Navigating through Dense Annotation Spaces

Branimir Boguraev and Mary Neff

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
bran@us.ibm.com, maryneff@us.ibm.com

Abstract

Pattern matching, or querying, over annotations is a general purpose paradigm for inspecting, navigating, mining, and transforming annotation repositories—the common representation basis for modern pipelined text processing frameworks. Configurability of such frameworks and expressiveness of feature structure-based annotation schemes account for the 'high density' of some such annotation repositories. This particular characteristic makes challenging the design of a pattern matching engine, capable of interpreting (or imposing) flat patterns over an arbitrarily dense annotation lattice. We present an approach where a finite state device carries out the application of (compiled) grammars over what is, in effect, a linearized 'projection' of a unique route through the lattice; a route derived by a mix of static pattern (grammar) analysis and interpretation of navigational directives within the extended grammar formalism. Our approach achieves a mix of finite state scanning and lattice traversal for expressive and efficient pattern matching in dense annotations stores.

1. Dense Annotation Spaces

Recent years have seen a strong trend towards evolving notions of robust and scalable architectures for natural language processing, crucially utilizing *annotation-based representation* for recording, and transmitting, results of individual component analysis (Cunningham and Scott, 2004).

Beyond the generalized view of annotation formats developed in (Bird and Liberman, 2001), annotation-based representational schemes have evolved to support complex data models and multiply-layered annotation-based analyses over the same corpus. For instance, to address some issues of reusability, interoperability and portability, (Hahn et al., 2007) at JULIE Lab[1] have developed a comprehensive annotation type system capturing document structure and meta-information, together with linguistic information at morphological, syntactic and semantic levels. This naturally will result in multiple annotations over the same text spans. Orthogonally, initiatives like the NSF project on Unified Linguistic Annotation (ULA) and the Linguistic Annotation Framework (LAF) developed within ISO[2] (see, for instance, (Verhagen et al., 2007)'s MAIS, (Ide and Suderman, 2007)'s GrAF) argue for the need for annotation formats to support multiple, independent, and alternative annotation schemes, where a specific type of *e.g.* semantic analysis can be maintained separately from, and without interfering with, semantic annotations at other layers: consider, for instance, the ULA focus on integrating PropBank-, NomBank-, and TimeBank-style annotations over the same corpus, while maintaining open-endedness of the framework so other annotation schemes can be similarly accommodated. Multiple annotations—possibly even carrying identical labels over identical spans—are also a likely characteristic of such environments.

From an engineering standpoint, such complexity in annotation formats and schemes is already tractable. For instance, by providing a formal mechanism for specifying annotations within an arbitrarily complex type hierarchy based on feature structures (further enhanced by multiple analysis views within a document, and awareness of namespaces for feature structure types), the Unstructured Information Management Architecture framework (UIMA; (Ferrucci and Lally, 2004)[3]) offers the representational backbone for the requirements of the JULIE project, as well as the ability to support (or be isomorphic to) MAIS' and GrAF's multiple annotation layers.

Such annotation frameworks, however, make for dense annotation spaces; these are, essentially, annotation lattices. Typically, there will be numerous multiple annotations over the same text span. Annotations will be deposited in a common annotations store by the individual components of the particular processing pipeline. Related, or inter-dependent components may—but do not have to—deposit annotations which are 'aligned' (*i.e.* not overlapping); non-related components will have no such mandate. (For instance, just bringing more than one tokenizer into the pipeline is certain to produce mis-aligned tokens.) Trees (or tree fragments) may be encoded by relative overlays of annotation spans, possibly mediated by an auxiliary system of features (properties) on annotations.

In such annotation spaces, matching (or querying) over annotations requires interpretation of sub-graphs within an annotation lattice. We assume that such a process is mediated via some formal language; this paper addresses some of the requirements—and underlying support—for such language to resolve the ambiguities of traversal associated with the kinds of lattices under discussion.

We argue for a specially adapted lattice traversal engine designed with the capability to focus—simultaneously—on *sequences of annotations*, on annotation structures isomorphic to *tree fragments*, and on specific *configurational*

[1]The Language and Information Engineering Lab at Jena University, http://www.julielab.de/.

[2]International Standards Organization, Technical Committee 37, Sub-Committee 4, *Language Resource Management*, http://www.iso.org/iso/iso_catalogue/catalogue_tc.

[3]See also http://incubator.apache.org/uima/.

relationships between annotations. For example, sequential constraints might be used in an annotation pattern for some meaningful relationship between a pre-nominal modifier and the nominal head following it; tree matching over a predicate-argument fragment may be used to identify the node annotation for a labeled argument; and some process may need to iterate over all annotations of certain type—say noun phrases—excluding from its considerations NP instances in a certain configurational relationship with other annotations (*e.g.* not sentence-initial, or not internal to a prepositional phrase), or which exhibit certain internal structural properties (*e.g.* containing an annotation over a temporal expression). (We will refer to such constraints, in general, by using terms like 'horizontal' and/or 'vertical'.)

