**RC23393 (W0410-163) October 25, 2004**
**Computer Science**

# IBM Research Report

## Annotation-Based Finite State Processing in a Large-Scale NLP Architecture

**Branimir K. Boguraev**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

# Annotation-Based Finite State Processing in a Large-Scale NLP Architecture

BRANIMIR K. BOGURAEV

# Annotation-Based Finite State Processing in a Large-Scale NLP Architecture

BRANIMIR K. BOGURAEV

*IBM T.J. Watson Research Center*

### Abstract

There are well-articulated arguments promoting the deployment of finite-state (FS) processing techninques for natural language processing (NLP) application development. This paper adopts a point of view of designing industrial strength NLP frameworks, where emerging notions include a pipelined architecture, open-ended intercomponent communication, and the adoption of linguistic annotations as fundamental analytic / descriptive device. For such frameworks, certain issues arise — operational and notational — concerning the underlying data stream over which the FS machinery operates. The paper reviews recent work on finite-state processing of annotations and highlight some essential features required from a congenial architecture for NLP aiming to be broadly applicable to, and configurable for, an open-ended set of tasks.

## 1 Introduction

Recent years have seen a strong trend towards evolving a notion of robust and scalable architectures for natural language processing (NLP). A quick browse through, for instance, (Cunningham 2002, Ballim & Pallotta 2002, Patrick & Cunnigham 2003, Cunnigham & Scott 2004), shows a broad spectrum of engineering issues identified within the NLP community as essential to successful development and deployment of language technologies. Without going into any detail, it is illustrative to list a representative sample of them.

With growing use of XML as data modeling language and analysis interchange transport, *uniform document modeling* addresses concerns of widely variegated natural language text sources and formats; this includes, as we will see, strategies for representing analysis results in XML too. For incremental development and reusability of function, *componentised architecture* is becoming the accepted norm. Overall system reconfigurability is facilitated by *component inter-operability*; components are managed either by integrating them within a *framework*, or by exporting functionality packaged in a *toolkit*. Broad coverage, typically requiring streamlined utilisation of background knowledge sources, is achieved by *transparent access to resources*, using common protocols.

Such principles of design, while beneficial to the overall goal of building usable systems, pose certain constraints on system-internal data structures and object representation, and ultimately affect the algorithmic embodiment(s) of the technologies encapsulated within.

A core characteristic of present day software architectures is that they all appeal to a notion of *annotation-based representation*, used to record, and transmit, results of individual component analysis. (For consistency, and to reinforce the uniformity of annotation-based representation, we will refer to such architectural components as "annotators".) A detailed description of general-purpose annotation formats can be found in (Bird & Liberman 2001). Specific guidelines for a linguistic annotation framework have recently been formulated by Ide & Romary (Forthcoming); for a representative sample of architecture-level considerations, and motivations, for adopting annotations as the fundamental representational abstraction, see, for instance, (Grishman 1996), and more recently, (Cunningham et al. 2002, Bunt & Romary 2002, Neff et al. Forthcoming).

The case for annotations has been made, repeatedly, and any of the publications cited above will supply the basic arguments, from an architectural point of view. There are, however, additional arguments which derive from observing a larger set of contexts in which NLP architectures have been put to use. Overall, these are characterised by situations where a particular technology needs to be used well outside the environment where it was developed, possibly by non-experts, maybe even not NLP specialists. Examples of such situations can easily be found in large research laboratories, and in corporate environments where technologies developed in the laboratories are incorporated in custom 'solutions'. Ferrucci & Lally (Forthcoming) describe an architecture for unstructured information management, which is largely informed by considerations of smooth exchange of research results and emerging technologies in the NLP area. Such an architecture would be deployed, and would require to be configured, within the organisation, not only by hundreds of researchers who will need to understand, explore, and use each other's results, but by non-specialists too.[1] It is in contexts like these that questions

[1] A recent article in *The Economist* (June 19th, 2003), "*Is Big Blue the Next Big Thing?*", makes a related point that 'front-line' users of emerging technologies — including NLP — are information technology specialists and consultants who would be work-

of uniformity, perspicuity, and generality of the underlying representation format become particularly relevant.

Annotations — when adorned with an appropriate feature system (Cunningham et al. 2002), and overlayed onto a graph structure (Bird & Liberman 2001) — combine simplicity of conceptualisation of a set of linguistic analyses with representational power (largely) adequate for capturing the intricacies of such analyses. The question still remains, however, of providing a capability (within the architecture) for exploring this space of analyses. As we will argue below, finite-state (FS) matching and transduction over annotations provides such capability, together with flexibility and convenience of rapidly configuring custom analysis engines which can use any, and all, analyses derived by a diverse set of upstream annotators.

There are yet other considerations arising specifically from concerns of industrial-strength NLP environments including, for instance: efficiency, both of the prototyping process, and at run time; reconciling different I/O behaviour from different, yet functionally identical, components; and ability to layer progressively more complex linguistic analyses on top of simpler annotation mark-up.

Typically, and even more so recently, when efficiency is brought up as a concern, the answer more often than not is "finite-state technology". This is not surprising, as many well-articulated arguments for finite-state processing techniques focus largely on the formal properties of finite-state automata, and the gains in speed, simplicity, and often size reduction when an FS device is used to model a linguistic phenomenon (Karttunen et al. 1996, Kornai 1999). Much rests on a particularly attractive property of FS automata, namely that after combining them in a variety of ways, the result is guaranteed to be an FS automaton. The combinations are described by regular expression patterns (van Noord & Gerdemann 2000, Beesley & Karttunen 2003), and the use of such patterns in NLP is becoming the *de facto* abstraction for conceptualising (and using) finite-state subsystems. This is reinforced by the clean separation of the algorithmic aspects of FS execution (which are hidden from the user), and the purely declarative statement(s) of configurational patterns which underlie a linguistic analysis.

