Reasoning with Black Boxes: Handling Test Concepts in CLASSIC

Alex Borgida*  
Dept. of Computer Science  
Rutgers University

Charles L. Isbell  
Artificial Intelligence Laboratory  
Massachusetts Institute of Technology

Deborah L. McGuinness  
Artificial Intelligence Principles Research  
AT&T Labs

1 Motivation

Description Logics (DLs) are distinguished by a number of characteristics, including decidable reasoning algorithms for tasks such as subsumption and consistency checking. The desire for terminating, even efficient, reasoning procedures has led to two alternative approaches to DL system design: complete reasoning for expressively limited languages, and incomplete reasoning for richer languages. The obvious problem with the first approach ([8]) is that in many applications one would want to express concepts that cannot be captured. The problem with the second approach is that the language implementer chooses the subset of inferences to be performed once and for all, and must then characterize this to system users. It seems unlikely that the designer's choices will be appropriate for all applications.

There are forces driving extensions of DLs. For example, many application designers are not content with simple individuals related by roles; they need to reason with complicated objects related in complex ways. There have been proposals for extending DLs with plans (whose instances are sequences of action individuals) [7], temporal values [13], mathematical objects (real numbers with inequality, tuples, etc.) [2; 9] and other data types. Furthermore, because of user demand, many practical DL systems (e.g., LOOM, CLASSIC, BACK) have introduced concept constructors for describing ranges of integers. We observe however that these languages do not have concept constructors for dealing with strings, dates, and other kinds of objects. And any decision on which of these to support will also be quite arbitrary.

For these reasons, it seems useful to provide a facility to add arbitrary new concept constructors to a language, and to be able to extend the reasoning of the system to cover these new constructors. This is especially true for CLASSIC, which takes a relatively extreme position on limiting the set of concept constructors provided, disallowing disjunction and existential quantifiers, for example.

2 Test concepts

There are two aspects of reasoning to consider for concepts: extensional aspects, dealing with how individuals are related to a concept, and intensional aspects, dealing with how concepts relate to each other.

2.1 Individual Reasoning

Concerning extensional aspects, CLASSIC originally took inspiration from the work of Abrial [1], and allowed concepts to be specified by a user-defined procedure that tests for the membership of an individual in the concept. For example, the concept (test someTallFriend) recognizes those individuals b for which the following Lisp function someTallFriend returns true1:

```
(defun someTallFriend (ind)
  (for-each-filler afriend (ind friend)
    (for-each-filler ht (afriend height)
      (when (> ht 6) (return t)))
    (return nil)))
```

Like all test functions, someTallFriend takes an individual as its first argument.

In an application where the notion of having some tall friend is useful, it would likely be the case that other test concepts such as someShortFriend or someOldEnemy would also be useful. It therefore makes sense to generalize test functions to take additional arguments that are specified as part of the concept definition. In the above example, we would therefore make the role name and class also be parameters of the test function, defining the function some as (defun some (ind role concept) ...); the above concept would then be specified as (test someTall). This is very close syntactically to the way one would specify this in a DL (like BACK) that has a built-in constructor for this kind of concept: (someTall).

1 For simplicity, we ignore several complexities having to do with individual processing, such as the need for a 3-valued, rather than a 2-valued response, and the need for setting dependency links.
2.2 Concept Reasoning
Unfortunately, because one cannot reason about the actions of Lisp functions, CLASSIC cannot deduce that to check whether ((test some friend Tall) subsumes (test some friend VeryTall)), it is sufficient to perform a subsumption test between the concept arguments Tall and VeryTall. Similarly, CLASSIC cannot infer that (and (test some friend Tall) (test some friend Short)) is incoherent, if Tall and Short can be deduced to be mutually exclusive. The best that CLASSIC can do for subsumption reasoning is check for the special cases of textual identity.