Such examples illustrate some of the issues in navigating annotation lattices; additional complexity may be added by the characteristics of a particular lattice, or by the specific needs of an application. For instance, multiple, non-aligned token streams may pose a challenge to figuring out what the 'next' token is, following some higher-level constituent. Likewise, an application may need to aggregate over a set of annotation instances—*e.g.* for consistency checking—before it posts a higher-level, composite, annotation (consider the constraint that an electronic signature may contain at least two of name, affiliation, address, or phone number, in any order; or that a temporal relation must have two arguments, one of which is a temporal expression).

2. Navigational Challenges

As the above examples further indicate, in many situations an annotation may present an inherent ambiguity: is it to be treated as an atomic object, or as a structured fragment, the shape of which matters? Consider, for instance, a representation scheme where proper names are annotated (say, by [PName] annotations), with their internal structure further made explicit by annotating, as appropriate, for [Title] and [Name], itself further broken into *e.g.* [First], [Middle], and [Last]. Some of these annotation types may, or may not, appear in any particular instance. Now, it is possible to conceive of an application which just needs to iterate over [PName]s; it is also possible, however, to imagine a need for only collecting *e.g.* titled proper names, or proper names whose [Last] components satisfy some constraint. The former iteration regime would only 'care for' [PName] annotations; the latter needs to both identify such annotations *and* simultaneously inspect their internal components—possibly located several layers below the top—within the same traversal step.

This unit-vs-structure dichotomy is orthogonal to a different kind of ambiguity with respect to an annotation: is it to be visible to an underlying lattice traversal engine or not? Visibility in this sense may be contingent upon focusing on one, and ignoring alternative, layers of annotation in, say, ULA-style of analysis (*e.g.* , iterate over NP's in PropBank annotations, but ignore NomBank NP annotations). Alternatively, visibility may also be defined in terms of what matters to an abstract 'pattern': in the example of parsing an electronic signature above, it is convenient to cast the rule in terms of meaningful components, and not even specify the optional use of punctuation tokens, such as commas, hyphens, *etc*.—the intuition here is that the formalism should drive a traversal regime which is only sensitive to some, but not all, annotation types encountered.

Yet another perspective on visibility derives from the combined effects of likelihood of multiple annotations over exactly the same span, *and* the application's need to inspect two (or more) of them, conceptually during the same traversal. This is complementary to our first scenario above, and highlights the broad range of iteration regimes which need to be supported by the annotation matching infrastructure.

Our work develops an annotation traversal framework which addresses the challenges of navigating dense annotation lattices. Fundamentally, the design seeks to exploit insights from research in finite-state (FS) technology, appealing both to the perspicuity of regular patterns and the efficiency of execution associated with FS automata. Our AFst framework, however, makes suitable adaptations in order to reduce the problem of traversing a lattice of annotations to that of FS-tractable problem of scanning an unambiguous stream of 'like' objects, in our case (UIMA) annotations.

Broadly speaking, we address similar challenges to those identified by research on querying annotation graphs (Bird et al., 2000). However, rather than focusing on strongly hierarchical representations and mapping queries to a relational algebra (SQL) as in for instance (Lai and Bird, 2004), we seek a solution ultimately rooted in 'linearizing' the annotation lattice into an unambiguous annotation stream, so that traversal can be realized as a finite-state process. This fits better not just activities like exploration of an annotated corpus, but also an operational model of composing an application, where a pattern-matching annotation engine implements, via a set of fully declarative grammars, an active annotator (*e.g.* a parser, a named entity detector, or a feature extractor). The focus of this paper is to present the basic design points of the framework (and associated pattern specification language) facilitating such linearization.

3. Related Work: Patterns over Annotations

Several approaches address the problem of matching over annotations. Abstractly, two broad categories can be observed. A class of systems, like those inspired by (Grefenstette, 1999), (Silberztein, 2000), (Boguraev, 2000), (Grover et al., 2000), (Simov et al., 2002), essentially deconstruct an annotations store data structure into a string (suitably adorned with in-line annotation boundary information) and apply FS matching over that string. At representational level, annotations may be represented internally either in a framework-specific way, or by means of XML markup. There are many attractions to using XML (with its requisite supporting technology, including *e.g.* schemas, parsers, transformations, and so forth) to emulate most of the functions of an annotations store (but see (Cassidy, 2002) for an analysis of some problems of adopting XML as an annotation data model, with XQuery as an interpreter). However, not all annotations stores can

be rendered as strings with *in-line* annotations: difficulties arise precisely in situations where ambiguities discussed in Section 2. are present. Consequently, overlapping, non-hierarchical, multi-layered annotation spaces present serious challenges to traversal by such a mechanism.

