Lang Resources & Evaluation (2010) 44:183–203
DOI 10.1007/s10579-010-9123-y

A framework for traversing dense annotation lattices

Branimir Boguraev · Mary Neff

Published online: 30 May 2010

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 architectures. The open-ended nature of these architectures and expressiveness of feature structure-based annotation schemes account for the natural tendency of such annotation repositories to become very dense, as multiple levels of analysis get encoded as layered annotations. This particular characteristic presents challenges for the design of a pattern matching framework capable of interpreting 'flat' patterns over arbitrarily dense annotation lattices. We present an approach where a finite state device applies (compiled) pattern grammars over what is, in effect, a linearized 'projection' of a particular route through the lattice. The route is derived by a mix of static grammar analysis and runtime interpretation of navigational directives within an extended grammar formalism; it selects just the annotations sequence appropriate for the patterns at hand. For expressive and efficient pattern matching in dense annotations stores, our implemented approach achieves a mix of lattice traversal and finite state scanning by exposing a language which, to its user, provides constructs for specifying *sequential, structural, and configurational* constraints among annotations.

Keywords AFst · UIMA · Annotation-based analytics development · Pattern matching over annotations · Annotation lattices · High density annotation repositories · Finite-state transduction · Corpus analysis

Abbreviations

UIMA	Unstructured information management architecture
FST	Finite state transduction

B. Boguraev (✉) · M. Neff
IBM T.J. Watson Research Center, Yorktown Heights, New York, USA
e-mail: bran@us.ibm.com

AFSt	Annotation-based finite state transduction
GATE	General architecture for text engineering
ULA	Unified linguistic annotation

1 Multi-faceted annotation representations

Annotation-based representation of linguistic analyses has gradually become the established mechanism for recording analysis results, across a broad range of analytical components and frameworks, and for a variety of purposes (gold standards/ground truth annotation, linguistic mark-up, formal expression of analytical output, definition of standards, to name but a few). To a large extent, the notion of annotations has driven the strong trend towards evolving robust and scalable architectures for natural language processing (Cunningham and Scott 2004). Frameworks like GATE[1], UIMA[2] and NLTK[3] (Cunningham 2002; Ferrucci and Lally 2004; Bird 2006) have wide acceptance in the community (Dale 2005), and have demonstrated the viability of feature-rich annotation structures as an expressive device, contributing to componentization and interoperability.

Bird and Liberman (2001) developed a generalized view of annotation principles and formats. Since then, representational schemes have evolved to support complex data models and multiply-layered (stacked) annotation-based analyses over a corpus. For instance, to address some issues of reusability, interoperability and portability, Hahn et al. (2007) at JULIE Lab[4] 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, stacked in layers as the number of annotation types grows to meet the representational demands of arbitrary deep analytics.

Orthogonally, initiatives like the NSF project on Unified Linguistic Annotation[5] (ULA) and the Linguistic Annotation Framework (Ide and Romary 2004) developed within ISO[6] argue for the need for annotation formats to support multiple, independent, and alternative annotation schemes; see, for instance, Verhagen et al.'s MAIS (2007) and and Ide and Suderman's GrAF (2007). In such schemes 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

[1] GATE: General Architecture for Text Engineering.

[2] UIMA: Unstructured Information Management Architecture.

[3] NLTK: Natural Language Toolkit.

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

[5] http://www.verbs.colorado.edu/ula2008/.

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

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), UIMA offers the representational backbone for the requirements of the JULIE project, as well as the ability to support (or be isomorphic to) the multiple annotation layers of MAIS and GRAF.

1.1 Dense annotation lattices

Such annotation frameworks, however, make for densely populated annotation spaces; these are, essentially, *annotation lattices*. Typically, there will be numerous annotations over the same text span. This may be the result of layering different kinds of annotation, as discussed earlier, in the case of e.g. syntactic and semantic layers. Or (as we will see in Sects. 2 and 5.4 below), this may be due to a particular characteristic of the annotation scheme: say, trees (or tree fragments) may be encoded by relative overlays of annotation spans, possibly mediated by an auxiliary system of features (properties) on annotations.