Moreover, functions with additional arguments like some must usually include a check on the correctness of their arguments. For example, (test some age 12) should generate an error if age is not a declared role or 12 is not the name of a class. Unfortunately, CLASSIC has no knowledge about the types of the arguments expected by the Lisp procedure some, so this error checking must be performed by the some function itself each time it is invoked on some individual. It would be much more efficient if this type-check could be performed just at "concept compile time", whenever a test-defined concept using the function some is encountered.

3 Adding Subsumption for Test Concepts
Based on our previous research [3; 5] on the ProtoDL extensible DL system, we have introduced a mechanism in CLASSIC to allow it to reason about test concepts. Our approach (in contrast to [2]) is intended to work with so-called normalize-compare implementations, where the aim is to find a normal form for concepts which explicates implicit facts, detects inconsistencies and eliminates redundancies. As a result, comparing two concepts for subsumption is a simple matter of comparing "structurally similar" elements. Based on our earlier analysis, it appears that this can be accomplished through the use of a small set of functions:

- NormalizeTerm[K](syntacticArgument) takes a concept built with concept constructor K, and computes its normal form. In the process, it will check syntactic and semantic restrictions on the valid arguments to the constructor, as well as building any necessary data structures. For example, from "some friend Tall", the function would return a data structure corresponding to the term some(friend, Tall). (Here, the concept constructor is used as a functor for the term, and the arguments are themselves normalized CLASSIC objects, denoted by a leading @.)

- StructuralSubsumes?[K](Kterm1, Kterm2) compares two normalized terms built with the same concept constructor K, and determines whether one is more general than the other.

- Subsumes?[K](Kterm, normalizedConcept) is a general subsumption test, for the cases where structural subsumption is not enough. It determines whether a normalized term built with concept constructor K is implied by the general normalized concept (which represents a conjunction of terms built with other constructors).

- Conjoin[K](Kterm, normalizedConcept) takes a normalized term built with constructor K, and conjoins it to an already normalized concept, which is essentially the conjunction of built-in CLASSIC concepts and other test-defined concepts previously encountered.

We have found that Conjoin[K] can often be composed from a group of more specialized subroutines: ConjoinToSame[K], which computes the conjunction of two terms built with constructor K; ConjoinToM[K], which computes the conjunction of a K-term and an M-term; Coherent[K], which detects when a K-term denotes the empty set; InconsistentWith[K], which detects when a K-term conjoined with other parts of the a concept denotes the empty set; and Implies[K], which returns non-test terms that are implied by the conjunction of a K-term and a concept.

Empirically, this decomposition serves two purposes. First, it is often easier to gather the consequences of conjunction in this piecewise manner. Second, this allows the designer to make explicit decisions about the tradeoff between expressivity and efficiency. For a more complete discussion of these issues see [3; 5; 4], while for details of function definitions see [12].

3.1 String reasoning: an example
To illustrate the use of the extension mechanisms, in particular the normalization and subsumption reasoning of test concepts, we sketch three extensions to CLASSIC, intended to support reasoning about strings.

Ranges of string values
We wish to denote ranges of strings, similar in spirit to Pascal integer ranges like 1..40. To this end, we introduce the test-concept constructor stringRange. The syntax is stringRange attribute\(^2\) string1 string2, as in (test stringRange name "Alice" "Beth").

Normalization guarantees that the syntax is respected: there must be exactly three arguments, the first being a valid attribute name, the second and third being strings. In addition, normalization verifies that the value restriction is coherent (using Coherent[stringRange]), by checking whether the first string argument is lexicographically before the second. The normalized string stored therefore corresponds to stringRange(roleId, string1, string2), where string1 \(\leq\) string2.

Subsumption is easy: stringRange(attribute1, s1, s2) is subsumed by stringRange(attribute2, z1, z2) if

\(^2\)An attribute is a role with exactly one filler, and is chosen here to simplify the details.
and only if attrib1 = attrib2, and s1 ≤ s2 ≥ z2. (This assumes that there are no attribute hierarchies.)