Alternatively, overlaying FS technology on top of structured annotations which are 'first-class citizens' in their respective architecture environments is exemplified by a different class of systems, most prominently by GATE's JAPE (Cunningham et al., 2000) and DFKI's SProUT (Drożdżyński et al., 2004). While these two are substantially different, a crucial shared assumption boils down to the annotation traversal engine 'knowing' (with the knowledge then implicitly used in the rule sets/grammars) that components upstream of it will have deposited annotations of certain type(s) in ways such that the lattice to be traversed would behave like a flat sequence (in the case of GATE), or would encode a strictly hierarchical set of type instances (in the case of SProUT).This is not an assumption which necessarily holds for projects like the ones outlined earlier (Section 1.), nor does it adequately describe the proliferation of possibly ambiguous, or even contradictory, annotations (especially from early components in complex pipelines), typical of large-scale architectures where an arbitrary number of annotator components may deposit conflicting/partially overlapping spans in the annotations store.

4. Elements of Annotation-Matching Formalism

Our AFst framework addresses the design considerations for traversing and navigating annotation lattices by exposing a language which, to its user, provides constructs for specifying *sequential*, *structural*, and *configurational* constraints among annotations. It thus borrows notions from regular algebra for pattern matching, tree traversal for structure decomposition, and type prioritization for configurational statements; these are embedded in a notational framework derivative of cascaded regular expressions.

4.1. Pattern Specification

In an annotations store environment, where the only 'currency' of representation is the annotation-based type instance, FS operations have to be defined over annotations and their properties. AFst thus implements, in effect, *a finite state calculus over typed feature structures*, cf. (Drożdżyński et al., 2004), with pattern-action rules where patterns would be specified over configurations of type instances, and actions which manipulate annotation instances in the annotations store (see below). The notation developed for specifying FS operations is compliant with the notion of a UIMA application whose data model is defined by means of externally specified system of types and features.

At the simplest level of abstraction, grammars for AFst can be viewed as *regular expression (-like) patterns over annotations*. This allows for finding of sequences of annotations with certain properties (*e.g.* nouns following determiners, unbroken stream of tokens with certain orthographic feature (such as capitalized), or noun group–verb group pairs in particular contexts). However, given that each transition of the underlying FS automaton is licensed by an arbitrarily complex set of constraints, the notation incorporates more complex syntax for specifying what annotation to match, what are the conditions under which the match is deemed to be successful, and what (if any) action is to be taken with respect to modifying the annotations store. Much of that complexity is borne by a symbol notation, indicative of the set of operations (typically over the annotations store) that need to be carried out upon a transition within the transition graph compiled from the FS skeleton. Thus, for instance, where a character-based FS automaton would be examining the next character in its input tape, our AFst interpreter may be asked to perform progressively more complex operations. Examples of such operations, expressed as symbols on the arcs of an FS automaton, are typically: match over an annotation of type `T`, possibly examining values of its feature(s) (`Token[]`, `Person[kind=~"named"]`), and post a new annotation at the point of a successful match, over a given input span, with optionally setting values for its feature(s) (`NP[]/]Subj[passive="false"]`). Later, we will show how elements from both the symbol and grammar notations can be augmented to affect navigation.

A (very simple) grammar for noun phrases, defined over the part-of-speech tags of [`Token`] annotations, is shown below. The symbol `<E>` marks an empty transition (a match which always 'succeeds'), and the operators `.` and $\star$ specify, respectively, sequential composition, and zero or more repetitions, of sub-patterns. In effect, the grammar looks for a sequence of tokens, which starts with an optional determiner, includes zero or more adjectives, and terminates with a singular or plural noun. If such a sequence is found, a new [`NP`] annotation is created into the annotations store.

```
np = <E>/[NP .
      Token[pos=~"DT"]|<E> .
        Token[pos=~"JJ"]* .
          Token[pos=~"NN"] | Token[pos=~"NNS"] .
     <E>/]NP ;
```

Note that `Token` is just another type in a UIMA type system: there is nothing special about querying for its `pos` feature. Thus, if an upstream annotator has deposited, say, temporal expressions in the annotations store, the pattern above could incorporate certain type of expression as a nominal premodifier, *e.g.* by specifying `Timex[kind=~"date"]` as an alternative to `Token[pos=~"JJ"]`.