Annotations will be deposited in an annotations store by individual components of a particular processing pipeline. It may be reasonable to expect that closely related components would deposit annotations which are aligned—e.g. when later analyses operate on the output of prior annotators. Less inter-dependent components are less likely to be so well behaved. For instance, just bringing more than one tokenizer into the pipeline is certain to produce mis-aligned tokens. Similarly, combining a named entity annotator with an independently developed parsing component is certain to result in mis-alignment of boundaries of named entities and linguistic phrases, also leading to overlapping annotations.

Such multi-layered representational schemes give rise to 'tangled' lattices, which can be traversed along multiple paths representing different layers of granularity, e.g. tokens, named entities, phrases, co-referentially linked objects, and so forth. The lattices tend to become dense, as more and more levels of analysis get stacked on top of each other. This has significant implications for an operation common in annotation-based frameworks: matching (or querying) over an annotations store.[7] Typically, such an operation is mediated via a formal language (we discuss some formalisms in Sect. 3). Matching over annotations then would require interpretation of sub-graphs against an annotations lattice, assuming that somehow the right path of traversal has been chosen—right, that is, from the point of view of the intended semantics of a given pattern (query).

[7] Requesting all the text fragments which match a pattern is, conceptually, no different from querying an annotation repository for all annotations (or annotation configurations) which satisfy a certain set of constraints, themselves specified in a pattern (query).

This paper addresses some of the requirements—and underlying support—for a pattern specification 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 relationships* between annotations.

For example, *sequential constraints* might be used in an annotation pattern for some meaningful relationship between a preposition token and a noun phrase immediately following it, and the underlying interpretation mechanism needs to make sure that it finds the path through the lattice which exposes the linear order of annotation instances in question. Or, *tree matching* over a predicate-argument fragment may be used to identify the node annotation for a labeled argument—this would require awareness of how tree nodes are mapped to annotation layers. Or, annotation filtering may need to iterate over 'like' annotations—say noun phrases—excluding those which are in a certain *configurational relationship* with other annotations (e.g. not sentence-initial, in subject position only, and so forth), or which exhibit certain internal structural properties (e.g. containing a temporal expression annotation, or a named entity of a certain type). The key requirement here is to be aware of possibility of, and need for, different projections through the lattice.

Clearly, constraints like these (and the need to be sensitive to them) derive directly from the nature of a dense lattice and the properties of analyzed objects captured within. We will refer to such constraints by using terms like 'horizontal' and/or 'vertical'.

Examples like the ones above 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[8] may pose a challenge to figuring out what the 'next' token is, after the end of some higher-level constituent. Likewise, an application may need to aggregate over a set of annotation instances of a certain type—e.g. for consistency checking—before it posts a higher-level, composite, annotation (consider the constraint that an [`EmailSignature`] annotation[9] may 'contain' at least two of [`Name`], [`Affiliation`], [`Address`], or [`PhoneNumber`], in any order; or that a temporal relation[10] must have two arguments, one of which is a temporal expression).

In this work, we examine the implications for a mechanism for querying an annotations store which is known to be a dense lattice. In the next two sections we highlight some considerations to be taken into account when designing lattice traversal support, and we briefly summarize some salient points of related work addressing matching over annotations. Section 4 highlights some essential

[8] Tokens are just instances of an annotation type. Multiple tokenizers would introduce multiple token streams; it is not uncommon for complex applications to deploy multiple tokenizers, e.g. if models for different components have been trained over disparate pre-tagged text sources.

[9] The notation [`TypeName`] refers to an annotation of type `TypeName` in the text.

