Closed Terminologies in Description Logics

Robert A. Weida

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
weida@watson.ibm.com

Computer Science Department
Columbia University
New York, NY 10027
weida@cs.columbia.edu

Abstract

We introduce a predictive concept recognition methodology for description logics based on a new closed terminology assumption. During knowledge engineering, our system adopts the standard open terminology assumption as it automatically classifies concept descriptions into a taxonomy via subsumption inferences. However, for applications like configuration, the terminology becomes fixed during problem solving. Then, closed terminology reasoning is more appropriate. In our interactive configuration application, a user incrementally specifies an individual computer system in collaboration with a configuration engine. Choices can be made in any order and at any level of abstraction. We distinguish between abstract and concrete concepts to formally define when an individual's description may be considered finished. We also take advantage of the closed terminology assumption, together with the terminology's subsumption-based organization, to efficiently track the types of systems and components consistent with current choices, infer additional constraints on current choices, and appropriately guide future choices. Thus, we can help focus the efforts of both user and configuration engine.

Introduction

In a description logic (DL), a knowledge base (KB) includes a *terminology* composed of concept descriptions, plus descriptions of individuals in terms of those concepts (Woods & Schmolze 1992). Previous work in DL has not distinguished between knowledge engineering and problem solving phases of terminology usage. This paper pursues the idea of closing the terminology after it is developed and before problem solving begins. Essentially, our *closed terminology assumption* (CTA) holds that all relevant concepts are explicitly defined, and that every individual, once fully specified, will correspond directly to at least one concept. (We do not make a closed world assumption over individuals.) For certain applications, this CTA enables useful inferences that would be impossible otherwise, e.g., we introduce *predictive concept recognition*: given an individual's unfinished description and some assumptions, we show how to ascertain not only the concepts it already instantiates, but those it might eventually instantiate, and those it cannot.

Consider system configuration, where DL has already found practical success (Wright *et al.* 1993). DL is ideal for describing artifacts such as computer systems, and for maintaining the consistency of large configuration terminologies as they evolve over time. Figure 1 shows the system architecture we have in mind: a configuration engine, which specializes in configuration problem solving, utilizes a DL system to represent and reason with a KB, which contains both a configuration terminology and a description of of the individual configuration being worked on. The configuration engine can make significant use of traditional DL inferences, as well as new inferences introduced here.

The standard *open terminology assumption* (OTA), which presumes an incomplete terminology, is appropriate during knowledge engineering when we are still constructing a domain model of systems, components, and related concepts. (We coined the term OTA to describe standard practice.) However, during a specific configuration task, CTA reasoning is preferable because all relevant concepts, e.g., types of systems, are known. We will show how CTA enhances our ability to rule out concepts that an individual cannot instantiate, and thus draw conclusions from the remainder. We also distinguish between abstract and concrete concepts to formally define when an individual's description may be considered finished, given a closed terminology. Imagine that a user incrementally specifies an individual computer system, in collaboration with a configuration engine. The configuration engine is responsible for ensuring practical choices and recommending desirable ones. In general, choices can be made in any order and at any level of abstraction. Hence, it may remain unclear for some time precisely which type of system will result. We can exploit the closed terminology and its subsumption-based organization to (1) efficiently track the types of systems, components, etc., that are consistent with current choices, (2) infer constraints that follow from current choices and the terminology's specific contents, (3) appropriately restrict future choices, and (4) characterize the most general choices available for refining a particular individual's description. Thus, we can help focus the efforts of both user and configuration engine. This work evolved from DL-based plan recognition in the T-REX system (Weida & Litman 1994; 1992). It is implemented in the K-REP DL system (Mays, Dionne, & Weida 1991). Further details, including proofs of theorems, are provided in (Weida 1996).

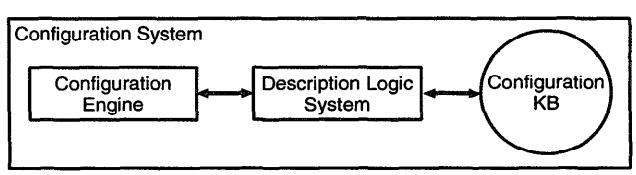

Figure 1: Configuration System Architecture

```
(define-prim-concept COMPANY)
(define-prim-concept CPU)
(define-prim-concept RISC)
(define-prim-concept RAM)
(define-prim-concept DISK)
(define-prim-concept SYSTEM)
(define-prim-concept OS
    SYSTEM)
(define-prim-concept UNIX
    OS)
(define-concept IBM-CPU
    (and CPU (fills vendor IBM)))
(define-concept RISC-CPU
    (and CPU (the technology RISC)))
(define-concept IBM-RISC-CPU
    (and CPU
        (fills vendor IBM)
        (the technology RISC)))
(define-concept IBM-PROCESSOR-DEVICE
    (all processor IBM-CPU))
(define-prim-concept COMPUTER-SYSTEM
    (and SYSTEM
        (the vendor COMPANY)
        (all processor CPU)
        (at-least 1 processor)
        (all primary-storage RAM)
        (at-least 1 primary-storage)
        (all secondary-storage DISK)
        (the operating-system OS)))
```

```
(define-concept UNIPROCESSOR-SYSTEM
    (and COMPUTER-SYSTEM
        (exactly 1 processor)))
(define-concept DUALPROCESSOR-SYSTEM
    (and COMPUTER-SYSTEM
        (exactly 2 processor)))
(define-concept DISKLESS-SYSTEM
    (and COMPUTER-SYSTEM
        (exactly 0 secondary-storage)))
```

```
(define-concept RISC-MULTIPROCESSOR-SYSTEM
    (and COMPUTER-SYSTEM
        (all processor RISC-CPU)
        (at-least 2 processor)))
(define-concept DUAL-IBM-PROCESSOR-SYSTEM
    (and COMPUTER-SYSTEM
        (all processor IBM-CPU)
        (exactly 2 processor)))
(define-concept UNIX-RISC-SYSTEM
    (and COMPUTER-SYSTEM
        (all processor RISC-CPU)
        (the operating-system UNIX)))
```