In essence, by appealing to UIMA representational devices—in particular, its type system specification notation which not only prioritizes types, but also defines a type subsumption hierarchy—both sequential (and even order-independent) patterns over annotations and vertical configurations among annotations may be specified at different levels of granularity, in an open-ended and application-agnostic fashion. Moreover, by relocating the data model to a specification outside of the traversal engine itself the framework allows for a relatively small set of AFst language constructs, which can manipulate annotations (both exist-

ing and newly posted) and their properties, without the need for *e.g.* admitting code fragments on the right-hand side of pattern rules (as GATE does), or appealing to 'back-door' library functions from an FST toolkit (as SProUT allows), or having to write query-specific functions (as XQuery would require). Both 'match' and 'transduce' operations appear as atomic transitions within a finite-state device. Matching operations are defined as subsumption among feature structures. Transduction operations (which largely remain outside of the scope of this paper) not only create new annotations, but also facilitate, by means of variable setting and binding, feature percolation and embedded references to annotations as feature values.

4.2. Navigational Constraints

Additional notational devices—both at symbol-matching and pattern-specification levels—capture, and convey to the pattern interpreter, information relevant to navigating the lattice. As we shall see, by referencing the UIMA type system, vertical configurational constraints can be interleaved within the normal pattern-matching operations.

In essence, AFst addresses the problem of explicitly specifying the route through the lattice as part of a regular pattern within the FST backbone by delegating the annotation lattice traversal to UIMA's native iterators—with, as we shall see, suitable provisions for control. In general, UIMA iterators are customizable with a broad set of methods for moving forwards and backwards, from any given position in the text, with respect to a range of ordering functions over the annotations (in particular, start/end location, type, and type priority; this last parameter refers to the intuitive notion of ordering of types with respect to which should be returned first, when an iterator encounters multiple type annotations over the same text span).

A key insight in our design is that a transition graph, once compiled, specifies exactly the type of annotation required by any given transition. At points in the lattice where this is underspecified, the notation allows for guided choice. (As we shall see, there is always a default interpretation, driven by the particular type hierarchy and system of type priorities.) Our insight thus translates into the dynamic construction of a special kind of *mixed* iterator, which is different for every grammar as it depends on the set of types over which the grammar is defined. It is this mixed iterator which mediates the traversal of the annotation lattice in a fashion corresponding to threading through it a route consistent with all, and only, the set of types of interest to the grammar. Note that grammar-level specification of horizontal *and* vertical constraints is compiled into a particular sequence of matches over annotations. In effect, the mixed iterator removes the fan-out aspects of lattice traversal and replaces them with a single pass-through route which behaves just like an unambiguous stream of 'like' objects. The following section examines this in more detail.

5. Support for Navigational Control

The previous section outlined how the symbol notation captures extensions to the notion of FS-based operations, to apply to a stream of 'like' objects: in this case, annotations picked—in a certain order—from an annotations store. Since these can be complex feature-structure objects, the symbol notation uses appropriate expressive devices, designed to inspect the internal make-up of annotation instances. The question remains, however, about the mechanism whereby the AFst interpreter constructs the annotations stream which is paired, at execution time, with the particular FST graph for a given grammar. In other words, how is the route through the annotations lattice chosen, resulting in a particular linearized projection of the arbitrarily dense lattice to a stream congenial to the FS machinery.

There are essentially two complementary aspects of the navigation problem. As we have already stated (Section 4.2.), route selection is delegated to the UIMA iteration subsystem: at a higher level of abstraction, an iterator is responsible for traversing the lattice in such a way that from the AFst interpreter point of view, there is always an annotation instance presented, unambiguously, as the `next()` object to be inspected (according to the transition graph). The type of this instance is defined with respect to a subset of all the types in the annotations store; the exact manner of this definition, and mechanisms for unambiguously selecting the `next()` one, are discussed in Section 5.1. below. Navigation control is thus, in effect, distributed among the definition of a suitable UIMA iterator and extensions to the notation (largely for symbols, less so at a grammar level) capable of informing the iterator.

We allow for a range of mechanisms for specifying, and/or altering, the iteration; accordingly, there are representational devices in the AFst notation for doing this. Broadly speaking, at grammar level there are three kinds of control:

- 'typeset' iterator, inferred from the grammar,
- declarations concerning behavior with respect to a match,
- distributing navigation among different grammars, via grammar cascading.

These controls mediate the left-to-right behavior of the interpreter. Additionally, at symbol specification level, there are devices for shifting the traversal path of the interpreter, in a more up-and-down (vertical) direction.

5.1. Iterator Induction

As we have seen, a transition symbol explicitly specifies the annotation type it needs to inspect at a given state. It follows that by examining a grammar, it is possible to derive a complete set of the annotation types of interest to this grammar. A *typeset* iterator, then, is a dynamically constructed[4] instance of a UIMA iterator, which filters for a subset of types from the larger application's type system, and is configured for unambiguous traversal of the annotations store. The grammar fragment in Section 4.1., for example, will induce the construction of a typeset iterator filtered for `[Token]`s only, no matter how many and what other types are in the type system. Of course, there is nothing special about `[Token]`'s (which are just types in a type system):

[4] At grammar compilation/load time.

a different grammar, for example, relabeling [NP] annotations to the left and right of a [VerbGroup] as [Subj] and [Obj], would be agnostic of [Token]s.