In the same spirit of observing that "*there is a certain mathematical beauty to finite state calculus*" (Beesley & Karttunen 2003), there is also

ing with customers on developing solutions that incorporate these technologies.

beauty in the ease and perspicuity of rule writing (especially if rules appeal to familiar regular expression notational conventions). If a rule-based system designed on top of FS principles, were to target an annotations store, it would facilitate both the exploration of analyses space and rapid configuration of 'consumers' of the analyses inside an arbitrarily configured instance of an NLP architecture. It is this observation alone that ultimately underpins the notion of finite-state processing over arbitrary annotations streams. It is also the case, however, that manipulating annotations within a finite-state framework offers transparent solutions to many problems referred to above, like reconciling outputs from different annotators and constructing, bottom-up and incrementally, elaborate linguistic analyses.

These kinds of observations motivate the use of FS technology in a number of contemporary NLP toolkits and architectures. The following sections discuss a variety of situated frameworks which incorporate FS techniques.

## 2 Finite-state technology and annotations

Broadly speaking, FS technology can be embedded in an NLP architecture in one of two ways. Conventionally, a functional component inside the architecture would be built on top of a generic finite-state transduction (FST) toolkit. As an example, consider the function of a part-of-speech (POS) tagger, a fairly common tool in most text processing pipelines. Both (Roche & Schabes 1995) and (Kempe 1997), for example, develop methods for modeling the tag disambiguation task by means of a finite-state device. Tagging, of course, is not the only function which can be realised through FS methods: see, for instance, (Pereira & Wright 1997) for FS approximations to phrase structure grammars, or (Abney 1996) and (Ait-Mokhtar & Chanod 1997) for finite state approaches to syntax.

Such encapsulation of FS technology contributes very little, architecturally, to a general purpose capability for manipulating annotations by means of FST. The fact that an FS toolkit is used to implement a particular component function does not necessarily make that toolkit naturally available to all components in the architecture. Of more interest to this discussion are the methods developed for packaging an FS toolkit as a generic annotator inside of an NLP architecture, a module which traffics in annotations and which can be configured to perform annotation matching at any point of a pipeline.

The examples above are realised, ultimately, as character-based FST systems. If these realisations were to be packaged for incorporation in an architectural pipeline, certain transformations would need to be carried out at the interfaces between the overall data model — e.g., of a token / lemma / POS internal object — and a character string encoding (later in this section we elaborate on general strategies for, and shortcomings of, such mappings).

Furthermore, while finite-state operations are defined over an (unambiguous) stream of input, conventionally characters, the set of annotations at any arbitrary point of a processing pipeline is typically organised as a lattice, with multiple ways of threading a passage through it. To the extent that an FS subsystem explicitly employs a notation for specifying FS rules over annotations, such as for instance that in (Wunsch 2003), the question of picking a particular route through the lattice is not addressed at all (we will return to this in Section 3 below).

Cunningham et al. (2000) claim that notwithstanding the non-determinism arising from such a traversal, "*...this is not the bad news that it seems to be....*" Maybe so, but the reason is due to the fact that many grammars targeting certain annotations configurations tend to be written to exploit a carefully designed system of annotations which model relatively flat analyses. The statement will not necessarily hold for grammars written to explore an annotations store, without prior knowledge of which annotator deposited what in there. Also, in 'tightly designed' systems annotations tend to be added, monotonically, on top of existing annotations, with longer spans neatly covering shorter ones. When such a configurational property holds, it is possible to design a grammar in a way which will match against all annotations in the (implied) tree. This is, in fact, a crucial assumption in systems like Abney's (1996) CASS and SRI's FASTUS (Hobbs et al. 1997).

Such an assumption, however, will not necessarily be true of systems where a diverse set of annotators may populate the annotations store with partially overlapping and / or non-contiguous annotations. Note that a number of componentised, distributed architectures used in diverse application environments (as the ones discussed in Section 1) fall in that category.

Still, it is not uncommon to use a character-based FS toolkit to explore some annotation spaces. In essence, the basic idea is first to flatten an annotation sequence into a character string, 'exporting' relevant properties of the annotations into the string image, and then run suitably configured transducers over that string. Matching over an annotation of a certain type would be by means of defining a 'macro', which is sensitive to the string characterisation of the relevant annotation type (label) and property (or properties): thus, for example, test for a POS tag of a token annotation could be realised as a regular expression skipping over what are known to be the token characters and focusing instead on the tag characters, identified by, say, some orthographic convention (see below). A successful match, when appropriate, would insert a pair of begin-end markers bracketing the span, possibly adorned with a label to indicate a newly created type. New types can be enriched with features, with values computed during the composition of the subsumed types. Higher-level abstractions can be created then by writing rules sensitive to such phrase markers and labels. After a sequence of transducers has run, the resulting string can be 'parsed' back into the annotations store, to absorb the new annotations and their properties from the markers, labels, and feature-values.

Without going into details, the following illustrates (assuming some relatively straightforward regular grammars defining contours for noun and verb groups) the result of a transduction which maps a string derived from part-of-speech annotations over text tokens into a string with demarcated syntactic noun and verb groups, with instantiated morphosyntactic properties.

```
"The/DT Russian/JJ-C-S executive/JJ-C-S branch/NN sees/VB+3S an/DT
 opportunity/NN to/TO show/VB+I Russia/NP 's/POS relevance/NN"

"[NG:sng The/DT Russian/JJ-C-S executive/JJ-C-S branch/NN NG]
 [VG:+3S sees/VB+3S VG] [NG:sng an/DT opportunity/NN NG]
 [VG:inf to/TO show/VB+I VG]
 [NG:sng:poss Russia/NP 's/POS relevance/NN NG]"
```

This kind of process builds upon ideas developed in (Grefenstette 1999)[2] and (Ait-Mokhtar & Chanod 1997), with the notion of special 'markers' to constrain the area of match originally due to Kaplan & Kay (1994). Boguraev (2000) develops the strategy for using the transduction capabilities of a character-based FS toolkit (INTEX; Silberztein 1999) inside of a general-purpose NLP architecture with an abstract data model, in mediating between an annotations store and a character string.