[10] Here, and in this paper in general, we assume that all annotations manipulated through the framework are text-consuming.

characteristics of a formal language for matching over annotations, and describes how it is adapted to the task of lattice traversal by means of a novel iteration device. It is within this formalism that we then develop the navigational elements which are the subject of this paper (Sect. 5). In essence, the language appeals to a set of finite-state operators, adapted to the task of matching against annotations and manipulating multi-faceted typed feature structures; we assume familiarity with finite-state devices and the mechanics of pattern matching, and do not detail all of the language. We conclude with some examples of applications of the framework we have developed, and highlight a couple of particularly salient extensions, as future work.

2 Challenges of lattice navigation

Revisiting the examples above, we notice that in many situations where an annotation in the store is matched against some annotation specification, it presents an inherent ambiguity. On the one hand, it may be considered as an object by itself: is it an annotation of a certain type?[11] The test can therefore be executed over a single annotation object.

Alternatively, the intent may be to consider it as an integral part of a larger structured object, the shape of which matters. Consider, for instance, a representation scheme where proper names are annotated (say, by [PName] annotations), with their internal structure made explicit by annotating, as appropriate, for [Title] and [Name], itself further broken into e.g. [First], [Middle], and [Last]. *"General Ulysses S. Grant"*, in such a rendering, would instantiate multi-layered annotations:

```
annotations:   |               PName               |
               | Title |           Name            |
               |       | First | Middle | Last |

text:          "General  Ulysses  S.      Grant"
```

For any particular instance of a [PName] found in the text, some of these annotation types may, or may not, appear (*"Max"*, *"Minister of Agriculture Stoyanov"*). The analysis relies, however, on the annotations strictly covering—but not crossing—each other. This allows the above structure to be construed as representing a tree by layering annotations, and by encoding a parent-child relationship between annotations by appropriately interpreting their relative spans: [PName] node is 'above' a [Title] node; [Name] is a parent to [First], and so on. (We note here, in passing, that such a relationship clearly cannot be inferred for two annotations with identical spans; for this, we need an additional system of type

[11] It may be the case that an annotation will have inner properties (or *features*: UIMA uses typed feature structures to represent annotations); in that case testing for a match would require checking values of features against their specifications too. Still, this operation is carried out just on the annotation itself.

priorities, and we will discuss this in detail in Sects. 4.2 and more specifically 5.4 below.)

Now, in addition to conceiving an application which just needs to iterate over [PName]s (for the purposes of an 'atomic' match), it is also possible to imagine a need for e.g. only collecting titled proper names, or proper names whose [Last] components satisfy some constraint, such as they are all of a certain nationality. The former iteration regime would only 'care for' [PName] annotations; the latter needs both to identify [PName]s *and simultaneously* to inspect their internal components which, by definition, are other annotations on a different path through the lattice. Since the different annotation layers are not connected explicitly (as tree nodes would), what is required is a 'smart iterator' which needs to be aware of the configurational properties of layered annotations.

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 have to specify the optional use of punctuation tokens, such as commas, hyphens, *etc.*—the intuition here is that a pattern language 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. An example here is an entity being tagged as (legitimately) belonging to multiple categories within a broad semantic class: a person may be an [Actor], a [Director] and a [Producer], and the annotations store would reflect this, by means of multiple co-terminous annotations over the same [PName] span. The semantics of these annotations is such that the runtime engine here would need to visit all of these, as opposed to making a choice (whether priority-based, or mandated by an explicit directive). This is complementary to most of the above scenarios, and further illustrates that a range of iteration regimes need to be supported by the annotation matching infrastructure. Obviously, part of that support would entail provisions—at the grammar language level—for specifying the kind of iteration to be carried out at runtime.

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. However, given the non-linear nature of the input and the variety of constraints on traversal regimes outlined above, our framework of annotation-based finite-state transduction (henceforth AFst) makes suitable adaptations in order to reduce the problem of traversing an annotation lattice to that of FS-tractable problem of scanning an unambiguous stream of ‘like’ objects, in our case structured 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’ (fragments of) the annotation lattice into an unambiguous annotation stream, so that pattern matching can then 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 component such as 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 the associated pattern specification language elements) facilitating such linearization.

3 Related work: patterns over annotations

Several approaches have been developed to address the problems of matching over annotations. Glossing over the details somewhat, two broad categories can be observed.

