{array} \right] \wedge \\ \exists 1y \left[\begin{array}{c} \text{before}(x,y) \wedge \text{time}(y) \wedge \\ \wedge \forall z[(\text{after}(x,z) \wedge \text{time}(z)) \Rightarrow \text{followed-by}(y,z)] \end{array} \right] \wedge \\ \exists 1y[\text{after}(x,y) \wedge \text{time}(y)] \wedge \exists y[\text{contents}(x,y) \wedge \text{data}(y)] \end{array} \right\}$$

Figure 4: Every data transfer has exactly one *from* and one *to* location, and exactly one *before* and one *after* time, and some *contents*, which is data. The *from* location contains all the *contents* data at the *before* time. The *to* location contains all the *contents* data at the *after* time. The *before* time is followed by the *after* time.

to Figure 3. Suppose we made a change at the (data-transfer >contents data) site. This site is one slot distant from the base concept, data-transfer. The system would have to consider whether the modified meaning of *contents* was consistent with the other slot-fillers defined on data-transfer. In addition, each co-reference which traverses through either the newly added slot, or the *contents* slot must be checked.

We believe the number of site tests required is generally independent of the number of concepts defined in the KB. Obviously, this claim must be proved, and that is one focus of our work currently.

The site test determines whether the combination of slots defined at a site are satisfiable. (Note that the slots' fillers are individually guaranteed to be satisfiable since they have already been added to the KB.) For example, in the case of :1-n versus :0 quantification shown in Figure 5, the two *knows* slots may not be jointly satisfiable. A test based on the slot quantification and fillers is used. The test must reject every unsatisfiable combination; however, it could sometimes reject a satisfiable combination, provided such combinations were neither too common nor especially useful to the user. The test may be conservative as long as it is not impractically conservative.

In order to allow the site test to be performed efficiently, the concept satisfiability guarantee and the representation language have been designed so that pairwise slot/filler comparisons among assertions at the site will suffice. (Again, this claim is not proved here. We give an illustrative example below.) Since our system does not currently support a slot hierarchy, only matching slot names must be considered. Each

KM2:

```
agent
>knows :1-n (data₁)
>knows :0 (data₂)
```

P.C.: agent$(x) \Rightarrow$

$$\left[\begin{array}{c} \exists y[\text{knows}(x,y) \wedge \text{data}_1(y,x)] \wedge \\ \neg \exists y[\text{knows}(x,y) \wedge \text{data}_2(y,x)] \end{array} \right]$$

Figure 5: Every agent knows one or more data_1 things but no data_2 things.

comparison depends on whether the filler of one slot may subsume the filler of the other. In Figure 5, assuming the fillers can be separately satisfied, the site is unsatisfiable whenever data_2 subsumes data_1.

Since subsumption relations must be specified explicitly in our system, testing subsumption often involves little more than traversing the ISA hierarchy. In certain cases (which are beyond the scope of this paper) we do need more involved tests, but we can conservatively approximate the difficult ones.

One reason pairwise slot comparisons suffice stems from a current (and hopefully temporary) limitation of the representation language: it cannot partition a concept among disjoint specializations. This limitation allows a form of semantic drift as the KB is built. For example, suppose the user originally wanted to partition people into vegetarians, who eat no meat, and meat-eaters. The user intended to represent knowledge that all persons are in exactly one of the two categories. The KM2 version (Figure 6) leaves open the possibility

vegetarian (ISA person)	seafood
>eats :0 (meat)	>eaten-by :1-n (person)
meat-eater (ISA person)	>eaten-by :0 (vegetarian)
>eats :1-n (meat)	>eaten-by :0 (meat-eater)

Figure 6: Vegetarians eat no meat. Meat-eaters eat some meats. Seafood is eaten by some people, but not by vegetarians and not by meat-eaters.

that other kinds of people could exist.

Later, the user adds assertions about seafood, which are accepted because concept satisfiability is not violated.[3] However, if the partitioning of persons among vegetarians and meat-eaters was captured in the language, then the seafood assertions would not be jointly satisfiable.

This example illustrates both a strength and a weakness of our approach. The weakness is that since our system does not automatically classify concepts into subsumption hierarchies, it cannot discover all inconsistencies in the KB. The strength is that it need not perform the intractable computations that can result from this feature. (See Appendix A of (Borgida & Patel-Schneider 1994) for an example of such intractability.) These computations can still be performed in the background, without requiring potentially large delays in response time to the user.

To summarize the complexity of ensuring concept satisfiability: we believe that (1) the guarantee can be enforced for any single operation by testing a limited number of sites, (2) a site can be tested with a limited number of pairwise slot comparisons, (3) a slot comparison depends principally on a conservative subsumption test, and (4) subsumption is usually easy to test because it follows from the ISA hierarchy. For these reasons we believe that the complexity will be generally independent of the number of concepts defined in the KB.

6. Related Work

Our view of knowledge representation shares many characteristics with Conceptual Graphs (Sowa 1984). The use of canonical forms in that work enforces syntactic constraints on how concepts can be built. However, since our main thrust is supporting iterative refinement of knowledge, we avoid using lambda-calculus interfaces to concept definitions. Conceptual graphs support inference in order to discover contradictions, but they do not encompass a systematic strategy for preventing inconsistencies as a knowledge base is constructed.