A number of problems arise with this approach. Some affect adversely the performance (which is an issue for successful deployment):

[2] Grefenstette's term for this is "*light parsing as finite-state filtering*."

character input / output is massively inefficient, and (logical) match against a high-level constituent — consider, in the above example, rules to match an NG or a VG — requires scanning over long character sequences, with cumulative expense costs. Other problems arise from cumbersome overheads: decisions need to be made concerning the (sub-)set of features to export; extra care has to be taken in choosing special characters for markers[3] which will be guaranteed not to match over the input; custom code has to be written for parsing the output string and importing the results into the annotations store, being especially sensitive to changes to its initial (pre-transduction) state; no modifications should be allowed to the character buffer underlying the input text, as chances are that *all* annotations in the system are defined with respect to the original text.

Fundamentally, however, this approach breaks down in situations where the annotations are organised in a lattice, as it requires committing to a particular path in advance: once an annotation sequence has been mapped to a string, traversal options disappear. Also, to the extent that ambiguity in the input is to be expected, a mapping strategy like the one outlined above can handle category ambiguity over the same input segment (such as part-of-speech ambiguities over tokens), but not different partitioning of the input, with segments of different length competing for a ‘match’ test at any point in the lattice traversal.

## 3 Pattern matching over annotations

The strategy for adapting a character-based FS framework to an annotations store outlined in the previous section offers a way of ‘retro-fitting’ FS operations into an annotations-based architecture. As we have seen, this does not meet the goal of having an infrastructure which, by design, would treat structured annotations as input ‘symbols’ to a finite-state calculus, and would be programmable to control the non-determinism arising from the lattice-like nature of an arbitrary annotations store.

A number of recent architectural designs incorporate a pattern matching component directly over annotations. Typically, this operates as a rule-based system, with rules’ left-hand-side (and, possibly, right-hand-side too; see 3.3 below) specifying regular expressions over annotation sequences. The recognition power of such systems is, therefore, no greater

[3] In contrast, consider Kaplan & Kay’s "<" and ">", which are truly special, as they are external to the alphabet.

than regular.

Annotations are assumed to have internal structure, defined as clusters of property-value pairs. (As we will see later, where special purpose notations are developed to define annotation 'symbols' to a grammar rule, annotation structures tend to be flat, without embedding of lower level annotations as values to properties of higher level ones.) Both annotations and properties can be manipulated from the rules. Appealing to the formal notion of composition of FST's, patterns (or pattern grammars) can be phased into cascades. The underlying interpretation engine can be instructed to operate in a variety of matching / control regimes.

### 3.1 *Finite-state cascades: Background*

Organising grammars in sequences (cascades) for finite-state parsing is largely associated with the work of Hobbs et al. (1997) and Abney (1996), but finite-state methods were applied in this domain as early as 1958 (Joshi & Hopely). Abney's CASS develops a notation for regular expressions directly over syntactic categories, specifically for the purpose of partial parsing by finite-state cascades; his rewrite rules are thus defined in syntactic terms, and there is only a conceptual mapping between a grammar category and a linguistic annotation. CASS is not an architecture for NLP, and it would be hard to argue that the notation would generalise. The FASTUS system, on the other hand, as developed by Hobbs et al. is closer to the focus of this paper: the TIPSTER framework motivated one of the earlier definitions of annotations-based substrate of an NLP architecture (Grishman 1996), and within that framework, the insights gained from developing and configuring FASTUS were incorporated in the design of a Common Pattern Specification Language (CPSL; Cowie & Appelt 1998).

CPSL evolved as a pattern language over annotations, and does not fully map onto a functionally complete finite-state toolkit. In particular, there is the question whether the formalism is declarative: the language allows for function invocation *while* matching. Also, the provisions for specifying context as pre- and post-fix constraints, coupled with the way in which rulesets are associated with grammar 'phases' (a ruleset for phase is considered as a single disjunctive operation), suggests that there is no compilation of a single automaton for the entire ruleset. Furthermore, there is no notion of *transduction*, as an intrinsic FS operation; instead, similar effect is achieved by binding matched annotations, on the left hand side (LHS), and using the bound annotations on the right hand side (RHS).[4] However, the decoupling of binding from subsequent use, and the non-declarative nature of rules, additionally compounded by some syntactic ambiguity in the notation, introduces semantic problems for caching annotations.

Altogether, CPSL is tailored to the task of matching over linear sequences of annotations (as opposed to exploring tree-shaped annotation structures) and layering progressively more structured annotations over simpler ones; even so, there are limits to the complexity (or expressiveness) of the system: annotations cannot be values to other annotations' attributes.

For a while, CPSL was widely available[5], thus promoting the idea of finite-state operations over annotations with some internal structure, without having to 'bolt' such operations on top of popular character-based FST toolkits. For some interesting extensions to CPSL see, for instance, the BRIEFS project (Seitsonen 2001): regular expression matching over tokens is added to the language (useful if the architecture does not support tokenisation), wild carding over category / annotation attributes is supported, and most importantly, the need for matching over trees is motivated, and defined (even if in somewhat rudimentary form) as an operation within the notation.

### 3.2 *Matching over annotations mark-up*

A recent trend in NLP data creation and exchange — the use of explicit XML markup to annotate text documents — offers a novel perspective on structured annotation matching by finite-state methods. XML naturally traffics in tree structures, which can be viewed as explicit representations of text annotations layered on top of each other. To the extent that a system can be assumed not to require partially overlapping, or stand-off, annotations, it is possible to make use of 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. In such an approach, annotation properties are encoded via attributes; the tree configuration also supports an encoding whereby annotations can be, in effect, viewed as properties of other (higher level) annotations. It is not too hard then to conceive of a combination of some finite-state machin-

[4] CPSL does not provide a facility for deleting annotations either.

