Optimizations and Parameters in FLEX++

Frank W. Bergmann,
Technische Universität Berlin,
Franklinstr. 28/29, D-10857 Berlin, Germany
phone/fax: ++49 30 691 66 74,
e-mail: fraber@cs.tu-berlin.de

Abstract

The application of Description Logics (DL) to natural language processing (NLP) is one of the most promising approaches to many linguistic problems. However, when dealing with real-world language fragments, performance problems appear due to the rapidly increasing size of the domain models.

This paper reports on DL optimization efforts done within the context of a particular NLP project. It also describes the parameters that were used to adapt the system behavior to this particular application and it briefly explains how these parameters have to be changed to adapt the system to applications with different profiles.

However, the known general DL performance problems became even worse due to the large number of implication links and default rules introduced to evaluate the linguistic phenomena. For that reason, a decision was taken in 1995 to reimplement the original DL system ("Prolog-FLEX") in C++ ("FLEX++"). The aim was to increase the DL performance by a factor 100 to allow the VERBMOBIL disambiguation application to process its linguistic tasks in real-time.

In the following chapters, the motivation for the FLEX++ reimplementations and its optimization strategies and parameters are described. Chapter 2 introduces to the application context of the VERBMOBIL project, so that the particular requirements of FLEX++ become clear, chapter 3 presents some optimization techniques and chapter 4 describes the parametrization of FLEX++ that allow to adapt the inference algorithms to the particular requirements.

1 Introduction

Processing of semantic information in the area of natural language processing has been long known as one of the most appealing areas to apply description logic. However, when applying DL to disambiguate natural language expressions derived from the VERBMOBIL project [4], several practical problems appeared, rising doubts on whether DL is really the right formalism for that task. Two main problems in this context are the restricted expressivity of DL compared with procedural approaches and performance problems due to a large knowledge base.

In order to overcome the first problem, a particular strategy was developed by the project team to represent VERBMOBIL natural language expressions as networks of objects. In this representation [8], "implication links" are used to draw linguistic inferences much in the style of a term replacement system. Additionally, a preferential default logic [10, 11] provides sufficient expressive power to represent fuzzy linguistic phenomena. To provide special support for these features, the DL system "FLEX" [9] was developed based on ideas of the "BACK" system [3].

2 Application Context

FLEX++ has been developed in the context of the VERBMOBIL project which aims to build a speech to speech translation system from German to English. As this project is described in detail by a paper of Uwe Küßner that appears on the same conference, please refer to this paper for a detailed description.

Here I restrict myself to present the DL structures that represent the interface between the linguistic modules and the FLEX system. Figure 2 shows a graphical representation of a typical DL structure to be processed by FLEX.

3 FLEX++ Implementation, Optimization and Parameterization

The implementation of FLEX++ as described in this chapter is based on (Prolog-) FLEX [9], which again is based on the BACK system [3]. The main challenge of the development of both systems was to optimize the systems to fit the specific requirements of VM11 disambiguation while maintaining the general applicability.
c₁ <= c₂ and ... <=> c₁ = "prim(c₁)" and c₂ and ...

Figure 5: Primitive and Defined Introduction of Concepts

tation time"), all possible inferences are drawn and all invalidated inferences are recalculated. During lookup ("query time") it is never necessary to draw an inference.

- Lazy: In this mode, a cache only "remembers" the results of query operations. A special data structure keeps the information on which inferences have already been drawn. When inserting or deleting an item, the invalidated inferences are deleted from the cache. During query time, it is first checked whether the cache already contains sufficient information to answer the query. If this is not the case, the inference is drawn and added to the cache.

- No caching: This mode is particularly important for debugging and benchmarking.

Concept Hierarchy Cache The most well known example for a cache is certainly the "concept hierarchy lattice" (or concept hierarchy cache, CHC), which consists of a directed anticyclic graph (DAG), representing the subsumption hierarchy between all concepts in the system. The underlying idea of this structure is that all concepts in the system are classified against this cache and inserted into their appropriate position.