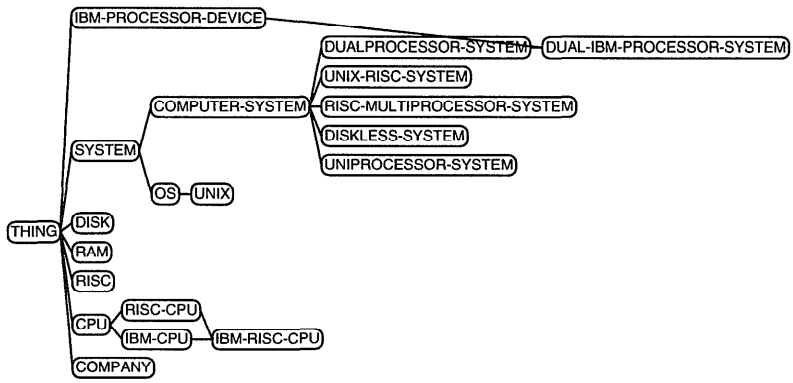

Figure 2: Sample Concept Definitions and Taxonomy

Description Logics

DL systems reason about intensional descriptions of concepts, such as COMPANY, and their instances, such as IBM. Key DL inferences include subsumption, classification, and recognition. Concept *C1 subsumes* concept *C2* when every possible instance of *C2* is also an instance of *C1*, written $C2 \Rightarrow C1$. DL implementations maintain an explicit concept taxonomy, where each concept subsumes its descendants and is subsumed by its ancestors, without exception. When a new concept is defined, *classification* automatically integrates it into the taxonomy. The standard *recognition* inference determines if individual *I* currently instantiates concept *C*.

The K-REP DL provides a formal language for defining concepts and individuals, where *roles* denote binary relations between individuals. A role's *value restriction* is a concept which constrains the range of the relationship; only instances of the value restriction may *fill* the role. Cyclic descriptions are forbidden. A role also has *at-least* and *at-most* restrictions which constrain how many fillers it may have. Restrictions on role R are referred to as *value-restriction(R)*, *fillers(R)*, *at-least(R)*, and *at-most(R)*; they default to THING (the universal concept), the empty set, 0, and ∞, respectively. A concept or individual may inherit from *base* concepts, i.e., subsumers explicitly stated in its definition. Local and inherited properties are combined by logical intersection. The set of roles restricted by concept or individual *X*, written restricted-roles(*X*), is as follows, where R_X denotes role *R* of *X*:

Definition 1 *The* restricted roles *of concept or individual* X *are the roles* R *of* X *for which value-restriction*(R_X) *is properly subsumed by* THING, *or* $|\,fillers(R_X)\,| > 0$, *or at-least*(R_X) > 0, *or at-most*(R_X) $< \infty$.

This paper uses concepts defined in Figure 2, where *all* gives a value restriction. As shorthand, *exactly* combines at-least and at-most restrictions of the same cardinality, and *the* combines a value restriction with a cardinality restriction of exactly one. The resulting taxonomy also appears in Figure 2. Individuals such as IBM are not shown. Assuming that IBM has been defined as a COMPANY and 486DX-33MHz-123 as an individual PROCESSOR, the following description of an individual COMPUTER-SYSTEM is well-formed:

```
(create-individual COMPUTER-SYSTEM123
    (and COMPUTER-SYSTEM
        (fills vendor IBM)
        (fills processor 486DX-33MHz-123)))
```

Although its primary-storage, secondary-storage, and operating-system roles are not yet filled, COMPUTER-SYSTEM123 inherits restrictions on those roles from its base concept, COMPUTER-SYSTEM.

COMPUTER-SYSTEM is a primitive concept; whereas *fully defined* concepts specify necessary and sufficient conditions for class membership, *primitive* concepts specify only necessary conditions. Primitive concepts do not subsume

other concepts unless the subsumption is explicitly sanctioned, e.g., COMPUTER-SYSTEM subsumes UNIPROCESSOR-SYSTEM. A description's primitiveness is characterized by the primitive concepts among its ancestors (inclusive) in the taxonomy:

Definition 2 *The* primitives *of concept or individual* X *are the primitive concepts in the set containing* X *and the transitive closure of its base concepts.*

For example, both primitives(COMPUTER-SYSTEM) and primitives(COMPUTER-SYSTEM123) are the set {COMPUTER-SYSTEM, SYSTEM}.

K-REP supports explicit declarations that sets of primitive concepts are mutually disjoint. Our sample terminology has none for brevity, but declaring suitable disjointness conditions is crucial to sound knowledge engineering. Moreover, it helps to guide recognition as we shall see.