A class of systems, like those of (Grefenstette 1999; Silberztein 2000; Boguraev 2000; Grover et al. 2000; Simov et al. 2002), essentially deconstruct an annotations store data structure into a string which is suitably adorned with in-line annotation boundary information; FS (or FS-like) matching is then applied over that string. At the implementational 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 Sect. 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 architecture environments is exemplified by a different class of systems, most prominently by GATE’s JAPE (Cunningham et al. 2000) and DFKI’s SPROUT (Droẑdẑyński et al. 2004). While the two are substantially different, a crucial shared assumption is that the annotation traversal engine ‘knows’ that components upstream of it will have deposited annotations of certain type(s). This knowledge is then implicitly used during grammar application. As a result, the lattice to be traversed has certain ‘well-formed’ characteristics, stemming from the advanced knowledge of types to be encountered therein, and their relationships with each other—whether fanning out at certain ambiguous points (in the case of GATE), or within a strictly hierarchical set of type instances (in the case of SPROUT).

Operationally, an iterator behaves as an ambiguous one, examining multiple choice points in the lattice, in a pre-determined order. This is not an assumption which necessarily holds for projects like the ones outlined earlier (Sect. 1), nor does it adequately address the proliferation of possibly ambiguous, or even contradictory, annotations typical of large-scale architectures where an arbitrary number of annotator components may deposit semantically conflicting and/or configurationally partially overlapping spans in the annotations store.

Clearly, the one common theme underlying most of these approaches is the notion of adapting an annotations store so that FS-like matching can be applied to the objects in it. As we already mentioned in the previous section, such is the strategy we adopt as well. However, none of the work outlined here addresses the issues and challenges of explicitly controlling end-to-end navigation through the lattice-like structure of such annotations stores. This is the subject of the remainder of this paper.

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, and from tree traversal for structure decomposition; additionally it utilizes type prioritization for the interpretation of configurational statements. These elements 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. (Droždžyński et al. 2004), with pattern-action rules where patterns are specified over configurations of type instances, and actions 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 sequences of annotations with certain properties, e.g. nouns following determiners, unbroken stream of tokens with certain orthographic feature (such as capitalization), or noun group–verb group pairs in particular contexts.

However, given that transitions of the underlying FS automaton are mediated by a complex set of constraints, the notation incorporates additional 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 (e.g. by creating and posting new annotations, deleting or modifying existing ones, arbitrary manipulation of features and values, and promoting instances of types within the type hierarchy). Note that conditions for successful match may include filters on annotation structure components (*i.e.*its feature set), as well as contextual factors determined by other annotations to be found above, below, and around the one in the focus of the match (cf. vertical and horizontal configurational constraints, introduced in Sect. 1).

Much of the complexity is borne by a *symbol notation*, indicative of the set of operations that need to be carried out upon a transition within the transition graph compiled from the FS grammar 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 over the next annotation in its input stream. Examples of such operations, expressed as symbols on the arcs of an FS automaton, are:

- `Token[]` : straightforward match conditioned only on type, or
- `Person[kind=~"named"]` : match over an annotation of type `Person`, examining the value of its `kind` feature; license the transition only for `"named"` [`Person`]s;
- `NP[]/] Subj[passive="false"]` : match over an [`NP`]; if successful, post a new annotation, [`Subj`], with a feature named `passive` set to a string `"false"`.[12]

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 * specify, respectively, sequential composition, and zero or more repetitions, of sub-patterns. In effect, this 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 found, a new [`NP`] annotation is posted to the annotations store; its span is over the entire matching sequence of `Token`s (denoted by the matching pairs of `/[NP` and `/] NP` transduction symbols).

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

[12] Elements of the formalism which translate into posting new, or modifying existing, annotations are somewhat orthogonal to issues of navigation; we will not discuss transduction symbols or mechanisms here. We also deliberately gloss over the question of what the span of the new [`Subj`] annotation should be, but see the example grammar immediately below.