More than one type may (and likely will) end up in the iterator filter, either by explicit reference on a grammar symbol or implicitly, as a result of the grammar specifying a common supertype as licensing a transition (supertypes act as 'wild cards' in the AFst notation). At points in the lattice, then, where more than one of the types of interest have a common 'begin' offset, the iterator will—in line with its unambiguous nature, and crucially for effectively linearizing the lattice—have to make a choice of which annotation to return as the `next()` one.

By default, the typeset iterator follows the natural order of annotations in the UIMA annotations store: first by start position ascending, then by length descending, then by type priority. Type priorities thus control the iteration over annotations; they are particularly important in situations where annotations are stacked one above the other, with the 'vertical' order conveying some meaningful relationship between types. A representation for proper names, like the one outlined in Section2., would capture—by means of explicit priorities definition—statements like [PName] over [Title] and [Name], and [name] over [First]/[Last]; similarly, intuitions like [Sentence] sits 'higher' in the lattice vertical order than [Phrase], itself above [Tokens].

With its broader filter, the typeset iterator for a grammar like the one outlined above (relabeling [NP]-[VG]-[NP] triples as [Subj]-[VG]-[Obj], and additionally making references to [Token]s) would face traversal ambiguities at points where the [NP] and [VG] annotations start—as there are underlying [Token]s starting there as well. The iterator will behave, however, unambiguously, according to the priority constraints above; this default behavior is largely consistent with grammar writers' intuitions. We will shortly show how to alter this behavior.

Conversely, there may be situations where a pattern may be naturally specifiable in terms of lower-level (priority-wise) annotation types, but the navigation regime needs to account for presence of some higher types in the annotations store, even if they are not logically part of the pattern specification. Consider an address annotator, where numbered tokens may be part of a street address, and a date annotator, which might interpret (some) numbers as years. If there are [Address] annotations already, a [Date] annotator should not 'descend' under them, to inspect [Address]-internal [Token]s; yet there is no natural way in which date patterns might be made aware of (pre-annotated) address components. In fact, this is a common situation in pipelined text processing environments, where multiple annotators of varied provenance operate in sequence, but not necessarily sharing knowledge of each other.

5.2. Grammar-Wide Declarations

In such situations, the second control device comes into play: types external to a grammar can be brought into the typeset iterator filter by means of an `honour` *declaration*:

```
honour % Address[] ;

month  = Token[lemma=~"January"] |
         Token[lemma=~"February"] |
         ... ;
date   = <E>/[Year . :month|<E> .
         Token[string=~":^[12]\d{3}$:]/]Year ;
```

Without the `honour` declaration, the grammar fragment above would induce a typeset iterator over [Token]s, and the pattern would trigger over something like ... *1650 Sunset Boulevard*, posting [Year] over *1650*. The declaration adds [Address] to the typeset iterator filter, preventing inspection of the [Token]s under [Address].

Other declarations affecting navigation are `boundary`, `focus`, `match`, and `advance`. Typically, the scope of the iterator is defined with respect to a covering annotation type; by default, this is [Sentence]. The intent here is to prevent posting of new annotations across sentence boundaries. The `boundary` declaration caters for other situations where the scope of pattern application is important: we would not want to, for instance, have the [Subj]-[Obj] relabeling pattern to trigger across a clause boundary, with a verb group marking the beginning of a [Clause] annotation—`boundary % Clause[];` sees to that. (Note that there may be multiple `boundary` annotations.) Our typeset iterator is thus defined as a *sub-iterator* under a boundary annotation, with the first annotation of a type in the set that starts at or after the beginning of the boundary annotation and finishing with the last one of a type in the set that ends at or before the end of the boundary annotation.

The `focus` declaration allows restricting the operation of a grammar to just those segments of the text source 'below' one or more `focus` annotation types. Arbitrary constraints (and arbitrary levels of nesting) can be specified on a `focus` type. This caters to situations where different (sets of) grammars are appropriate to *e.g.* different sections of documents, and allows for retargeting of grammars.

A `match` declaration controls how the iterator decides what match(es) to return as successful; parameters here may be, for instance, `match % longest;` (the default), or `match % all;`). While not directly affecting navigation *per se*, this affects the iterator behavior, and thus plays into the mix of devices whereby the grammar writer can fine-tune the pattern application process.