Incremental Instantiation

To help decide if an individual's current description may be finished, we distinguish between *concrete* and *abstract* concepts. A similar distinction was made in configuration work by (Kramer 1991), but we will formally define the notion of a finished individual. Abstract concepts represent commonality among a class of concrete concepts, e.g., an actual system's processor may be of type 486DX-33MHz, which is concrete (fully specific), but not merely of type CPU, which is abstract (too general). All concepts in Figure 2 are abstract; concrete concepts are omitted for brevity. Figure 3 illustrates the taxonomic relationship among abstract concepts (A) and concrete concepts (C); wide arrows connect a concept to its most specific subsumer(s). Thin dashed arrows connect individuals (I) to the most specific concepts they instantiate. We will see that finished individuals must instantiate concrete concepts; before then they can instantiate abstract and/or concrete concepts. We will further characterize the abstract/concrete distinction after introducing the helpful notion of *bijective instantiation.*

A bijective instantiation demonstrates that concept C explicitly accounts for each of individual I's primitives and role restrictions, and that I explicitly respects all of C's primitives and role restrictions:

Definition 3 *Individual* I bijectively instantiates *concept* C *iff*

1. *primitives*(I) $\equiv$ *primitives*(C)
2. *restricted-roles*(I) $\equiv$ *restricted-roles*(C)
3. *For every role* R *on restricted-roles*(I)
   (a) *at-least*(R_I) $\geq$ *at-least*(R_C)
   (b) *at-most*(R_I) $\leq$ *at-most*(R_C)
   (c) *value-restriction*(R_I) $\Rightarrow$ *value-restriction*(R_C)
   (d) *fillers*(R_I) $\supseteq$ *fillers*(R_C)

For example, COMPUTER-SYSTEM123 bijectively instantiates COMPUTER-SYSTEM. It inherits all of COMPUTER-SYSTEM's primitives and restricted roles, and does not add any others (it just adds fillers to already restricted roles).

The distinction between *concrete* and *abstract* concepts is characterized in terms of bijective instantiation:

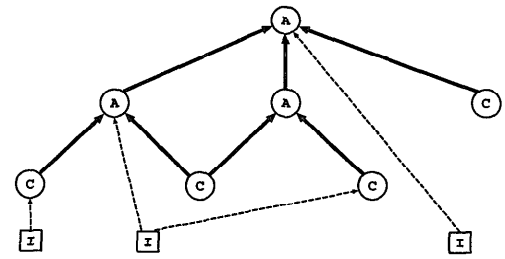

Figure 3: Abstract / Concrete Concepts with Individuals

Definition 4 *Concept* C *is* concrete *iff an individual which bijectively instantiates* C *might be both complete and sufficiently specific for the purposes of the intended application.*

In configuration, 486DX-33MHz would be designated concrete, whereas more general concepts such as CPU would be designated abstract:

Definition 5 *A concept is* abstract *iff it is not concrete.*

Like the distinction between individuals and concepts, the distinction between concrete and abstract concepts is clear in principle, but can be a knowledge engineering choice in practice. The "possibility" alluded to in Definition 4 is realized if the individual in question is also *finished* with sufficient role fillers that are in turn finished:

Definition 6 *Individual* I *is* finished *when*

1. I *bijectively instantiates a concrete concept, and*
2. *For every role* R *on restricted-roles*(I)
   (a) $|fillers(R_I)| \geq$ *at-least*(R_I)
   (b) *Every filler of* R_I *is finished*

Our language forbids cyclic descriptions, so Definition 6 is well-founded.

From the DL system's perspective, configuration consists of describing a finished individual computer system. Observe that COMPUTER-SYSTEM123 is not yet finished. To finish it, we could add suitable role fillers that are themselves finished, e.g., an individual operating system, so it instantiates a concrete subsumee of COMPUTER-SYSTEM (as noted, concrete concepts are not shown in Figure 2). With respect to configuration, a finished individual system is complete and specific such that it can be ordered from the manufacturer. Of course, it may still be updated, e.g., by adding fillers to a role, subject to the role's at-most restriction.

A well-formed terminology should respect certain criteria regarding abstract and concrete concepts. First, all leaf concepts should be concrete. For the purposes of this paper, all non-leaf concepts should be abstract. In configuration, where the very goal is to describe a finished system with finished components, etc., every concrete concept should admit a finished instantiation as in Definition 6. It may also be appropriate to require that all concrete concepts (leaves), are mutually disjoint.

Assumptions and Goals for Recognition

Our predictive concept recognition methodology will make two essential assumptions: CTA and a provisional assumption of monotonic updates. We now define CTA precisely:

Definition 7 *Under the* closed terminology assumption, *it is assumed that no concepts will be added to the terminology during problem solving (by the application) and that every individual will ultimately be finished according to Definition 6.*

CTA further implies that all relevant roles of concepts are explicitly restricted, so roles not explicitly restricted by a concept are assumed to be unsatisfiable. CTA is essential for ruling out concepts that an individual cannot instantiate, and thus drawing conclusions from the remainder. CTA applies only at the level of concepts; we neither assume that the set of individuals is closed, nor do we perform closed world reasoning over individuals, as in (Donini *et al.* 1992).

By Definition 6, once an individual is finished, it will bijectively instantiate at least one concrete concept. Such concepts are referred to as *ultimate* concepts:

Definition 8 *Given that an individual, when finished, will bijectively instantiate one or more concrete concepts, those concepts are its* ultimate *concepts.*

While an ultimate concept restricts every property of an individual, it may do so at an abstract level, in the same sense that COMPUTER-SYSTEM requires a processor, but only constrains it to be some CPU. Furthermore, when a role of an ultimate concept does not have a positive at-least restriction, the individual may decline to fill that role by specifying an at-most restriction of zero, e.g., (at-most 0 SECONDARY-STORAGE) for a diskless workstation.