ConjoinToSame[stringRange] has no effects on arguments stringRange([attrib1, s1, s2] and stringRange([attrib2, z1, z2]) if attrib1 ≠ attrib2; however, if attrib1 = attrib2, it returns stringRange([attrib1, max(s1, z1), min(s2, z2)]) as the normal form of conjunction. The result is also checked for coherence.

Finally, Implies[stringRange] needs to recognize that (test stringRange r s z) implies (all r STRING), and in the special case when s = z, it is equivalent to (all r (one-of s)).

Substrings
Suppose we also wish to have concepts denote objects that have some attribute that contains a specific substring. For this we will use the test concept constructor substring, as in (test substring sender "mit.edu").

Structural subsumption is sufficient in this case, too, with (test substring p s) subsumed by (test substring q z) iff p = q and z is itself a substring of s.

Normalization verifies the syntax and the type of the arguments, and constructs a term with the attribute and the string as arguments.

In the case of conjunctions, such as

(and (test substring header "edu")
 (test substring header "eecs")
)

we could think of combining them in some way, but there seems to be no advantage to this. Therefore ConjoinToSame[substring] does nothing, and the term stored for substring is a list of pairs. Conjunction with stringRange also yields no inferences.

As before, the presence of a substring test-concept involving a role r implies (all r STRING), and in fact (test substring r "") adds no additional information, so it can be eliminated.

Regular Expressions as string classes
Regular expressions are natural descriptions for sets of strings, so we can consider adding a test-concept constructor allStringsIn, as in (test allStringsIn mailAddr "*.edu"). Without describing a precise syntax for regular expressions (we can assume one like that used by UNIX's grep), we outline a high-level view of what it would take to implement this new constructor in the presence of the previous ones.

In general, reasoning with regular expressions (REs) is known to be impractical, so one converts each RE to a finite automaton (FA), which acts as the normal form; however, we keep the RE around for printing purposes. The normalized form of an allStringsIn test concept is stored as a triple (attribute, representation of FA, string of RE).

Structural subsumption is just automaton containment, while ConjoinToSame[allStringsIn] is automaton conjunction. Both of these operations are well documented in the computer science literature. Note that, in addition, ConjoinToSame[allStringsIn] needs to combine the regular expressions of the conjuncts, obtaining from "RE1" and "RE2" the result "(RE1) ∨ (RE2)", in order to support appropriate printing of concepts.

On the other hand, allStringsIn also interacts with the other concept constructors involving strings. For example, (test substring r s) is equivalent to (test allStringsIn r "*s*"), so we could replace substring by the corresponding allStringsIn construct. However, finite automaton reasoning in general is much less efficient than the special case of substrings. Instead, Subsume[allStringsIn](([attrib, fa1, rel], c)) will check in c for a substring(attrib,s) construct on the same attribute, and then build an FA for "*s*" on the fly.

Also, in order to reason about interactions with stringRange we need to derive from the finite automaton associated with an allStringsIn the lexicographically smallest and greatest string accepted by it, if it exists; the corresponding stringRange expression needs to be conjoined to the original concept.

4 Explanation
Part of the CLASSIC philosophy is that the system should be able to explain the deductions that it performs [11]. In this case, we need to explain any normalization or subsumption deductions.

The general approach suggested in [10; 11] considers the inferences performed by the system as applications of declarative proof rules. For example, the conjunction and incoherence of stringRange are captured by the following rules:

\[ C \implies \text{stringRange}(r, s1, s2) \]
\[ C \implies \text{stringRange}(r, z1, z2) \]
\[ C \implies \text{stringRange}(r, v1, v2) \]
\[ v1 = \max(s1, z1), \quad v2 = \min(s2, z2) \]
\[ C \implies \text{stringRange}(r, s1, s2) \]
\[ C \implies \text{nothing} \]
\[ \{ s1 > s2 \} \]

These rules can be paraphrased as: If a description C is subsumed by a description whose fillers of attribute r must be in the range between s1 and s2, and C is also subsumed by a description whose fillers of the same attribute r must be in the range between z1 and z2, then C must have r fillers in the intersection of the s1 - s2 range and the z1 - z2 range. The second rule says that if description C is subsumed by a description whose r-fillers must be between s1 and s2, given that s1 is greater than s2 (thus it is impossible to be in that range), then C is inconsistent (it is subsumed by the special bottom concept called nothing); this is because the attribute can have no value in the appropriate range, yet it is required to have one.