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 also incorporate dates in the noun phrase contour, e.g. by specifying `Timex [kind=~"date"]` as an additional nominal pre-modifier (*c.f. "this December 25th tradition*).

In line with similar matching frameworks, like GATE and SPROUT, both 'match' and 'transduce' operations appear as atomic transactions within a finite-state device. Matching operations are defined as subsumption among feature structures. Transduction operations create new annotations, delegating to the native UIMA annotation-posting mechanism; they also facilitate, by means of variable setting and binding, feature percolation and embedded references to annotations as feature values. By their nature and function, transductions are largely outside of the scope of this paper.

In essence, by appealing to UIMA's type system 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 type granularity,[13] 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 existing 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 in special cases), 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).

4.2 Navigational constraints

There are essentially two components to the AFst framework. The previous section outlined the pattern-based matching. In order for the patterns to be applied, however, navigation through the lattice must happen in order for a stream of annotations to be generated—the lattice gets linearized, from the perspective of the FST graph compiled from the grammar.

Both navigation and matching are by their nature runtime elements. Navigation, however, crucially requires information gathered from static grammar analysis: the set of types a grammar refers to and the configurational constraints among annotations to match—implicitly inferrable and explicitly specified via custom notational devices.

Such notational devices assert control both at symbol-matching and pattern-specification levels. In particular, we will see below how, by referencing the UIMA

[13] Patterns may refer to both type instances and supertypes: the framework will admit e.g. a `PName` annotation as an instance of a `Named` supertype specified in the grammar as a match target; supertypes thus are akin to wild cards. Note that if both [`A`] and [`B`] are defined to be subtypes of [`Element`], a pattern specification ... `Element [ ] . Element [ ]` ... would match both sequences [`A`] followed by [`B`], and [`B`] followed by [`A`]; this allows for order-independent grammars.

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 suitable provisions for control.

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. Primary among these are: start/end location, type, and type priority. This last parameter refers to the intuitive notion of specifying an ordering of types with respect to which should be returned first, when an iterator encounters multiple type annotations over the same text span; among other things, priorities among types are crucial for encoding tree-like information via annotation spans (Sect. 2; see also 5.4 below).

A key insight in our design is that a compiled transition graph specifies exactly the type of annotation required by any given transition. At points in the lattice where this is ambiguous, the notation allows to choose among alternative outgoing annotations. (There is a default interpretation, given a particular type hierarchy and system of type priorities.) Our insight thus translates into the dynamic construction of a special kind of *typeset iterator*, which is different for every grammar as it depends on the set of types over which the grammar is defined.

As a simple example, the noun phrase grammar earlier in this section tests, at all transition points, for a single annotation type: [`Token`]. Consequently, no matter how dense the annotation lattice, iterating over [`Token`]s only, in text order, would be adequate for the AFst runtime interpreter, as it tries to apply this grammar.

This typeset iterator mediates annotation lattice traversal in a fashion corresponding to threading through it a route consistent with the set of types relevant to the grammar, and no more. It is configured to emit a stream of only those annotation instances referenced in a grammar file, according to a left-to-right traversal of the annotations store, and compliant with type priorities where a fan-out point in the lattice is reached. For instance, if a grammar is concerned not just with [`Token`]s, but with, say, [`NP`] annotations as well, the question what to do at points in the lattice where instances of both kinds of annotation share a start position, is resolved by default with the iterator returning the one with higher priority (presumably, the [`NP`]); this strategy resonates with intuitions for analytics development, but it can be overriden.

Grammar-level specification of horizontal *and* vertical constraints is compiled into a particular sequence of matches over annotations. The iterator-generated stream of annotations is the input to the AFst interpreter, as it captures the annotation sequence over which the pattern is applied. This, then, is overlayed over the lattice. In effect, the typeset 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 syntax, designed to inspect the internal make-up of annotation instances. This syntax additionally incorporates part of the mechanism whereby the AFst interpreter constructs the annotations stream paired, at execution time, with the FST graph for a given grammar. Also, as we shall see below, there are iterator-directed statements in the grammar itself. In other words, the route projection discussed in the previous section, which results in a linearization of a particular path in the dense lattice, is carried out by navigational directives both at symbol and grammar notation levels. Here we look at the range of devices which select the elements of annotation lattice appropriate to present to the FS matching machinery.