We also assume, provisionally, that an individual will be updated monotonically:

Definition 9 *Under the* monotonic update assumption, *it is assumed that all updates of an individual will entail adding base concepts and/or adding role restrictions (introducing a restriction on a role or tightening an existing one).*

This assumption is essential for conclusions based on the individual's current description. Still, if it proves unfounded, we can back out of it gracefully.

Our first major goal for predictive recognition is to track the potential instantiation relationships between an individual and concepts in the terminology while the individual is being specified. Given our assumptions, a concept can be categorized as either necessary, optional, or impossible, relative to an individual's current description:

Definition 10 *Concept* C *is* necessary *with respect to individual* I *iff* I *instantiates* C.

Definition 11 *Concept* C *is* optional *with respect to individual* I *iff* I *does not instantiate* C *but can be monotonically updated to do so.*

A concept is *possible* if it is either necessary or optional. Otherwise it is impossible:

Definition 12 *Concept* C *is* impossible *with respect to individual* I *iff* I *neither instantiates* C *nor can be monotonically updated to do so.*

Another major goal is to exploit the necessary/optional/impossible trichotomy to provide information and to derive additional constraints on an individual from the commonality among its optional concepts.

Closed Terminology Consistency

During configuration, we want to know which concepts are CTA-consistent with a partially described individual, and which are not, so we can distinguish between its possible and impossible concepts. CTA is vital to our methodology: CTA holds that an individual must eventually be finished in accord with Definition 6. Thus, an individual can be monotonically updated to instantiate concept C under CTA iff it can be monotonically updated to bijectively instantiate an explicitly defined concept (perhaps C itself) when it does so. We are also concerned with CTA-consistency between concepts, both in this section, to help decide CTA-consistency of value restrictions on roles, and later, to help speed recognition. We decide CTA-consistency via the mutually recursive definitions that follow. For a pair of concepts, we must consider two cases. First, a pair of concepts are directly consistent under CTA whenever a bijective instantiation of one can also instantiate the other. To capture this case, we introduce a *direct consistency* inference from concept *C1* to concept *C2*, written $C1 \mapsto C2$:

Definition 13 C1 $\mapsto$ C2 *iff*

1. *primitives(*C1*)* $\subseteq$ *primitives(*C2*)*
2. *No primitive of* C1 *is disjoint from any primitive of* C2
3. *restricted-roles(*C1*)* $\subseteq$ *restricted-roles(*C2*)*
4. *For every role* R *on restricted-roles(*C1*),* R_{C1} *and* R_{C2} *are CTA-consistent*

Intuitively, $C1 \mapsto C2$ demonstrates that the primitives and role restrictions of *C2* admit a bijective instantiation of *C2* which also instantiates *C1*. For example, IBM-PROCESSOR-DEVICE $\mapsto$ COMPUTER-SYSTEM; even though neither one subsumes the other, some individual computer systems may have IBM-CPU processors. Direct CTA-consistency between concepts can occur in either direction:

Definition 14 *Concepts* C1 *and* C2 *are* directly CTA-consistent *iff* C1 $\mapsto$ C2 *or* C2 $\mapsto$ C1.

We must also define CTA-consistency for roles:

Definition 15 *Roles* R_X *and* R_Y *are* CTA-consistent *iff*

1. *Their cardinality restrictions intersect*
2. *If at-least(*R_X*)* > 0 *or at-least(*R_Y*)* > 0*, then value-restriction(*R_X*) and value-restriction(*R_Y*) are CTA-consistent*
3. $|$ *fillers(*R_X*)* $\cup$ *fillers(*R_Y*)* $| \leq$ *minimum(at-most(*R_X*), at-most(*R_Y*))*
4. *Every filler of* R_X *is CTA-consistent with value-restriction(*R_Y*), and every filler of* R_Y *is CTA-consistent with value-restriction(*R_X*)*

When R_X and R_Y are CTA-consistent, their restrictions can be simultaneously satisfied under CTA. Clearly, the processor and operating-system roles of UNIPROCESSOR-SYSTEM and UNIX-RISC-SYSTEM are CTA-consistent. In addition, both concepts inherit their primitives and their remaining role restrictions from a common source, COMPUTER-SYSTEM, so UNIPROCESSOR-SYSTEM $\mapsto$ UNIX-RISC-SYSTEM (and *vice versa*). We conclude that they are CTA-consistent.

There is a second, indirect case of CTA-consistency between concepts. Even if two concepts are not directly consistent, their consistency may still be explicitly sanctioned by a third concept consistent with both. This situation can arise when each concept has a primitive or restricted role that the other lacks. For example, IBM-CPU lacks a technology role and RISC-CPU lacks a vendor role. Only from the existence of IBM-RISC-CPU can we conclude that they are CTA-consistent. As a result, we have:

Definition 16 *Concepts* C1 *and* C2 *are indirectly* CTA-consistent *iff there exists an explicitly defined concept* C3 *such that* C1 $\mapsto$ C3, C2 $\mapsto$ C3, *and for every role* R *restricted by both* C1 *and* C2, R_{C1} *and* R_{C2} *are CTA-consistent.*

Definition 16 ensures that *C1* and *C2* are separately consistent with *C3*, and if restrictions on *C1* and *C2* interact, they do so in a mutually consistent way. Hence, an instance of *C3* can simultaneously instantiate *C1* and *C2*. Combining the direct and indirect cases, CTA-consistency identifies pairs of concepts that can be instantiated simultaneously:

Definition 17 *Concepts* C1 *and* C2 *are* CTA-consistent *iff they are directly or indirectly CTA-consistent.*

This definition is justified as follows:

Theorem 1 *Under CTA, the extensions of concepts* C1 *and* C2 *intersect iff they are CTA-consistent.*

Now we examine CTA-consistency between an individual and a concept. We again distinguish between direct and indirect cases. A *direct consistency* inference from individual *I* to concept *C*, written $I \mapsto C$, essentially duplicates Definition 13:

Definition 18 I $\mapsto$ C *iff*

1. *primitives*(I) $\subseteq$ *primitives*(C)
2. *No primitive of* I *is disjoint from any primitive of* C
3. *restricted-roles*(I) $\subseteq$ *restricted-roles*(C)
4. *For every role* R *on restricted-roles*(I), R_I *and* R_C *are CTA-consistent*

Direct consistency follows immediately:

Definition 19 *Individual* I *and concept* C *are* directly CTA-consistent *iff* I $\mapsto$ C.

Intuitively, direct consistency of an individual with a concept establishes that the individual can bijectively instantiate the concept, perhaps after monotonic updates. Consider:

(create-individual COMPUTER-SYSTEM45
(and COMPUTER-SYSTEM (all processor RISC-CPU)))

Although COMPUTER-SYSTEM45 is directly consistent with both RISC-MULTIPROCESSOR-SYSTEM and DUAL-IBM-PROCESSOR-SYSTEM, it instantiates neither: regarding DUAL-IBM-PROCESSOR-SYSTEM, its processor role is neither restricted to IBM-CPU fillers nor a cardinality of exactly 2, and regarding RISC-MULTIPROCESSOR-SYSTEM, the cardinality of its processor role is not known to be at least 2.

In the context of a terminology, the fact that an individual is directly consistent with some concept may imply consistency with other concepts. Clearly, individual *I* is indirectly CTA-consistent with concept *C* if there exists a concept *C'* such that $I \mapsto C'$ and $C' \Rightarrow C$, i.e., if *I* can be monotonically updated to instantiate *C'* then it can be monotonically updated to instantiate any concept that subsumes *C'*. For example, COMPUTER-SYSTEM45 is directly consistent with DUAL-IBM-PROCESSOR-SYSTEM but not with IBM-PROCESSOR-DEVICE. However, COMPUTER-SYSTEM45 is indirectly consistent with IBM-PROCESSOR-DEVICE because the latter subsumes DUAL-IBM-PROCESSOR-SYSTEM. Not all indirectly consistent concepts can be identified via subsumption (but see the next section). Consider:

(create-individual COMPUTER-SYSTEM67
(and COMPUTER-SYSTEM (exactly 1 processor)))

We can not make a direct consistency inference from COMPUTER-SYSTEM67 to IBM-PROCESSOR-DEVICE, e.g., because primitives(COMPUTER-SYSTEM67) $\not\subseteq$ primitives(IBM-PROCESSOR-DEVICE), or to its only subsumee, DUAL-IBM-PROCESSOR-SYSTEM, because the cardinality restrictions on their processor roles don't intersect. Still, we can monotonically update COMPUTER-SYSTEM67 to directly instantiate UNIPROCESSOR-SYSTEM such that it also instantiates IBM-PROCESSOR-DEVICE, by adding the restriction (all processor IBM-CPU). To generalize, if individual *I* can potentially instantiate concept *C'* so it also instantiates concept *C*, then *I* is *indirectly consistent* with *C* via *C'*:

Definition 20 *Individual* I *and concept* C *are* indirectly CTA-consistent *iff there exists a concept* C' *such that* I $\mapsto$ C', C $\mapsto$ C', *and for every role* R *restricted by both* I *and* C, R_I *and* R_C *are CTA-consistent.*

Either directly or indirectly, CTA-consistency identifies the concepts an individual might instantiate once it is finished:

Definition 21 *Individual* I *and concept* C *are* CTA-consistent *iff they are directly or indirectly CTA-consistent.*

We say that *I potentially instantiates C* when they are CTA-consistent. The correctness of CTA-consistency is established by the following:

Theorem 2 *Under CTA, individual* I *can be monotonically updated to instantiate concept* C *iff* I *and* C *are CTA-consistent.*

Terminology Augmentation

Testing indirect consistency straight from Definition 20 would mean run-time search for a concept *C'* through which indirection is licensed. However, notice that when individual *I* instantiates concepts *C* and *C'* simultaneously, it must instantiate their conjunction. This insight will allow us to quickly identify all cases of indirect consistency by traversing explicit subsumption links in the taxonomy. We can augment the terminology with concepts for K-REP's internal use as follows (these auxiliary concepts are not considered part of the domain model, so they do not violate CTA):

Definition 22 *A terminology is* augmented *iff for all concepts* C1 *and* C2 *such that* C1 $\mapsto$ C2, *there exists an explicitly defined concept* C3 *such that* C3 $\equiv$ C1 $\wedge$ C2.