Finally, an `advance` declaration specifies how/where to restart the iterator immediately upon a successful match. By default, the iterator starts again with the next annotation after the *last one* it posts. This allows any specified right context (to the pattern just applied) to be considered for the next match (the current pattern). There are two alternative behaviors that can be invoked via this declaration: an `advance % skip;` or `advance % step;`. In the former case, the iterator is advanced to the first position after the *end* of the match; in the latter, the iterator is advanced to the next position after the *start* of the match. `skip` thus does not examine right context to a prior match; the alterna-

tive regime is useful in situations where more fine-grained context examination is essential for pattern application.

The scope of all declarations is the entire grammar. Note that it is always possible to partition a grammar and derive an equivalent grammar cascade, with different declarations applying to the pattern subsets in the multiple grammar sources.

5.3. Grammar Cascading

In fact, grammar cascading is the third global mechanism for controlling navigation. Grammar cascades were originally conceived as a device for simplifying the analysis task, by building potentially complex structures by partial, incremental, analysis and from the bottom up (*e.g.* first find [NP] annotations, then do some more syntactic phrase analysis, and only then use all the information in the annotations store to promote some [NP]s to [Subject]s).

Grammar cascading, however, has an additional role to play in facilitating navigation, especially in dense annotation spaces with multiple annotations present over the same text span. Separating patterns which target related, and small, subsets of types into different grammars achieves, in effect, a stratification of the annotations store. Different patterns, at different levels of granularity of specification, can be concisely and perspicuously stated, without bringing too many different annotation types (especially from different levels of analysis and representation), into the typeset iterator's filter.

5.4. Up-Down Attention Shifts

There are two primary notational devices for redirecting the iterator's attention in vertical, as opposed to horizontal, direction. One of them deals with situations we encountered earlier: how to register a match over a 'higher' annotation, while simultaneously detecting a pattern over its components. In Section 5.1. we saw how to point the typeset iterator at the higher, or lower, level of traversal. Here, we introduce another special purpose iterator: *mixed*, for dual scanning regimes.

Mixed iteration is essential for a common task in pattern matching over layered annotations stores: examining a 'composite' annotation's *inner contour*. We already saw examples of this, such as collecting titled proper names only, or proper names whose [Last] components satisfy some constraint (Section 2.); matching on noun phrases with temporal premodifiers (Section 4.1.). Arguably, this kind of traversal can be realized as a single-level, left-to-right, scan over annotations with appropriately rich and informative feature structure make-up (*i.e.* have features carry the information whether a [PName] instance has a [Title] annotation underneath it. In an environment, however, where annotators can (and will) operate independently of each other, and where, furthermore, annotations from different processes can coexist, we cannot rely on consistent application of disciplined recording of annotation inner structure exclusively by means of features.

In order to see whether a sequence of annotations that a higher annotation spans conforms to certain configurational constraints, what we would need to communicate to the interpreter amounts to the following complex command:

- test for an annotation of a certain type, with or without additional constraints on its features;
- upon a successful match, *descend* into this annotation;
- test whether a given pattern matches *exactly* the sequence of lower annotations covered by the higher match;
- if the sub-pattern matches, pop back (*ascend*) to a point *after* the higher level annotation;
- succeed,
- and then proceed.

Implementationally, the 'upper iterator' is stacked, the current annotation becomes the boundary annotation, a new typeset subiterator is instantiated with the lower types in its filter, and the next lower level is linearized for performance.

The notational device used for such an operation employs a pair of *push* and *pop* operators, available as meta-specifiers on symbols. Conceptually, if `Higher[...]` is a symbol matching an annotation which could be covering other annotations, `Higher[...,@descend]` would signal the 'descend into' operation. (The `@descend`, and matching `@ascend`, are instances of *interpreter directives*—notational devices which, while syntactically conforming to elements in an AFst symbol specification, function as signals to the interpreter to shift to a higher/lower lattice traversal line.)

Dual scanning offers a way to perform tree traversal, in annotation structures where overlayed, edge-anchored annotations encode a tree structure, by means of interpreting full/partial alignment and relative coverage of spans. Consider the following convention.

- an annotation corresponds to a tree node;
- two annotations with different spans belong to the same sub-tree of their spans are strictly overlapping: *i.e.* the span of one must completely cover the span of the other;
- the annotation with the longer span defines a node which is 'above' the node for the annotation with the shorter span;
- if the two annotations are co-terminous at both ends, the annotation with higher priority defines the higher node of the two in the sub-tree.