[5] Courtesy of Doug Appelt: `http://www.ai.sri.com/~appelt/TextPro/`

ery, tailored to annotations, with a mechanism for traversing the XML tree as enabling technologies for flexible matching over annotation sequences — and, indeed, tree shapes — in an annotations store.

This is the insight underlying the work of Grover et al. (2000) and Simov et al. (2002). In essence, the notion is to develop a framework for defining and applying cascades of regular transducer grammars over XML documents. The components of such a framework are an XML parser, a regular grammar interpreter, a mechanism for cascading grammars, a set of conventions of how to define and interpret an input stream for a particular grammar in a cascade, and a way of traversing the document tree so as to identify and focus on the next item in that stream.

Grover et al.'s approach emphasises the utility of a general purpose transducer, `fsgmatch`, for analysis and transformations over XML documents. `fsgmatch` uses a grammar notation defined to operate over XML elements, viewing them both as atomic objects (manipulated as annotation configurations) and as strings (as targets to regular expression matches). The notation further incorporates primitives for identifying elements of interest, at a specific level and position in the XML tree, and for specifying how a new XML markup element — in effect, a new annotation — is to be created from the components of a successful match. `fsgmatch` is embedded, with other tools, inside of a toolkit for encoding and manipulating annotations as XML markup over documents; the tools operate over XML files, and are combined in a pipeline. One particular rendering of such a pipeline can be viewed as a cascade of regular grammars. A core part of the toolkit is a query language which mediates, external to any tool, the target element / sub-tree which constrains the grammar application.

Even if somewhat rudimentary in this particular approach, the introduction of such a query language in an annotations-matching framework marks an important extension of the notion of FS processing over annotations: in addition to allowing querying of left / right context for a rule (via a mechanism of 'constraints'), rules can also be made sensitive to (some of the) 'upper' context of the element of interest (by targeting a grammar to a subset of all possible sub-trees in the XML document).

Simov et al.'s CLaRK system facilitates corpus and processing, similarly starting from the position that linguistic annotations are to be represented by means of XML markup. Like `fsgmatch`, the regular grammars in CLaRK operate over XML elements and tokens; unlike `fsgmatch`, CLaRK offers a tighter integration of its FS operations. This is manifested in the uniform management of hierarchies of token (and element) types, and in the enhancement of the underlying finite-state engine for regular grammar application and interpretation, with native facilities for composing grammars in a cascade and for navigating to the element(s) of interest to any particular grammar (rule). CLaRK's insight is to use XPath — a language for finding elements in an XML tree — as the mechanism both for specifying an annotation of interest to the current match, and for mapping that into an arbitrarily detailed projection of that annotation and its properties: a projection which can take the form of one or more character string sequences.

Still, this kind of mapping is established outside of the grammar; in order to specify a rule correctly, one needs to have detailed grasp of XPath to be able to appreciate the exact shape of the element and value stream submitted to the grammar. Furthermore, since the input stream to a rule is now a sequence of strings, a grammar rule is far from perspicuous in conceptualising the precise nature of the match over the underlying annotation.

And that, fundamentally, is the problem with matching over annotations represented by XML: at its core, the operation is over strings, and similarly to the approach described in Section 2, it requires making explicit, in a string form, a particular configuration and contents of annotations arising in a document processing pipeline at any moment of time. In addition to opaque notational conventions, this calls for repeated XML parsing, which may turn out to be prohibitively expensive in large scale, production-level NLP architectures. And even if the XML-related processing overheads were to be put aside, the fact that only non-overlapping, strictly hierarchical, and in-line annotations can be rendered for matching is limiting in itself.

### 3.3 *Matching over structured annotations*

A different class of systems, therefore, explicitly address the issues of overlaying finite-state processing technology on top of structured annotations which are 'first-class citizens' in their respective architecture environments. The best known of these is GATE's JAPE (Java Annotation Patterns Engine; Cunningham et al. 2000). Derivative of CPSL, JAPE nonetheless stands in a class of its own: primarily because the architecture it is embedded in promotes — in a systematic and principled way — the notion of annotations as structured objects (Cunningham et al. 2002). Like

CPSL, JAPE has notions like matching over linear sequences of annotations, where annotations are largely isomorphic to a broadly accepted format[6] (Bird et al. 2000); manipulating annotations by means of named bindings on the LHS of rules; and querying of left / right context. Beyond CPSL's core set of facilities, JAPE tightens up some ambiguitites in the earlier notation specification, provides utilities for grammar partitioning and rule prioritising, allows for specification of different matching regimes, and encourages factoring of patterns by means of macros.

JAPE acknowledges the inherent conflict between the descriptive power of regular expressions and the larger complexity of an annotations lattice; this is characteristically reconciled by separating RHS activation from LHS application,[7] and by assuming that components 'upstream' of JAPE will have deposited annotations so that the lattice would behave like a flat sequence. As already discussed (Section 1), such an assumption would not necessarily hold in large-scale architectures where arbitrary number of annotators may deposit conflicting or partially overlapping spans in the annotations store. To a large extent, JAPE's infrastructure (e.g., the mechanisms for setting priorities on rules, and defining different matching regimes) is intended to minimise this problem. Additionally, the ability to execute arbitrary (Java) code on the RHS is intended to provide flexible and direct access to annotations structure: operations like attribute percolation among annotations, alternative actions conditioned upon feature values, and deletion of 'scratch' annotations are cited in support of this feature. These are clearly necessary operations, but there are strong arguments to be made (see, for instance, the discussion of CPSL earlier in this section, 3.2) against dispensing with the declarative nature of a formalism, especially if the formalism can be naturally extended to accomodate them 'natively' (see Section 4 below).