3This can be read off a deterministic FA, if one eliminates "dead" states.
When running in explanation mode, data structures are supposed to keep track of the inference being performed. For test concepts, this means that we must identify the inference rules being applied and record them appropriately.

For example, consider conjoining the normalized terms stringRange(name, "Alex", "Charles") and stringRange(name, "Debbie", "Louise"). ConjoinToSame[stringRange] should detect the incoherence of the resulting conjunction, and signal it. The programmer of ConjoinToSame[stringRange] can choose to provide as much detail about the source of the incoherence as deemed necessary. For example, the procedure could record that stringRange(name, "Debbie", "Charles") is incoherent because "Debbie" comes after "Charles", and name, as an attribute, is required to have a filler. Note that in this case, in the interest of shorter explanations, the inference being explained corresponds to the rule

\[ C \Rightarrow \text{stringRange}(r, s1, s2) \]
\[ C \Rightarrow \text{stringRange}(r, s1, s2) \]
\[ C \Rightarrow \text{NOTHING} \quad \{s2 < z1 \text{ or } z2 < s1\} \]

For simple examples, a result such as this may seem obvious. On the other hand, results that are obtained from the conjunction of many test restrictions with complex interactions may be much more difficult to understand.

If explanations are not provided by the programmer, CLASSIC will generate a default explanation. Often these default explanations are sufficiently informative for a particular result. In this particular example, CLASSIC would be able to report that the two restrictions produced an inconsistency, but not why.

Below we outline how the test extensions communicate with the explanation facility.

- NormalizeTerm[K] is expected to return an explanation (i.e. an inference rule and arguments) that indicates the source of an error whenever one is signaled. For example, (test stringRange name "one" 2) might generate the inference (not-a-string 2). If no explanation is provided with the error signal, CLASSIC simply notes that some syntax error occurred within the expression, e.g., (syntax-error (stringRange name "one" 2)).

- StructuralSubsumes[K] and Subsumes[K] return two values. The first is a boolean value indicating whether a concept subsumes another. The second is an explanation of why the result occurred.

- Conjoin[K] (within its subroutines) is expected to return an explanation along with every inference. This includes not only explanations for inconsistencies or errors, but also explanations for any new inferences. When explanations are not provided, CLASSIC notes the expressions that are involved whenever possible.

There are additional issues that arise when using explanation and test extensions that are beyond the scope of this paper. In particular, the process of conjoining several test expressions requires maintaining a great deal of information about intermediate values. For more details see [12].

5 Discussion

There is a trend in the DL literature and a need in many applications to extend expressively limited description logics.

A general approach, investigated by Baader and Hanschke [2], considers the modular addition of theories in a system implemented using a tableau-like calculus. Although elegant, this approach does not allow new concept constructors for ordinary objects with roles, like a some constructor.

CLASSIC chose to provide procedural definitions of concepts, called test-concepts, but these were meant for individual reasoning, and were treated as "black boxes" otherwise. In this paper, we described a structured way to allow CLASSIC to reason with these black boxes, enabling it to determine relationships such as subsumption between test concepts. We have implemented our approach and have integrated it into CLASSIC. We have described here implementations for various useful string constructors, but we have also extended CLASSIC to support reasoning about dates as well. Finally, because we extended the reasoning that CLASSIC is performing and because CLASSIC explains all of its inferences, we also provided for a way to explain the additional reasoning that is being performed.

Acknowledgments

We are indebted to Peter Patel-Schneider, who was the source of key ideas for individual reasoning with test concepts, as well as Lori Alperin Resnick and Loren Terveen, for their help.

References