As we have already described (Sect. 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 Sect. 5.1 below.

The other aspect of the navigation problem, complementary to route selection, is that of navigation control. Asserting control is, in effect, distributed among configuring a suitable UIMA iterator and using 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 notational devices in the AFst language 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*, devices exist for shifting the traversal path of the interpreter, in an 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. Therefore, 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[14] instance of a UIMA iterator, which filters

[14] At grammar load time, when the interpreter is initialized.

for a subset of types from the larger application's type system, and is configured for unambiguous traversal of the annotations store.

In the previous section, we already showed that the grammar fragment in Sect. 4.1, for example, would 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. A different grammar, for example, may conceive of relabeling [NP] annotations to the left and right of a [VG] (verb group) as [Subj] and [Obj]; this would be agnostic of [Token]s, as it would scan the annotations store for [NP] and [VG] instances only.

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 element on a transition. 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 (see Sect. 2). 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 Sect. 2, would capture—by means of explicit priority definition—statements like [PName] is above [Title] and [Name], and [Name] is above [First]/[Last]. Similarly, it is via priorities that we can capture intuitions like: [Sentence] sits 'higher' in the lattice vertical order than [Phrase], which is itself above [Token]s.[15]

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, however, behave unambiguously, according to the priority constraints above;[16] 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 application for which both an address *and* a date annotator need to be developed. Numbered tokens may be part of a street address, they also might be interpreted (within some orthographic conventions) as years. Both annotators traffic

[15] Note that, while appealing to 'common intuitions' in the interpretation of 'longer [PName] annotations stand for nodes in a tree hierarchy *above* shorter [Name] annotations' (Sect. 2), it is essential for the system's completeness and correctness that such relationships are explicitly encoded in a set of priority declarations.

[16] Assuming that [Phrase] is declared a common supertype to both [NP] and [VG].

in [Token]s. However, if there are [Address] annotations in the store already, a [Date] annotator should not 'descend' under them, to inspect [Address]-internal [Token]s: in the context of [Address] annotation over *"1600 Pennsylvania Avenue"*, there is no point in tagging *"1600"* as a [Year]; in fact, it would be wrong to do so. Yet we have seen no natural way in which date patterns might be made aware of (pre-annotated) address fragments. Still, 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, another device comes into play: a system of declarations has been developed to control both the matching and the iteration components of the framework.

With respect to the earlier example, where [Address]-internal numbered tokens need to be kept invisible to the AFst interpreter, types external to a grammar can be explicitly brought into the typeset iterator filter by means of an honour declaration:

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

Without the honour declaration, the grammar fragment above would induce a typeset iterator over [Token]s. The pattern would trigger over a fragment within the [Address] span of *" … 1650 Sunset Boulevard"*, posting [Year] over *"1650"*. The declaration adds [Address] to the typeset iterator filter; when the interpreter gets to the point in the lattice where both [Token] and [Address] annotations start at *"1650"*, the effect of the declarations will be to guide the choice according to the intent of the grammar writer, namely to prevent inspection of the [Token]s under [Address]. (We assume here that the address grammar is applied before the date grammar; see grammar cascading below.)

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 (outlined in Sect. 5.1) to trigger across the boundary of certain clause types, a "boundary % Clause[] ;" declaration sees to that. Note that there may be multiple boundary annotations.

We are now in a position to give a more precise definition of our typeset iterator. It is 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 re-targeting of grammars.

A `match` declaration controls how the iterator decides what match(es) to return as successful; usual parameters here include `"match % all;"`, or `"match % longest;"`, which is the default.

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. A `skip` directive thus does not examine right context to a prior match; the alternative (`step`) regime is useful in situations where more fine-grained context examination is essential for pattern application.