When *C1* subsumes or is subsumed by *C2*, we do nothing because the subsumee *is* their conjunction. As an example, recalling that UNIPROCESSOR-SYSTEM (*C1*) $\mapsto$ UNIX-RISC-SYSTEM (*C2*), we would add their conjunction (*C3*), defined:

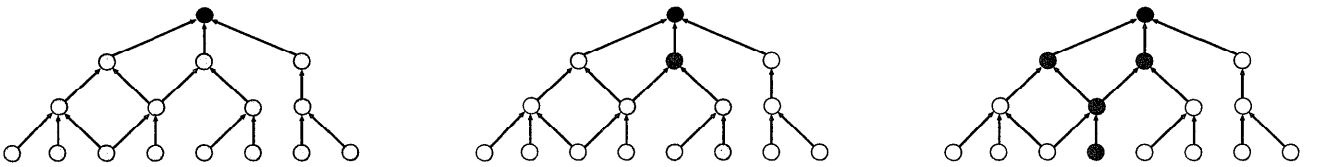

Figure 4: Initial, Intermediate, and Final Recognition States

(and COMPUTER-SYSTEM
(the processor RISC-CPU)
(the operating-system UNIX))

After augmentation, two concepts are CTA-consistent just in case they have a common subsumee (subsumption is reflexive). Consequently, we can reduce the indirect aspect of consistency testing to subsumption checking:

Definition 23 *Individual* I *is* indirectly CTA-consistent *with concept* C *in an augmented terminology iff there exists a concept* C′ *such that* $I \mapsto C'$ *and* $C' \Rightarrow C$.

With an augmented terminology, Definition 23 is correct:

Theorem 3 *Under CTA and with an augmented terminology, individual* I *can be monotonically updated to instantiate concept* C *iff* I *and* C *are CTA-consistent using Definition 23 for indirect consistency instead of Definition 20.*

Under CTA and our monotonic update assumption, an individual is always directly consistent with its ultimate concept(s), which are concrete according to Definition 6. Thus, we can improve on Definition 22 by limiting augmentation to cases where *C2* is a concrete concept. We need only augment a terminology once, after its development concludes and before problem solving begins. The possible drawback of augmentation is degraded performance due to proliferation of system-defined concepts. The number of system-defined concepts in an augmented terminology should be manageable in practice, assuming sufficiently distinct primitiveness and role restrictions among user-defined concepts.

Recognition via Terminology Partitioning

Given an individual *I*, a KB, and our assumptions, predictive recognition determines each concept's status, or *modality*, with respect to *I*. Recall that concept *C* is *necessary* if *I* instantiates *C*, else *optional* if *I* can be monotonically updated to instantiate *C*, else *impossible*. These definitions implicitly partition the taxonomy into regions as in Figure 4, where necessary, optional, and impossible concepts are black, gray, and white, respectively. Such a partition constitutes the *recognition state* of *I*. As an individual is incrementally updated, we can track its recognition state. Initially, all concepts are optional except the vacuous root concept, THING, which is trivially necessary. The monotonic update assumption implies that each update will change the modality of zero or more optional concepts to necessary or impossible, so the necessary and impossible regions expand as the individual is updated, further confining the optional region. Finally, (at least) one concrete concept and its ancestors are necessary; the remainder are impossible. (It is straightforward to test if an update is nonmonotonic. Then, we may need to re-expand the optional region. For brevity, we only elaborate here on monotonic updates.) Our objective is to efficiently bound the portion of the taxonomy containing optional concepts, thereby limiting the number of concepts that must be directly compared with the individual. We exploit the terminology's subsumption-based organization in two ways. First, the current partition is used to compute the next one. Second, the consequences of comparing an individual with a concept are propagated to other concepts, so those other concepts need not be directly compared with the individual. To these ends, we distinguish three sets of concepts: (1) the most specific necessary concepts (MSNs), (2) the most general optional concepts (MGOs), and (3) the most specific optional concepts (MSOs). The frontiers of the optional region are maintained by the MGOs and the MSOs. Those sets need not be disjoint. The MSNs serve to speed computation of the MSOs. Initially, the MSN set contains the root concept THING, the MGO set contains THING's children, and the MSO set contains the leaf concepts. When an individual is updated, our algorithm operates in three successive phases to update the MSNs, MSOs, and MGOs, respectively. The first phase discovers all newly necessary concepts, the second all newly impossible ones. The third simply prepares the new MGOs for the MSN phase of the next round of incremental recognition. Due to limited space, readers are referred to (Weida 1996) for further details of the algorithm. As an example, an initial configuration description might be:

(create-individual COMPUTER-SYSTEM89
(and COMPUTER-SYSTEM
(at-least 2 processor)
(at-least 1 secondary-storage)))

For this example, suppose that IBM-PROCESSOR-DEVICE is *omitted* from the terminology of Figure 2. Then, the necessary concepts are THING, SYSTEM and COMPUTER-SYSTEM. The optional ones are DUALPROCESSOR-SYSTEM, DUAL-IBM-PROCESSOR-SYSTEM, UNIX-RISC-SYSTEM, and RISC-MULTIPROCESSOR-SYSTEM. The remainder are impossible. This recognition state is captured as:

MSNs = {COMPUTER-SYSTEM}
MGOs = {DUALPROCESSOR-SYSTEM, UNIX-RISC-SYSTEM, RISC-MULTIPROCESSOR-SYSTEM}
MSOs = {DUAL-IBM-PROCESSOR-SYSTEM, UNIX-RISC-SYSTEM, RISC-MULTIPROCESSOR-SYSTEM}

This recognition state indicates precisely which concepts COMPUTER-SYSTEM89 might eventually instantiate. Notice that UNIPROCESSOR-SYSTEM and DISKLESS-SYSTEM are ruled out, as are many other concepts, e.g., COMPANY. If the user now specifies exactly 4 processors, the ensuing recognition state will also rule out DUALPROCESSOR-SYSTEM and DUAL-IBM-PROCESSOR-SYSTEM:

MSNs = {COMPUTER-SYSTEM}
MGOs = {UNIX-RISC-SYSTEM, RISC-MULTIPROCESSOR-SYSTEM}
MSOs = {UNIX-RISC-SYSTEM, RISC-MULTIPROCESSOR-SYSTEM}

Notice how the MGOs succinctly capture the most general choices available in the current state. A user interface can guide the user to choose a subset of the MGOs that describe the desired system.