A particularly effective mix of finite-state technology and complex annotation representations is illustrated by DFKI's SPPC system (Shallow Processing Production Center; Neumann & Piskorski 2000, 2002). It treats annotations with arbitrarily deep structures as input 'atoms' to a finite-state toolkit, derived from a generalisation of weighted finite-state automata (WFSA) and transducers (WFST; Mohri 1997). The toolkit func-

[6] Annotation components include a type, a pair of pointers to positions inside of the document content, and a set of attribute-value pairs, encoding linguistic information. In GATE, attributes can be strings, values can be any Java object.

[7] While maintaining recognition power to not beyond regular, this characterises JAPE as a pattern-action engine, rather than finite-state transduction technology.

tionality (Piskorski 2002) has been expanded to include operations of particular relevance to the realisation of certain text processing functions as finite-state machines, such as, for instance, Roche & Schabes' (1995) algorithm for local extension, essential for Brill-style deterministic finite-state tagging.

The breadth of the toolkit allows principled implementation of component functions in a text processing chain: SPPC's tokeniser, for instance, defers to a token classifier realised as a single WFSA, itself derived from the union of WFSA's for many token types. By appealing directly to library functions exporting the toolkit functionality, the tokeniser produces a list of token objects: triples encapsulating token start/end positions and its type.

SPPC encapsulates a cascade of FSA's/FST's, whose goal is to construct a hierarchical structure over the words in a sentence. The FS cascade naturally maps onto levels of linguistic processing: tokenisation, lexical lookup, part-of-speech-tagging, named entity extraction, phrasal analysis, clause analysis. The automata expect a list of annotation-like objects — such as the token triples — as input, and are defined to produce a list of annotation-like objects as output. The granularity and detail of representation at different levels are, however, different: the lexical processor, for instance, overlays a list of lexical items on top of the token list, where a lexical item (annotation) is a tuple with references to its initial and final token objects.

Thus, each cascade level is configured for scanning different lists, and employs predicates (on its automata transition arcs) specific to each level. The hierarchical structure constructed over the sentence words is maintained explicitly by means of references and pointers among the different level lists. In fact, Neumann & Piskorki (2002) describe SPPC as operating over a complex data structure called *text chart*, and only upon abstracting away from SPPC's details it is possible to imagine this structure being conceptually equivalent to an annotations store. It is important to realise, however, that SPPC is not a text processing architecture, with uniform concept of annotations and annotation abstractions specified on arcs of FS automata. Instead, SPPC is an application, configured for a particular sequence of text processing components with a specific goal in mind, and whose custom data structure(s) are exposed to an FST toolkit by appropriately interfacing to its library functions. In other words, while SPPC demonstrates the elegance inherent to manipulating annotations by means of FS operations, at the same time it falls short of encapsulating this in a framework which, by appealing to an annotations-centric notation would enable the configuration of arbitrary cascades over arbitrary annotation types.

Similar shortcoming can be observed of a recent system, also custom-tailored for a class of information extraction applications (Srihari et al. 2003). Like SPPC, InfoXtract maintains its own custom data structure, a *token list*, which in reality incorporates a sequence of tree structures over which a graph is overlaid for the purposes of relational information. There is clearly an interpretation of the token list as an annotations store (ignoring, for the moment, the relation encoding). For the purposes of traversing this data structure, a special formalism is developed, which mixes regular with boolean expressions. The notion is not only to be able to specify 'transductions' over token list elements, but also to have finer control over how to traverse a sequence of tree elements. This is what sets InfoXtract apart: grammars in its formalism are compiled to *tree walking automata*, with the notation providing, in addition to test/match instructions, direction-taking instructions as well.

Certain observations apply to the approaches discussed in this section. Not all configurations of annotations can be represented as hierachical arrangements: consequently, the range of annotation sequences that can be submitted to an FS device is limited. Attempts for more complex abstractions typically require ad-hoc code to meet the needs for manipulating annotations and their properties; even so, there are certain limitations to the depth of property specification on annotations. This leads to somewhat simplistic encapsulating of an 'annotation' abstraction, which even if capable of carrying the representational load of a tightly coupled system, may be inadequate for supporting co-existing — and possibly conflicting — annotations-based analyses from a multitude of 'third party' annotators. While most systems cater for left/right context inspection, examination of higher and/or lower context is, typically, not possible. In general, there is no notion of support for directing a scanner in choosing a path through the annotations lattice (apart from, perhaps, extra-grammatical means for choosing among alternative matching regimes).

## 4 A design for annotation-based FS matching

A particular design described here seeks to borrow from a combination of approaches outlined in the previous section; the design is, however, driven by the requirements of robust, scalable NLP architecture (as presented in Section1), and described in (Neff et al. Forthcoming). Briefly, the TALENT system (Text Analysis and Language ENgineering Tools) is a componentised architecture, with processing modules (annotators) organised in a (reconfigurable) pipeline. Annotators communicate with each other only indirectly, by means of annotations posted to, and read from, an annotation repository. In order to strictly maintain this controlled mode of interaction between annotators, the document character buffer logically 'disappears' from an annotator's point of view. This strongly mandates that FS operations be defined over annotations and their properties.

Essentially, annotations encapsulate data modeled as a typed feature system. Types are partitioned into families, broadly corresponding to different (logical) levels of processing: tag markup (e.g., HTML / XML), document structure, lexical (tokens or multi-words), semantic (e.g., ontological categories), syntactic, and discourse. Features vary according to type, and the system supports dynamic creation of new types and features.

Any configuration of annotations over a particular text fragment is allowed, apart from having multiple annotations of the same type over the same span of text. Annotations of different types can be co-terminous, thanks to a priority system which makes nesting explicit, and thus facilitates encoding of tree structures (where appropriate).