It is straightforward to visualize a tree for [PName] annotations (*e.g. "[[President] [[Vladimir] [Putin]]]*).

```
[PName
  [Title ... Title]
  [Name
    [First ... First] [Last ... Last] Name] PName]
```

Within such a convention, the following expression encodes, in effect, a query against the set of `Person` trees in the database, which will match all persons who can be referred to as *"Dr Smith"*:

```
findDrSmith =
  <E>/PName[@descend] .
    Title[string=~"Dr"] .
    <E>/Name[@descend] .
      First[]|<E> . Last[string=="Smith"] .
    <E>/Name[@ascend] .
  <E>/PName[@ascend] ;
```

In a number of situations, inspecting configurational properties of the annotations lattice requires an operation conceptually much simpler than tree traversal. The `@descend`/`@ascend` mechanism requires that the grammar writer be precise: the entire sequence of annotations at the lower level needs to be consumed by the sub-iterator pattern, and the exact number of level shifts (stack push and pop's) have to be specified, in order to get to the right level of interest.

In contrast, the expressive power of the notation gains a lot just by being able to query certain positional relationships among annotations in vertical direction. A set of different interpreter directives, again cast to fit into the syntax of AFst symbols, test for relative spans overlap, coverage, and extent. Symbols specifying such configurational queries may look like the following.

```
Token[_costarts=~Sentence[]]
Subject[_covers=~PName[]]
PName[_costarts=~NP[],_coends=~NP[]]
```

The first example matches only on sentence-initial tokens, the second tests if there is a proper name within the span of [Subject], and the third one examines whether a [PName] annotation is co-terminous with an [NP] annotation.

The inventory of such directives is small; in addition to the three examples above, there is also _below. In contrast to the way 'descend/ascend' operates, here inspection of appropriate context above, or below, is carried out without disturbing the primary, left-to-right iterator movement. This improves the clarity of pattern specification, results in a more efficient runtime characteristics, and allows for testing for configurational constraints among two levels of a lattice separated by arbitrary (and perhaps unknown) number of intermediate layers.

6. Conclusion

This paper focuses largely on support for navigating annotation spaces: *i.e.* those aspects of a notational system whereby patterns over annotation sequences and constraints over annotation configurations can be succinctly expressed and efficiently carried out by an interpreter largely operating over an FST graph. The full language specification can be found in (Boguraev and Neff, 2007). The AFst framework is fully implemented as a UIMA annotator, complete with grammar and symbol compilers and a runtime engine. A number of optimizations (most prominently to do with pre-indexing of all instances of annotations from within the current typeset iterator, maintaining order and span information on all possible routes through the lattice instantiating only the iterator type set) ensure efficient performance in the light of real data.

The framework supports diverse analytic tasks. Most commonly, it has been used to realize a range of named entity detection systems, in a variety of domains. Named entity detection has typically been interleaved with shallow syntactic parsing, also implemented as a cascade of AFst grammars (Boguraev, 2000). The ability to mix, within the same application, syntactic and semantic operations over an annotations store offers not just well known benefits like generalizing over syntactic configurations with certain distributional properties—*e.g.* for terminology identification in new domains (Park et al., 2002). More recently, we combined fine-grained temporal expression parsing (realized as a kind of named entity recognition for time expressions) with shallow parsing for phrase, and clause, boundaries, for the purposes of extracting features for classification-based temporal anchoring (Park et al., 2002).

While the bulk of the grammar formalism evolved from the requirements of 'linear' pattern specification, considerations of *e.g.* constraining patterns to certain contexts only, expressly managing lattice traversal at higher levels of a grammar cascade, and resolving ambiguities of choice between *e.g.* lexical (token-based), semantic (category-based), and syntactic (phrase-based) annotations over identical text spans, have informed extensions of the formalism to do specifically with lattice traversal, and motivated the notational devices described in the previous sections. Issues of reconciling syntactic phrase boundaries with semantic constraints on *e.g.* phrase heads, especially where semantic information is encoded in types posted by upstream annotators unaware of constraints upon the grammars intended to mine them, have largely led to the design of our different iterator regimes, up-and-down attention shifts, scan controls, and principles of type priority specification and use. Most recently, we have encountered situations where the density of the annotations lattice—due to proliferation of semantic types—is such that a strictly unambiguous iteration regime is poorly served by the need to control it by explicit up/down directives, informed by a type priority system: after all, if the upstream annotator(s) responsible for depositing the plethora of types in the annotations store do not have a uniform and consistent notion of priorities, it may be the case that such a notion can be inferred at the point where a set of AFst grammars come to play.

This motivates one of the principal items in our future work list: extending the run-time with a new iterator, designed to visit more than one annotation at a given point of the input. Informally, this is to be thought of as a 'semi-ambiguous' iterator: it will still be like a typeset iterator, but in situations where instances of more than one type (from its type set) being encountered in the same context, the iterator will visit all of them (in contrast to choosing the higher priority one, or following explicit `@descend`/`@ascend` directives.

From an implementation point of view, the AFst architecture already allows for 'plugging' in of different iterators,

effectively swapping the (default) unambiguous typeset iterator with the semi-ambiguous variant outlined above. Given the inherently grammar-wide 'scope' of an iterator, the cascade model allows for mixing different iterators while still processing the same input.