The procedural aspects of `match` and `advance` are not unfamiliar: pattern-matching systems like GATE and CPSL (Common Pattern Specification Language; Appelt and Onyshkevych 1996) appeal to similar notions. We highlight here the fact that while not directly affecting navigation *per se*, these declarations affect the iterator behavior, and thus play into the mix of devices whereby the grammar writer can fine-tune the pattern application process.

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. Cascades of grammars 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. The more annotation types referenced by a grammar, the harder for a grammar writer it is to anticipate conflict situations with multiply-layered annotations, which would require explicit navigational control through the grammar (as described in Sect. 5.4 below). Conversely, smaller grammars lead to iterators with smaller number of types in their filter; this, in its own turn, eases the grammar writer's burden of having to be explicit about lattice traversal.

Separating the patterns which target related 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 as separate grammars, 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 particular pattern over its components. In Sects. 5.1 and 5.2 we saw how to point the typeset iterator at the higher, or lower, level of traversal. Here, we introduce another special purpose iterator: a *mixed iterator*, 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* (cf. `[PName]` in Sect. 2). We already saw examples of this, such as collecting titled proper names only, or proper names whose `[Last]` components satisfy some constraint (Sect. 2), or matching on noun phrases with temporal premodifiers (Sect. 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 effect, this would require earlier (upstream) annotators to 'anticipate' the kinds of queries to be posed later—and 'cache' the answers as feature values on the annotation they post.

However, in an environment 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 directive:

- test for an annotation of a certain type, with or without additional constraints on its features;
- upon a successful match, *descend* under 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 immediately *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 sub-iterator is instantiated with the lower types in its filter, and the next lower level is linearized for execution.

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 under' operation. (The matching `@descend` and `@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 configurations 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 if 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 (see Sect. 2) defines the higher node of the two in the sub-tree.

Remembering the tree structure implied by the lattice configuration for *"General Ulysses S. Grant"* (Sect. 2), the following expression encodes, in effect, a query against the set of `PName` trees in the database, which will match all proper names of the form *"General … Grant"*:

```
findG = PName[@descend] .
          Title[string=="General"] .
          Name[@descend] .
            First[]|<E> . Middle[]|<E> . Last[string=="Grant"] .
          Name[@ascend] .
        PName[@ascend] ;
```

In a number of situations, inspecting configurational properties of the annotation 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. 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 [`Token`]s, the second tests if there is a proper name ([`PName`]) within the span of a [`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 in advance) number of intermediate layers.

6 Conclusion

This paper focuses largely on support for navigating through 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, and 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 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 (Boguraev and Ando 2005).

The bulk of the grammar formalism evolved from the requirements of 'linear' pattern specification. It is, however, 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, that have informed extensions of the formalism to do specifically with lattice traversal, and have 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 due to proliferation of semantic types in rich domains (we outlined this scenario in the [`Actor`]/[`Director`]/[`Producer`] example in Sect. 2), the density of the annotation lattice is very high. A strictly unambiguous iteration regime—with its requisite needs for up/down attention shifts and priority specification—may not be the optimal way to search through an annotations store. 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 cannot 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 runtime system 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) are 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). This appears similar in spirit to JAPES iteration regime (Sect. 3); there are differences, however, mainly in the fact that we still require the type filtering, in order to control single-path traversal of the lattice outside of the areas where ambiguous regime makes sense—this is necessary to deal with situations where conflicting annotation layers have been deposited by upstream annotators.

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 ability to cascade grammars 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 Sects. 2 and 5.4). 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.

Finally, we note that the framework described here operates over text-consuming annotations. Not all annotations-based representational schemes are bounded by such an assumption. For instance, recent work on identifying relations in unstructured text tends to represent a relation among two (or more) entities in the text as a feature structure with references to annotations, not necessarily spanning any text itself. While references to annotations can be captured and manipulated in AFst, it will need to be extended to handle non-consuming (zero-length) annotations.