A uniform mechanism for traversing the annotation repository is provided by means of custom iterators with a broad set of methods for moving forward and backward, from any given position in the text, with respect to a range of ordering functions over the annotations (in particular, start or end location, and priority). In addition, iterators can be 'filtered' by family, type, and location. As a result, it is possible to specify, programmatically, precisely how the annotations lattice should be traversed. This underlies one of the distinguishing features of our FST design: rather than rely exclusively (as, say, JAPE does) on specifying different control regimes in order to pick alternative paths through a lattice, we provide, inside of the FS formalism, notational conventions for directing the underlying scan (which exploit the iterators just described). This is similar to InfoXtract's notion of directive control, but broadly defined in terms of annotation configurations, uncommited to e.g., a tree structure.

Finite-state matching, as a system-level capability, is provided by packaging FS operations within a (meta-)annotator: TALENT's FS transducer (henceforth TFST) encapsulates matching and transduction capabilities and makes these available for independent development of grammar-based linguistic filters and processors. TFST is configurable entirely by means of external grammars, and naturally allows grammar composition into sequential cascades. Unlike other annotators, it may be invoked more than once. Communication between different grammars within a cascade, as well as with the rest of the system (including possible subsequent TFST invocations) is entirely by means of annotations; thus, in comparison to mapping annotations to strings (cf. Sections 2 and 3 above), there are no limitations to what annotation configurations could be inspected by an FS processor.

An industrial strength NLP architecture needs to identify separate components of its overall data model: in TALENT, annotations are complemented by e.g., a lexical cache, shared resources, ontological system of semantic categories, and so forth; see (Neff et al. Forthcoming). In order for the grammar writer to have uniform access to these, the notation supports the writing of grammars with reference to all of the underlying data model.

Trying to keep up with the breadth of data types comprising a complex data model makes for increasingly cumbersome notational conventions. Indeed, it is such complexities — aiming to allow for manipulating annotations and their properties — that can be observed at the root of the design decisions of systems discussed in Section 3. The challenge is to provide for all of that, without e.g., allowing for code fragments on the right-hand side of the rules (as GATE does), or appealing to 'back-door' library functions from an FST toolkit (as SPPC allows). Both problems, that of assuming that grammar writers would be familiar with the complete type system employed by all 'upstream' (and possibly third party) annotators, and that of exposing to them API's to an annotations store have already been discussed already.

Consequently, we make use of an abstraction layer between an annotation representation (as it is implemented, in the annotation repository) and a set of annotation property specifications which relate individual annotator capabilities to granularity of analysis. Matching against an annotation — within any family, and of any particular type — possibly further constrained by attributes specific to that type, becomes an atomic transition within a finite state device. We have developed a notation for FS operations, which appeals to the system-wide set of annotation families, with their property attributes. At any point in the annotations lattice, posted by annotators prior to the TFST plugin, the symbol for current match specifies the annotation type (with its full complement of attributes) to be picked from the lattice and considered by the match operator. Run-time behaviour of this operator is determined by a symbol compiler which uses the type system and the complete range of annotation iterators (as described above) to construct, dynamically (see below), the sequence of annotations defined by the current grammar as a particular traversal of the annotations lattice, and to apply to each annotation in that sequence the appropriate (also dynamically defined) set of tests for the specified configuration of annotation attributes.

Within the notation, it is also possible to express 'transduction' operations over annotations — such as create new ones, remove existing ones, modify and / or add properties, and so forth — as primitive operations. By defining such primitives, by conditioning them upon variable setting and testing, and by allowing annotations for successful matches to be bound to variables, arbitrary manipulation of features and values (including feature percolation and embedded references to annotation types as feature values) are made possible. (This, in its own right, completely removes the need for e.g., allowing code fragments on the RHS of grammar rules.) Full details concerning notation specifics can be found in (Boguraev & Neff 2003).

The uniform way of specifying annotation types on transitions of an FST graph hides from the grammar writer the system-wide design features separating the annotation repository from other components of the data model. For instance, access to conventional lexical resources, or to external repositories like lexical databases or gazetteers, appears to the grammar writer as querying an annotation with morpho-syntactic properties and attribute values, or looking for an annotation defined in terms of a semantic ontology. Similarly, a rule can update an external resource by using notational devices identical to those for posting annotations.

The freedom to define, and post, new annotation types 'on the fly' places certain requirements on the FST subsystem. In particular, it is necessary to infer how new annotations and their attributes fit into an already instantiated data model. The TFST annotator therefore incorporates logic which, during initialisation, scans an FST file (generated by an FST compiler typically running in the background), and determines — by deferring to the symbol compiler — what new annotation types and attribute features need to be dynamically configured and incrementally added to the model.

As discussed earlier, the implementation of a mechanism for picking a particular path through the annotations lattice over which any given rule should be applied — an essential component of an annotation-based regime of FS matching — is made possible through the system of iterators described above. Within such a framework, it is relatively straightforward to specify grammars, for instance, some of which would inspect raw tokens, others would abstract over vocabulary items (some of which would cover multiple tokens), yet others might traffic in constituent phrasal units (with an additional constrain over phrase type) or / and document structure elements (such as section titles, sentences, and so forth).

For grammars which examine uniform annotation types, it is possible to infer, and construct (for the run-time FS interpreter), an iterator over such a type (in this example, as is the default, sentences). It is also possible to further focus the matching operations so that a grammar only inspects inside of certain 'boundary' annotations. The formalism is thus capable of fine-grained specification of higher and / or lower context, in addition to left / right context — an essential component of lattice traversal. In general, expressive and powerful FS grammars may be written which inspect, at different — or even the same — point of the analysis annotations of different types. In this case it is essential that the appropriate iterators get constructed, and composed, so that a felicitous annotation stream gets submitted to the run-time for inspection; TALENT deploys a special dual-level iterator designed expressly for this purpose.