Several optimization techniques in FLEX++ concerning the CHC have been adopted from [1]. In particular the monotonicity property of subsumption is extensively used to reduce the number of subsumptions required to classify a concept wrt. a given concept hierarchy. In comparison to the classification of objects wrt. non-monotonic hierarchies where an object has to be compared with every single concept, in a monotonic DL dialect it is possible to skip entire subtrees from classification, if the top of the subtree is known not to subsume the object. In addition, if an object is known to be subsumed by a certain concept, it is also known to be subsumed by all of the superconcepts of the concept. Similar properties hold for disjointness. Based on these conclusions, several classification algorithms are proposed.

A major speedup concerning the CHC and classification in general could be reached by evaluating the primitive components (prim()) of concepts. These "prims" are added to a concept normalform when a concept is introduced primitively. The correspondence is depicted in figure 3.3. Due to the fact that these prims are not allowed as part of valid DL formulas, a concept, object or Ilink that owns this property must have inherited it from the corresponding prim concept, and thus is known to be subsumed by the prim concept. In other words: To check subsumption between an arbitrary entity and a primitive concept, it is sufficient to check if the entity contains the corresponding prim property.

Role Hierarchy Cache Similar to the CHC, it is possible to construct a cache for the role hierarchy (RHC). This cache proved to be useful since many normalization and subsumption rules apply subsumption checks between roles.

Ilk Link Hierarchy Cache A particular speedup for object introduction could be reached by the introduction of a hierarchy cache for Ilinks (IHC). During the assertion of object, the following steps have to be completed:

1. The object descriptions are expanded wrt. the domain model (inheritance) and the resulting formulas are normalized.

2. All applicable implication rules are applied. For that purpose, each object has to be compared with all links in the system. If the the normalform of the object (ONF) is subsumed by the left hand side (LHS) of an Ilink, the right hand side (RHS) of this Ilink is added to the object, the ONF is renormalized and the process is repeated until no further Ilinks are applicable.

3. Properties of object that affect other object are propagated. Figure 3.2 shows an example of such a propagation rule. Such propagations lead to the modification of the affected object and, thus, to a reaplication of Ilinks.

Certainly, a comparison (subsumption check) has to be made between the ONF an the LHSs of an Ilink for every Ilink. Even worse, these comparisons have to be repeated every time a ONF is modified by a propagation of an Ilink application. To speed up this very essential algorithm the IHC was introduced.

The main effect of the IHC is that an object does not have to be (sequentially) compared with all Ilinks. Instead, the object is classified into the IHC, which results in a reduction of the number of subsumption checks. Entire subtrees can be skipped if the top node of the subtree does not subsume the ONF. The achievable speedup depends on the structure of the IHC.

A further optimization related to the IHC is called "IHC compilation" and aims to reduce the number of iterations of Ilink applications for chains of Ilinks. For example, the two Ilinks c₁ => c₂ and c₂ => c₃ can be transformed into c₁ => c₂ and c₃ and c₁ and c₂ => c₃. c₃ could be added to the RHS of the first Ilink, because the second Ilink is always applied.
Figure 1: Graphical Representation of "ich dachte wir treffen uns noch im april"

3.1 Language

The language of FLEX++ was in particular chosen avoid any "reasoning by case" inferenees, therefore qualified number restrictions were excluded. Cyclic structures are not allowed. The treatment of this language is considered to be complete.

The language is composed by the following structures (C=concept, R=role, O=object, W=weight, X=query variable, Cname/Rname/Oname=names, Olist=list of objects): Conjunctions (and, or, not), concept forming operators (Cname, oneof(Olist), some(R,C), no(R,C), all(R,C), filler(R,O)), role forming operators (Rname, domain(C), range(C), inv(R)), introduction operators (primitive concept introduction Cname := C, defined concept introduction Cname := C primitive role introduction Rname := R, defined role introduction Rname := R, object introduction/modification Cname := C, Inlink introduction C => C, Dlink introduction C "W" C) and query operators (concept subsumption C ?< C, role subsumption R ?< R, object-concept instanciation Cname := C, concept instances X := C, object superconcepts Cname := X)

Concepts, roles and objects will be called "entities" in the rest of the paper.