We have experimented with our partitioning algorithm using a small but realistic configuration terminology. It was developed by an independent group and engineered for a configuration system that uses a K-REP KB. The terminology has 501 concepts, including 270 leaves. Although space limitations preclude much detail, performance is easily fast enough for interactive applications. In one experiment, a program conducts predictive recognition while incrementally instantiating each leaf concept in turn. The recognition algorithm spends 86 percent of its time in the MSO phase, 13 percent in the MSN phase, and only 1 percent in the MGO phase. The "bushiness" of the concept taxonomy, plus the inherently bottom-up nature of the MSO search, explains why most time is devoted to finding MSO concepts.

Constraint Derivation

We now take advantage of predictive recognition to derive additional constraints (primitives and/or role restrictions) on an individual. Suppose the terminology is partitioned according to the current description of some individual, *I*. If *I* does not bijectively instantiate an MSN concept which is concrete, then by CTA, *I* will ultimately instantiate some optional concept. Therefore, the commonality among MGO concepts constitutes a set of implicit constraints imposed on *I* by the closed terminology. Cohen, et al., showed how to compute the *least common subsumer* (LCS) of a set of concepts (Cohen, Borgida, & Hirsh 1992). LCS(MGOs) is a concept representing just their commonality (we never use the LCS concept for any other purpose, so it is not permanently installed in the terminology). Any constraint on LCS(MGOs) not reflected in *I* should be added to *I*. The basic strategy is, until arriving at a fixed point, repeatedly:

1. Compute LCS(MGOs)
2. If LCS(MGOs) implies further constraints on *I*
   (a) Fold LCS(MGOs) into *I*'s description, and
   (b) Update the recognition state of *I*

Since each iteration of this process can add but never remove restrictions, it must eventually terminate. Thus, K-REP may be able to infer constraints on *I* after it is first created and whenever it is updated. Similar reasoning applies recursively to fillers of *I*'s roles.

Recall COMPUTER-SYSTEM89's second recognition state in the previous section. Since the only MSN concept (i.e., COMPUTER-SYSTEM) is not concrete, COMPUTER-SYSTEM89 will eventually instantiate at least one MGO concept. LCS({UNIX-RISC-SYSTEM, RISC-MULTIPROCESSOR-SYSTEM}) ≡ (and COMPUTER-SYSTEM (all processor RISC-CPU)). Hence, K-REP discovers that COMPUTER-SYSTEM89's processors must be of type RISC-CPU. This conclusion can only come from CTA reasoning. Thus, future choices regarding the processors will be suitably constrained.

The preceding was simplified for clarity; we can often do better. For details and further examples, see (Weida 1996).

Open vs. Closed Terminologies

CTA limits the concepts that an individual can instantiate, perhaps after monotonic update, since ultimately it must bijectively instantiate an explicitly defined concrete concept. That requirement also imposes constraints on an individual's description e.g., the combination of roles it may restrict and the particulars of those role restrictions. Consider the following individual, which instantiates IBM-RISC-CPU, even though IBM-RISC-CPU isn't stated as a base concept:

(create-individual IBM-RISC-CPU86
  (and CPU (fills vendor IBM) (the technology RISC)))

Under OTA, IBM-RISC-CPU86 is in the extension of both IBM-CPU and RISC-CPU, whether or not IBM-RISC-CPU is explicitly defined. Now suppose for the remainder of this paragraph that our terminology *does not* contain IBM-RISC-CPU. Then, under CTA, IBM-RISC-CPU86 is not allowable, and indeed no individual CPU can be both an IBM-CPU and a RISC-CPU. (Since restricted-roles(IBM-CPU) is neither a subset nor a superset of restricted-roles(RISC-CPU), no bijective instantiation of one can also instantiate the other. Also, they would have no common subsumee.) Thus, CTA restricts the extensions of IBM-CPU and RISC-CPU. In general, a concept's extension under CTA is a subset of its extension under OTA. Furthermore, all CTA-consistency relationships must hold under OTA, but the converse is not true. For example, IBM-PROCESSOR-DEVICE and UNIX-RISC-SYSTEM are OTA-consistent. Without the presence of IBM-RISC-CPU, they would not be CTA-consistent, because their processor roles would not have CTA-consistent value restrictions. For further examples and discussion, see (Weida 1996).

Finally, constraint derivation via the LCS inference depends crucially on CTA. Without CTA, the "optional" concepts might include any number of concepts not explicitly defined in the terminology. Consequently, taking the LCS of optional *known* concepts in order to derive constraints on an individual would be entirely unjustified.

Related and Future Work

Previous work shows that DL is well suited to configuration problems. The BEACON configurator based on KNET reached the advanced prototype stage (Searls & Norton 1990). In (Owsnicki-Klewe 1988), the entire configuration problem was cast as a matter of maintaining internal KB consistency, i.e., logical contradictions should follow from all invalid configurations but no valid ones. This goal is more ambitious than ours or that of (Wright *et al.* 1993). However, (Owsnicki-Klewe 1988) considered only "an (admittedly limited) example." PROSE (Wright *et al.* 1993) is a successfully deployed configurator featuring product KBs written in CLASSIC (Borgida *et al.* 1989). Like BEACON and PROSE, our work positions the DL system as a product knowledge reasoner — a key module in a larger configurator architecture, i.e., the DL system maintains internal KB consistency during configuration but relies on configuration-specific modules for further reasoning. We share the PROSE developers' views that (1) DL fosters a reasonable, even natural approach to product knowledge representation, and (2) knowledge engineering efforts benefit from enforcing

internal KB consistency. However, no previous work has pursued CTA reasoning in DL.