An additional extension of the framework is motivated by the observation that with the extended expressiveness of annotation-based representational schemes—especially in line with UIMA's feature-based subsumption hierarchy of types—syntactic trees can be directly encoded as sets of annotations, by means of heavy use of pointer-based feature system where a (type-based) tree node explicitly refers to its children (also type-based tree nodes). Such a representation differs substantially from the implied tree structure encoded in annotations spans (as outlined in Section 1.). Within the iterator plug-in architecture discussed here, such tree traversal can be naturally facilitated by a special-purpose, 'tree walk' iterator. Note that this is a different, and potentially more flexible, solution than one deploying tree-walking automata, like reported for instance in (Srihari et al., 2008)—as it naturally addresses the variability in encoding schemes mediating between tree characteristics (possibly dependent upon linguistic theory and processing framework) and the corresponding annotation-based representation.

These proposed extensions would complete the set of devices necessary for annotation lattice navigation, no matter how dense the lattice might be. Overall, the AFst formalism—and in particular the notational components for considering, and reacting to, both horizontal and vertical contexts—offers a perspicuous, efficient, scalable and portable mechanism for exploring and mining dense annotation spaces.

7. References

Steven Bird and Mark Liberman. 2001. A formal framework for linguistic annotation. *Speech Communication*, 33(1–2):23–60.

Steven Bird, Peter Buneman, and Wang-Chiew Tan. 2000. Towards a query language for annotation graphs. In *Second International Language Resources and Evaluation Conference*, Athens, Greece, May.

Branimir Boguraev and Mary Neff. 2007. An annotation-based finite state system for UIMA: User documentation and grammar writing manual. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, New York.

Branimir Boguraev. 2000. Towards finite-state analysis of lexical cohesion. In *Proceedings of the 3rd International Conference on Finite-State Methods for NLP,* INTEX-3, Liege, Belgium, June.

Steve Cassidy. 2002. XQuery as an annotation query language: a use case analysis. In *Third International Language Resources and Evaluation Conference*, Las Palmas, Spain, May.

Hamish Cunningham and Donia Scott. 2004. Software architectures for language engineering. Special Issue, *Natural Language Engineering*, **10**(4).

Hamish Cunningham, Diana Maynard, and Valentin Tablan. 2000. JAPE: A Java annotation patterns engine. Technical Memo CS-00-10, Institute for Language, Speech and Hearing (ILASH), and Department of Computer Science, University of Sheffield, Sheffield.

Witold Drożdżyński, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu. 2004. Shallow processing with unification and typed feature structures — foundations and applications. *Künstliche Intelligenz*, (1):17–23.

David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(4). Special Issue on Software Architectures for Language Engineering.

Gregory Grefenstette. 1999. Light parsing as finite state filtering. In András Kornai, editor, *Extended finite state models of language*, Studies in Natural Language Processing, pages 86–94. Cambridge University Press, Cambridge, UK.

Claire Grover, Colin Matheson, Andrei Mikheev, and Marc Moens. 2000. LT-TTT: A flexible tokenisation tool. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 1147–1154, Spain.

Udo Hahn, Ekaterina Buyko, Katrin Tomanek, Scott Piao, John McNaught, Yoshimasa Tsuruoka, and Sophia Ananiadou. 2007. An annotation type system for a data-driven NLP pipeline. In *Linguistic Annotation Workshop (the LAW); ACL-2007*, Prague, Czech Republic, June.

Nancy Ide and Keith Suderman. 2007. GrAF: a graph-based format for linguistic annotation. In *Linguistic Annotation Workshop (the LAW); ACL-2007*, Prague, Czech Republic, June.

Catherine Lai and Steven Bird. 2004. Querying and updating treebanks: A critical survey and requirements analysis. In *Australasian Language Technology Workshop*, Sydney, December.

Youngja Park, Roy Byrd, and Branimir Boguraev. 2002. Automatic glossary extraction: beyond terminology identification. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING)*, pages 772–778, Taiwan.

Max Silberztein. 2000. INTEX: An integrated FST development environment. *Theoretical Computer Science*, 231(1):33–46.

Kiril Simov, Milen Kouylekov, and Alexander Simov. 2002. Cascaded regular grammars over XML documents. In *Proceedings of the Second International Workshop on NLP and XML (NLPXML-2002)*, Taipei, Tawian, September.

Rohini K. Srihari, Wei Li, Thomas Cornell, and Cheng Niu. 2008. Infoxtract: A customizable intermediate level information extraction engine. *Natural Language Engineering*.

Marc Verhagen, Amber Stubbs, and James Pustejovsky. 2007. Combining independent syntactic and semantic annotation schemes. In *Linguistic Annotation Workshop (the LAW); ACL-2007*, Prague, Czech Republic, June.