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.

References

Appelt, D. E., & Onyshkevych, B. (1996). The common pattern specification language. In *Proceedings of a workshop held at Baltimore, Maryland* (pp. 23–30). Morristown, NJ, USA: Association for Computational Linguistics.

Bird, S. (2006). NLTK: The natural language toolkit. In *Demonstration session, 45th annual meeting of the ACL*. Sydney, Australia.

Bird, S., Buneman, P., & Tan, W.-C. (2000). Towards a query language for annotation graphs. In *Second international language resources and evaluation conference*. Athens, Greece.

Bird, S., & Liberman, M. (2001). A formal framework for linguistic annotation. *Speech Communication, 33*(1–2), 23–60.

Boguraev, B. (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.

Boguraev, B., & Ando, R. K. (2005). TimeML-compliant text analysis for temporal reasoning. In *Nineteenth international joint conference on artificial intelligence (IJCAI-05)*. Edinburgh, Scotland.

Boguraev, B., & Neff, M. (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.

Cassidy, S. (2002). XQuery as an annotation query language: A use case analysis. In: *Third international language resources and evaluation conference*. Las Palmas, Spain.

Cunningham, H. (2002). GATE, a general architecture for language engineering. *Computers and the Humanities, 36*(2), 223–254.

Cunningham, H., Maynard, D., & Tablan, V. (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.

Cunningham, H., & Scott, D. (2004). Software architectures for language engineering. Special Issue. *Natural Language Engineering, 10*(4).

Dale, R. (2005). Industry watch. *Natural Language Engineering, 11*, 435–439.

Droẑdẑyński, W., Krieger, H.-U., Piskorski, J., Schäfer, U., & Xu, F. (2004). Shallow processing with unification and typed feature structures—Foundations and applications. *Künstliche Intelligenz*, (1), 17–23.

Ferrucci, D., & Lally, A. (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.

Grefenstette, G. (1999). Light parsing as finite state filtering. In A. Kornai (Ed.), *Extended finite state models of language, studies in natural language processing*, (pp. 86–94). Cambridge UK: Cambridge University Press.

Grover, C., Matheson, C., Mikheev, A., & Moens, M. (2000). LT-TTT: A flexible tokenisation tool. In *Proceedings of the second international conference on language resources and evaluation*, (pp. 1147–1154). Spain.

Hahn, U., Buyko, E., Tomanek, K., Piao, S., McNaught, J., Tsuruoka, Y., & Ananiadou, S. (2007). An annotation type system for a data-driven NLP pipeline. In *Linguistic annotation workshop (the LAW); ACL-2007*. Prague, Czech Republic.

Ide, N., & Romary, L. (2004). International standard for a linguistic annotation framework. *Natural Language Engineering, 10*(4). Special Issue on Software Architectures for Language Engineering.

Ide, N., & Suderman, K. (2007). GrAF: A graph-based format for linguistic annotation. In *Linguistic annotation workshop (the LAW); ACL-2007*. Prague, Czech Republic.

Lai, C., & Bird, S. (2004). Querying and updating treebanks: A critical survey and requirements analysis. In *Australasian language technology workshop*. Sydney.

Park, Y., Byrd, R., & Boguraev, B. (2002). Automatic glossary extraction: Beyond terminology identification. In *Proceedings of the 19th international conference on computational linguistics (COLING)*, (pp. 772–778). Taiwan.

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

Simov, K., Kouylekov, M., & Simov, A. (2002). Cascaded regular grammars over XML documents. In *Proceedings of the second international workshop on NLP and XML (NLPXML-2002)*. Taipei, Tawian.

Srihari, R. K., Li, W., Cornell, T., & Niu, C. (2008). InfoXtract: A customizable intermediate level information extraction engine. *Natural Language Engineering*.

Verhagen, M., Stubbs, A., & Pustejovsky, J. (2007). Combining independent syntactic and semantic annotation schemes. In *Linguistic annotation workshop (the LAW); ACL-2007*. Prague, Czech Republic.