There has been surprisingly little work on finding inconsistencies in task-independent, real-world knowledge bases. Some work in the field of description logics (Brachman & Schmolze 1985) has addressed both practical and theoretical issues. Approaches based on description logic can be divided into those that emphasize completeness and those that emphasize expressiveness.

The basic core of the widely-used CLASSIC system emphasizes completeness and tractability of inference (Borgida & Patel-Schneider 1994). However, the expressiveness of such languages is limited to a subset of first-order logic for which logical entailment can always be computed in polynomial time. For examples of useful knowledge that cannot be expressed under this restriction, see (Doyle & Patil 1991). Systems have combined the expressiveness of description logics and Horn clause rules (Levy & Rousset. 1996), but the focus of this work is on trading expressive power for completeness rather than trading completeness for *limited* consistency guarantees.

By contrast, the LOOM representation system emphasizes expressiveness at the expense of completeness (MacGregor 1991). This system includes a plethora of descriptive forms and supports numerous inference methods, including several approaches to classification. In addition to representing facts, LOOM supports specification of procedural reasoning methods and production rules. Although much more can be said in such a language, much less is guaranteed to the user than in a CLASSIC-style description logic in regards to the inter-relation of concept descriptions and the outcome of inference.

The Knowledge Sharing Effort (Patil *et al.* 1992; Swartout, Neches, & Patil 1993) has focused on how ontologies can be translated and used as the basis of communication or cooperation. Our work is complementary to this effort because we address the issue of how the shared ontologies are originally built. Furthermore, our work shows how separately built ontologies, once translated into a common language, can be analyzed to detect clashes in ontological commitments.

Much work in knowledge acquisition assumes a task model, and uses the model to test the contents of the knowledge base for undesirable conditions. For example, the EXPECT system (Gil & Melz 1996) ensures that the knowledge base contains the structures needed by the problem solving methods declared to the system. This system does not try to address the problem of creating a consistent set of assertions over the concepts in the KB, and it uses a procedural-style language to express problem solving methods.

In the field of expert systems, analysis or verification of OPS-style production rule bases with various practical goals has been investigated, for example in (Ginsberg & Williamson 1993). However, inference in these systems is defined procedurally rather than in terms of first-order predicate calculus semantics. Other work on detecting inconsistencies includes verification of propositional (Ginsberg 1988), or extended propositional (Rousset 1988), knowledge bases.

[3] The instances seafood(s_1), person(p_1), eaten-by(s_1, p_1) satisfy the seafood definition because there is no way to deduce vegetarian(p_1) or meat-eater(p_1).

7. Conclusion

We have presented an approach to building large knowledge bases that provides consistency guarantees. Our approach has parts. First, we provide a set of editing operations for iterative refinement of knowledge bases. Second, for each operation, we implement guard procedures that ensure that concepts are used in a manner consistent with their meaning. In order to perform the guard computations in an interactive time frame, we design the guarantees, editing operations, and representation language to avoid having to compute global consistency.

Acknowledgements We would like to thank Peter Clark, Rob Holte, James Lester, Ken Murray, Jeff Rickel, Nirad Sharma, Art Souther, and the reviewers for their feedback on these ideas. This research was supported by a contract from the Air Force Office of Scientific Research (F49620-93-1-0239).

References

Borgida, A., and Patel-Schneider, P. F. 1994. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research* 1:277–308.

Brachman, R., and Schmolze, C. 1985. An overview of the KL-ONE knowledge representation system. *Cognitive Science* 9:171–216.

Doyle, J., and Patil, R. 1991. Two theses of knowledge representation: language restrictions, taxonomic classification, and the utility of representation services. *Artificial Intelligence* 48(3):261–297.

Gil, Y., and Melz, E. 1996. Explicit representations of problem-solving strategies to support knowledge acquisition. In *Proceedings of the Thirteen National Conference on Artificial Intelligence*, 469–476. AAAI Press.

Ginsberg, A., and Williamson, K. 1993. Inconsistency and redundancy checking for quasi-first-order-logic knowledge bases. *International Journal of Expert Systems: Research and Applications* 6.

Ginsberg, A. 1988. Knowledge-base reduction: a new approach to checking knowledge-bases for inconsistency and redundancy. In *Proceedings of the Seventh National Conference of Artificial Intelligence*, 585–589. Morgan Kaufmann.

Levy, A., and Rousset., M. 1996. Verification of knowledge bases based on containment checking. In *Proceedings of the Thirteen National Conference on Artificial Intelligence*, 585–591. AAAI Press.

MacGregor, R. M. 1991. The evolving technology of classification-based knowledge representation systems. In Sowa, J., ed., *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann.

Patil, R.; Fikes, R.; Patel-Schneider, P.; Kay, D.; Finin, T.; Gruber, T.; and Neches, R. 1992. The DARPA knowledge sharing effort: Progress report. In *Principles of Knowledge Representation and Reasoning: Proceeding of the Third International Conference, Cambridge MA*. Morgan Kaufmann.

Rousset, M. 1988. On the consistency of knowledge bases: the COVADIS system. In *Proceedings of the European Conference on Artificial Intelligence*, 79–84.

Sowa, J. F. 1984. *Conceptual Structures*. Addison-Wesley.

Swartout, W.; Neches, R.; and Patil, R. 1993. Knowledge sharing: Prospects and challenges. In *Proceedings of the 1993 International Conference on Building and Sharing of Very Large-Scale Knowledge Bases.*