Additional features of the TFST subsystem allow for seamless integration of character-based regular expression matching (not limited to tokens, and uniformly targeting the 'covered string' under any annotation), morpho-syntactic abstraction from the underlying lexicon representation and part-of-speech tagset (allowing for transparent change in tagsets and tagger / models), and composition of complex attribute specification from simple feature tests (such as negation and conjunction). Overall, such features allow for the easy specification, via the grammar rules, of a variety of matching regimes which can transparently query the results of upstream annotators of which only the externally published capabilities are known.

## 5 Conclusion

The TALENT TFST system described in the previous section has been implemented in a framework with a fixed number of annotation families.

While simplifying the task of the symbol compiler, this has complicated the design of a notation where transduction rules need to specify just the right combination of annotation family, type, and properties.

In order to be truly compliant with notions like declarative representation of linguistic information, representational transparencies with respect to different components of a data model, and ability to support arbitrary levels of granularity of a set of analyses (which might have both potentially incompatible, and/or mutually dependent, attribute sets), our framework has recently adopted a system of typed feature structure-based (TFS) annotation types (Ferrucci & Lally Forthcoming). A redesign of the formalism, to support FS calculus over TFS's, brings us close to the definition of "annotation transducers", introduced by Wunsch (2003), where matching operations are defined over feature-based annotation descriptions and the match criterion is subsumption among feature structures. This is, is itself, derivative of the SProUT system (Droẑdẑyński et al. 2004). SProUT openly adopts full-blown TFS's to replace regular expressions' atomic symbols, with the matching operation itself defined as unifiability of TFS's. Feature structure expressions are also used to specify the shape and content of the result of a transduction operation which is creating, conceptually, a new annotation type, constructed by unification with respect to a type hierarchy.

The design of TFST has benefited greatly from the research described in the first two sections of this paper. Extending this, we bring together the advantages of a flexible, principled, rich and open-ended representation of annotations with novel mechanisms for traversing an annotations lattice and deriving an annotation stream to be submitted to an FS device. On the descriptive side, this makes it possible to develop grammars capable of examining and transforming any configuration of annotations in an annotations store created under real, possibly noisy circumstances, by potentially conflicting annotators. On the engineering side, the benefits of TFS-based representations underlying non-deterministic FS automata with unification — in particular, their compilability into super-efficient execution devices — have already been demonstrated: see, for instance, Brawer 1998, who reports matching speeds of up to 21 million tokens per second.

## REFERENCES

Abney, Steven. 1996. “Partial Parsing via Finite-State Cascades”. *Natural Language Engineering* 2:4.337-344.

Aït-Mokhtar, Salah & Jean-Pierre Chanod. 1997. “Incremental Finite-State Parsing”. *Proceedings of the Fifth Conference on Applied Natural Language Processing*, 72-79. Washington, D.C.

Ballim, Afzal & Vincenzo Pallotta, eds. 2002. *Robust Methods in Analysis of Natural Language Data* (= Special Issue of *Natural Language Engineering*, 8:2/3). Cambridge: Cambridge University Press.

Beesley, Kenneth & Lauri Karttunen. 2003. *Finite State Morphology*. Stanford, Calif.: CSLI Publications.

Bird, Steven & Mark Liberman. 2001. “A Formal Framework for Linguistic Annotation”. *Speech Communication* 33:1/2.23-60.

Bird, Steven, David Day, John Garofolo, John Henderson, Christophe Laprun & Mark Liberman. 2000. “ATLAS: A Flexible and Extensible Architecture for Linguistic Annotation”. *Proceedings of the Second International Conference on Language Resources and Evaluation* (*LREC-2000*), 1699-1706. Athens, Greece.

Boguraev, Branimir & Mary Neff. 2003. “The Talent 5.1 TFst System: User Documentation and Grammar Writing Manual”. Technical Report RC22976, IBM T.J. Watson Research Center, Yorktown Heights, New York, U.S.A.

Boguraev, Branimir. 2000. “Towards Finite-State Analysis of Lexical Cohesion”. *Proceedings of the 3rd International Conference on Finite-State Methods for NLP*, INTEX-3, Liege, Belgium.

Brawer, Sascha. 1998. *Patti: Compiling Unification-Based Finite-State Automata into Machine Instructions for a Superscalar Pipelined RISC Processor*. M.Sc. thesis, University of the Saarland, Saarbrücken, Germany.

Bunt, Harry & Laurent Romary. 2002. “Towards Multimodal Content Representation”. *Proceedings of the Workshop on International Standards for Terminology and Language Resource Management at the 3rd International Conference on Language Resources and Evaluation* (*LREC-2002*) ed. by K. Lee & K. Choi, 54-60, Las Palmas, Canary Islands, Spain.

Cowie, Jim & Douglas Appelt. 1998. “Pattern Specification Language”. TIPSTER Change Request–`http://www-nlpir.nist.gov/related_projects/tipster/rfcs/rfc10` [Source checked in May 2004]

Cunningham, Hamish & Donia Scott. 2004. *Software Architectures for Language Engineering* (= Special Issue of *Natural Language Engineering*, 10:4) Cambridge: Cambridge University Press.

Cunningham, Hamish. 2002. “GATE, a General Architecture for Language Engineering”. *Computers and the Humanities* 36:2.223-254.

Cunningham, Hamish, Diana Maynard & 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, U.K.

Cunningham, Hamish, Diana Maynard, Kalina Bontcheva & Valentin Tablan. 2002. “GATE: An Architecture for Development of Robust HLT Applications”. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (*ACL’02*), 168-175. Philadelphia, Pennsylvania.

Droẑdẑyński, Witold, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer & Feiyu Xu. 2004. “Shallow Processing with Unification and Typed Feature Structures — Foundations and Applications”. *Künstliche Intelligenz* 1:17-23.