3.2 The FLEX++ Normalize-Compare Algorithm

The basic operation of all DL systems is "subsumption checking", a process to compare (the formulas of-) two DL entities to see whether the second entity is implied by the first entity with respect to a specific terminological logic. In order to check for subsumption between two concepts an algorithm exists which requires, first, to transform both entities into a canonical normalform ("normalize") and second, to compare structurally these normalforms ("compare"). This normalize-compare algorithm was originally developed by [15]. Several DL systems have adapted variants of this algorithm, including CLASSIC [12], LOOM [6] and BACK [3]. FLEX++ has also adapted this approach, with modifications based on [13] and [9]. Here, the calculation of the normalform is based on the notion of "inference rules", derived by adapting the inference rules of the Genzen Sequent Calculi [2, 16] to DL [13]. Conforming to these terminological Sequent Calculi, subsumption is calculated by first, applying repetitively a set of normalization rules and second, by comparing the resulting normalforms with respect to a set of subsumption rules. Figure 3.2 presents a typical example of a normalization rule, figure 3.2 shows a subsumption rule.

To be able to adapt to particular applications, FLEX++ uses "flexible inference rules" [9]. According to this approach, most inference rules exists as both, normalization and subsumption rules.

3.3 FLEX++ Caching Structures

A major concern of the FLEX++ implementation was to avoid two main sources of inefficiencies in DL systems - the multiple calculation of the same inference and the calculation of inferences that are never used. In order to attack these problems, FLEX++ provides several "caches" which are data structures that contain entities (roles, concepts, objects, links) together with the subsumption relation between them.

Caching Principles The problem with caches is that insertion or deletion of entities often results in an invalidation of previously calculated subsumptions stored in that cache or even in other caches. For this reason most caches in FLEX++ can be configured to run in one the following three modes:

- Greedy: In this mode, a cache is always up to date. When new entities are inserted to the cache ("insert-

all(R,C), filler(R,O) => O :: C.

Figure 4: Example of a Propagation Rule (informal notation)
when the first Il ink is applied. In general terms: If the LHS of an Il ink is known now to be subsumed by the conjunction of LHS with the RHS of another Il ink, then the first Il ink has always to fire when the second Il ink fired. Thus, the RHS of the first Il ink can be added to the consequence (RHS) of the second Il ink.

As a parameters to the IHC both, classification and compilation, can be switched off.

**Dlink Hierarchy Cache** The hierarchical organization of the IHC can be used for the DHC. However, due to the different logics of application, the DHC cannot be compiled.

**Object Instanciation Caches** A second major class of caches in FLEX++ is related to the classification of objects with respect to the CHC, IHC and DHC. The idea is to maintain the results of previous subsumption checks, so that no subsumption has to be drawn twice. For this reason, a particular data structure is attached to every object that holds those classification results, that remain valid in case information is added to the object.

**Concept Instances Cache** This cache is attached to every concept in the CHC, holding all instances (subsumed objects) of the concept. Using this cache, queries such as \( x \supseteq C \) (resulting in \( x \) being instanciated to the list of objects subsumed by \( C \)) can be answered by a simple lookup [7]. To build up this cache, every new object is classified against the CHC. Also, objects have to be reclassified after being modified.

Although this strategy proved to be an optimization for applications such as information systems, applications such as VM11 disambiguation do not frequently state such queries. For this reason, the CIC can be switched off.

### 3.4 Access Optimization

A large class of optimizations are related to access problems, occurring in DL queries. For example, in subsumption rules, the subsume is frequently queried for specific qualities. If no particular order of the elements in a normalform is given, a sequential search is required with linear complexity. Given a suitable organization of data structures, the search can be converted into a lookup so that the complexity can be reduced to be logarithmic or even constant.