This work draws on a predictive recognition methodology for constraint networks representing rich temporal patterns, e.g., plans (Weida & Litman 1992; 1994; Weida 1996). Our terminology partitioning algorithm bears some resemblance to the candidate-elimination algorithm of (Mitchell 1982), however candidate-elimination uses explicit negative examples to exclude concepts, while we derive the concepts to be excluded using CTA. Also, candidate-elimination operates on sets of different positive and negative examples, while we are concerned with successive descriptions of the same (positive) instance.

Some inferences described above are syntactic in nature, thus dependent on the DL language under consideration. Our framework appears extensible for the remaining DL operators used in PROSE (Wright *et al.* 1993), among others. We are not yet concerned with disjunction and negation operators because K-REP does not support them, but it is admittedly unclear how efficiently they can be incorporated in the present framework. Importantly, CLASSIC has achieved great success in the configuration arena without supporting either disjunction or negation (Wright *et al.* 1993).

We are investigating a formal semantics for CTA reasoning in DL based on circumscription. There is an interesting analogy with Kautz's use of circumscription to close an event hierarchy for deductive plan recognition (Kautz 1991): Kautz's hierarchy is composed of events related by steps; our concept taxonomy is composed of concepts related by roles.

Conclusion

This paper presents a predictive concept recognition methodology for DL which demonstrates, for the first time, the value of CTA reasoning. These ideas apply to tasks such as system configuration where all relevant concepts are known during problem solving. No previous work has differentiated between the knowledge engineering and problem solving phases of terminology usage in DL. Our configuration application led us to introduce the distinction between abstract and concrete concepts to DL, so we can judge when an individual's description may be finished. It also led us to devise inferences for characterizing the completeness and specificity of incremental instantiation, i.e., bijective and finished instantiation. We take advantage of the closed terminology and its subsumption-based organization in several ways. As an individual system is incrementally specified, we efficiently track the types of systems that may result via CTA-consistency inferences, which are stronger than OTA-consistency inferences. We incrementally partition the terminology by categorizing concepts as necessary, optional, or impossible with respect to the system's current specification, based respectively on instantiation, potential instantiation, or lack thereof. This information is inherently useful for both user and configuration engine. We also exploit the current partition through novel use of the LCS inference to derive constraints on the system implied by the KB under CTA. Similar reasoning applies recursively to the system's components. Finally, we take advantage of derived constraints to inform the user and configuration engine, and to appropriately guide future choices. This work is implemented in K-REP. Although we focused on configuration, our methodology is domain-independent.

Acknowledgements: I am extremely grateful to Diane Litman for thoughtful thesis advice, Eric Mays for valuable insights and support, Bonnie Webber, Mukesh Dalal, Steffen Fohn, Brian White, and anonymous reviewers for helpful comments on earlier drafts, and Bernhard Nebel and Werner Nutt for useful discussions.

References

Borgida, A.; Brachman, R. J.; McGuinness, D. L.; and Resnick, L. A. 1989. Classic: A structural data model for objects. In *Proceedings of 1989 ACM SIGMOD*, 58–67.

Cohen, W.; Borgida, A.; and Hirsh, H. 1992. Computing least common subsumers in description logics. In *Proceedings of AAAI-92*, 754–760.

Donini, F.; Lenzerini, M.; Nardi, D.; Schaerf, A.; and Nutt, W. 1992. Adding epistemic operators to concept languages. In *Proceedings of KR'92*, 342–354.

Kautz, H. A. 1991. A formal theory of plan recognition and its implementation. In Allen, J.; Kautz, H.; Pelavin, R.; and Tenenberg, J., eds., *Reasoning About Plans*. Morgan Kaufmann. 69–125.

Kramer, B. 1991. Knowledge-based configuration of computer systems using hierarchical partial choice. In *International Conference on Tools for AI*, 368–375.

Mays, E.; Dionne, R.; and Weida, R. 1991. K-rep system overview. *SIGART Bulletin* 2(3):93–97.

Mitchell, T. M. 1982. Generalization as search. *Artificial Intelligence* 18:203–226.

Owsnicki-Klewe, B. 1988. Configuration as a consistency maintenance task. In *Proceedings of GWAI-88*, 77–87.

Searls, D. B., and Norton, L. M. 1990. Logic-based configuration with a semantic network. *Journal of Logic Programming* 8:53–73.

Weida, R., and Litman, D. 1992. Terminological reasoning with constraint networks and an application to plan recognition. In *Proceedings of KR'92*, 282–293.

Weida, R., and Litman, D. 1994. Subsumption and recognition of heterogeneous constraint networks. In *Proceedings of CAIA-94*, 381–388.

Weida, R. A. 1996. *Closed Terminologies and Temporal Reasoning in Description Logic for Concept and Plan Recognition*. Ph.D. Dissertation, Department of Computer Science, Columbia University.

Woods, W. A., and Schmolze, J. G. 1992. The kl-one family. *Computers and Mathematics with Applications* 74(2-5).

Wright, J. R.; Weixelbaum, E. S.; Brown, K. E.; Vesonder, G. T.; Palmer, S. R.; Berman, J. I.; and Moore, H. H. 1993. A knowledge-based configurator that supports sales, engineering, and manufacturing at at&t network systems. In *Proceedings of the Fifth IAAI*.