Ferrucci, David & Adam Lally. Forthcoming. “Accelerating Corporate Research in the Development, Application and Deployment of Human Language Technologies.”. To appear in *Software Architectures for Language Engineering* (= Special Issue of *Natural Language Engineering*, 10:4).

Grefenstette, Gregory. 1999. “Light Parsing as Finite State Filtering”. *Extended Finite State Models of Language* ed. by András Kornai (= *Studies in Natural Language Processing*), 86-94. Cambridge, U.K.: Cambridge University Press.

Grishman, Ralph. 1996. “TIPSTER Architecture Design Document”. Technical report, DARPA, Version 2.2.

Grover, Claire, Colin Matheson, Andrei Mikheev & Marc Moens. 2000. “LT-TTT: A Flexible Tokenisation Tool”. *Proceedings of the Second International Conference on Language Resources and Evaluation* (*LREC-2000*), 1147-1154. Athens, Greece.

Hobbs, Jerry, Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel & Mabry Tyson. 1997. “FASTUS: A Cascaded Finite-State Transducer for Extracting Information from Natural-Language Text”. *Finite-State Language Processing* ed. by Emmanuel Roche & Yves Schabes (= *Language, Speech, and Communication*), 383-406. Cambridge, Mass.: MIT Press.

Ide, Nancy & Laurent Romary. Forthcoming. “International Standard for a Linguistic Annotation Framework”. To appear in *Software Architectures for Language Engineering* (= Special Issue of *Natural Language Engineering*, 10:4) Cambridge: Cambridge University Press.

Joshi, Aravind K. & Philip Hopely. 1999. “A Parser from Antiquity: An Early Application of Finite State Transducers to Natural Language Parsing”. *Extended Finite State Models of Language* ed. by András Kornai (= *Studies in Natural Language Processing*), 6-15. Cambridge, U.K.: Cambridge University Press.

Kaplan, Ronald M. & Martin Kay. 1994. “Regular Models of Phonological Rule Systems”. *Computational Linguistics* 20:3.331-378.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette & Anne Schiller. 1996. “Regular Expressions for Language Engineering”. *Natural Language Engineering* 4:1.305-328.

Kempe, André. 1997. “Finite State Transducers Approximating Hidden Markov Models”. *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and the 8th Conference of the European Chapter of the Association for Computational Linguistics* (*ACL-EACL'97*), 460-467. Madrid, Spain.

Kornai, Andras, ed. 1999. *Extended Finite State Models of Language*. Cambridge, U.K.: Cambridge University Press.

Mohri, Mehryar. 1997. “Finite-State Transducers in Language and Speech Processing”. *Computational Linguistics* 23:2.269-311.

Neff, Mary, Roy Byrd & Branimir Boguraev. Forthcoming. “The Talent System: TEXTRACT Architecture and Data Model”. To appear in *Software Architectures for Language Engineering* (= Special Issue of *Natural Language Engineering*, 10:4). Cambridge: Cambridge University Press.

Neumann, Günter & Jakub Piskorski. 2000. “An Intelligent Text Extraction and Navigation System”. *Proceedings of Content-Based Multimedia Information Access* (*Recherche d'Informations Assistee par Ordinateur RIAO-2000*), 239-246. Paris, France.

Neumann, Günter & Jakub Piskorski. 2002. “A Shallow Text Processing Core Engine”. *Journal of Computational Intelligence* 18:3.451-476.

Patrick, Jon & Hamish Cunningham, eds. 2003. *Workshop on The Software Engineering and Architecture of Language Technology Systems at HLT-NAACL 2003*. Edmonton, Alberta, Canada.

Pereira, Fernando & Rebecca Wright. 1997. “Finite-State Approximation Of Phrase Structure Grammars”. *Finite-State Language Processing* ed. by Emmanuel Roche & Yves Schabes (= *Language, Speech, and Communication*), 149-174. Cambridge, Mass.: MIT Press.

Piskorski, Jakub. 2002. “The DFKI Finite-State Toolkit”. Technical Report RR-02-04, German Research Center for Artificial Intelligence (DFKI), Saarbruecken, Saarland, Germany.

Roche, Emmanuel & Yves Schabes. 1995. “Deterministic Part-of-Speech Tagging with Finite-State Transducers”. *Computational Linguistics* 21:2.227-253.

Seitsonen, Lauri. 2001. “BRIEFS Information Extraction — Phase 2”. Technical report, TAI Research Center, Helsinki University of Technology, Helsinki, Finland.

Silberztein, Max. 2000. “INTEX: An Integrated FST Development Environment”. *Theoretical Computer Science* 231:1.33-46.

Simov, Kiril, Milen Kouylekov & Alexander Simov. 2002. “Cascaded Regular Grammars over XML Documents”. *Proceedings of the Second International*

*Workshop on NLP and XML (NLPXML-2002)*, Taipei, Tawian. — `http://www.bultree bank.org/papers/XCRG-NLPXML-2002.pdf` [Source checked in May'04]

Srihari, R. K., Wei Li, Cheng Niu & Thomas Cornell. 2003. "InfoXtract: A Customizable Intermediate Level Information Extraction Engine". *Proceedings of the Workshop on Software Engineering and Architectures of Language Technology Systems at HLT-NAACL*, 52-59. Edmonton, Alberta, Canada.

van Noord, Gertjan & Dale Gerdemann. 2001. "An Extendible Regular Expression Compiler for Finite-State Approaches in Natural Language Processing". *Automata Implementation. 4th International Workshop on Implementing Automata, WIA '99, Potsdam Germany, July 1999, Revised Papers* ed. by O. Boldt & H. Juergensen (= *Lecture Notes in Computer Science*, 2214), 122-141. Berlin: Springer.

Wunsch, Holger. 2003. *Annotation Grammars and Their Compilation into Annotation Transducers*. M.Sc. thesis, Univ. of Tübingen, Tübingen, Germany.

## Index of Subjects and Terms