**Organization of the FLEX++ Concept Normalforms** Experiences with Prolog-FLEX proved that most CPU-time in subsumption was spent to access role-related properties of concept normalforms (which are also part of objects, Il ink and Dink LHS and RHS). Following this insight, normalforms in FLEX++ are split into two parts: the "constant" part holding information about non-role-related properties such as prim and oneof components, and the "dynamic" part that contains all other information, sorted by the role as the first key. Access is implemented via a hash table with constant lookup complexity. All remaining information is stored in a single C++ record (constant lookup complexity) with exception of the qualified number restrictions \( \text{atleast}(N,R,C) \) and \( \text{atmost}(N,R,C) \) which involve a second hash table level for the concept\(^1\).

**Set Implementation** A second major access problem affects the C++ "set" library routines, in particular when dealing with large numbers of prim subsumptions. These operations are implemented as set inclusion. With the standard implementation of sets as lists of integers, these checks are of linear complexity [5], which leads to performance problems with a rising number of prim properties per entity. In addition, the tree implementation of sets proved inadequate due to the large insertion complexities.

As an approach to this problem, a specific bitset is used to hold an "abstract" of those sets, that are most frequently subject to inclusion checks (such as Il ink- and object prims). The desired property of these "abstracts" is that abstract inclusion is a necessary condition for the regular inclusion. This way, most of the regular inclusion checks can be avoided. To achieve this property, every element of the original set is transformed into a fixed range by a modulo operation, so that a fixed length set is created, that can be checked for inclusion with other fixed length sets with constant complexity.

Due to the fact that this optimization is only useful with particular application profiles, it is possible to switch it off.

### 4 Parameter Optimization

In the last chapter, various optimization strategies are presented that all utilize certain properties of "regular" FLEX++ input data. However as praxis shows, the input data of DL systems can vary heavily. For example a typical information system, an "ontology checker" and the VERBMOBIL application differ widely in their requirements towards an underlying DL system. An information system in general consists of a rather small ontology together with a large amount of objects. During runtime, very few insertions of concepts or roles are made while the main task is to find the instances of concepts or the subsuming concepts of an object. The task of an "ontology checker" mainly consists of inserting concepts and roles. Finally, in the VERBMOBIL

---

\(^1\)Qualified number restrictions are not part of the complete language fragment.
application a small number of objects is compared wrt a large number of links.

For a single DL system such as FLEX++, to support such different types of input not only requires the support the corresponding syntactic specialities. Also, most of the optimization techniques presented in the last chapter become inappropriate with atleast one of these applications. Thus the approach of FLEX++ is to provide parameters to be able to modify the behaviour of the DL system according to the input data.

Parameters to be Optimized During the development of FLEX++, the various caches proved to be the most interesting candidates for optimization. For this reason it is possible in FLEX++ to determine the caching strategy for most caches. Additionally, the entire inference process is subject to parametrization, because many inference rules can be configured to be activated either at subsumption or at normalization time.

The configuration of all parameters currently is handled using the “rule of thumb”. However this area is subject to active research. More detailed results will be presented in a future paper.

5 Conclusion

Taken together, the optimization techniques presented in this paper allowed to speedup the execution of the VM11 disambiguation application by a factor 100, compared with the 1995 Prolog-FLEX implementation. The conclusion to be drawn from this are:

- to avoid to calculate unnecessary inferences
- to avoid the multiple calculation of the same inferences
- and to set the parameters of all caching structures so that not too many inferences are calculated unnecessarily or too often.

6 Acknowledgements and Disclaimer

This is a report on work done within the context of the KIT-VM11 project, Technische Universität Berlin. This paper would not have been possible without the guidance and contributions of the project members Guido Dunker, Ivonne Kellner, Uwe Küßner, Nina Ruge, Birte Schmitz and in particular J. Joachim Quantz. This paper presents my personal point of view. Any errors or omissions are exclusively of my own.

References


