SSF: A Common Representation Scheme for Language Analysis for Language Technology Infrastructure Development

Akshar Bharati
Akshar Bharati Group
Hyderabad
sangal@iiit.ac.in

Rajeev Sangal
IIT (BHU), Varanasi
sangal@iiit.ac.in

Dipti Sharma
IIIT, Hyderabad
dipti@iiit.ac.in

Anil Kumar Singh
IIT (BHU), Varanasi
nlprnd@gmail.com

Abstract

We describe a representation scheme and an analysis engine using that scheme, both of which have been used to develop infrastructure for HLT. The Shakti Standard Format is a readable and robust representation scheme for analysis frameworks and other purposes. The representation is highly extensible. This representation scheme, based on the blackboard architectural model, allows a very wide variety of linguistic and non-linguistic information to be stored in one place and operated upon by any number of processing modules. We show how it has been successfully used for building machine translation systems for several language pairs using the same architecture. It has also been used for creation of language resources such as treebanks and for different kinds of annotation interfaces. There is even a query language designed for this representation. Easily wrappable into XML, it can be used equally well for distributed computing.

1 Introduction

Building infrastructures for human language technology is a non-trivial task. There can be numerous issues that have to be addressed, whether linguistic or non-linguistic. Unless carefully managed, the overall complexity can easily get out of control and seriously threaten the sustainability of the system. This may apply to all large software systems, but the complexities associated with humans languages (both within and across languages) only add to the problem. To make it possible to build various components of an infrastructure that scales within and across languages for a wide variety of purposes, and to be able to do it by re-using the representation(s) and the code, deserves to be considered an achievement.

GATE[1] (Cunningham et al., 2011; Li et al., 2009), UIMA[2] (Ferrucci and Lally, 2004; Bari et al., 2013; Noh and Padó, 2013) and NLTK[3] (Bird, 2002) are well known achievements of this kind. This paper is about one other such effort that has proved to be successful over the last decade or more.

2 Related Work

GATE is designed to be an architecture, a framework and a development environment, quite like UIMA, although the two differ in their realization of this goal. It enables users to develop and deploy robust language engineering components and resources. It also comes bundled with several commonly used baseline Natural Language Processing (NLP) applications. It makes strict distinction between data, algorithms, and ways of visualising them, such that algorithms + data + GUI = applications. Consequently, it has three types of components: language resources, processing resources and visual resources.GATE uses an annotation format with stand-off markup.

UIMA is a middleware architecture for processing unstructured information (UIM) (Ferrucci and Lally, 2004), with special focus on NLP. Its development originated in the realization that the ability to quickly discover each other's results and rapidly combine different technologies and approaches accelerates scientific advance. It has powerful search capabilities and a data-driven framework for the

This work is licenced under a Creative Commons Attribution 4.0 International License. Page numbers and proceedings footer are added by the organizers. License details: http://creativecommons.org/licenses/by/4.0/

[1] http://gate.ac.uk/
[2] https://uima.apache.org/
[3] http://www.nltk.org/

Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT, pages 66–76,
Dublin, Ireland, August 23rd 2014.

development, composition and distributed deployment of analysis engines. More than the development of independent UIM applications, UIMA aims to enable accelerated development of integrated and robust applications, combining independent applications in diverse sub-areas of NLP, so as to accelerate the research cycle as well as the production time. In UIMA, The original document and its analysis are represented in a structure called the Common Analysis Structure, or CAS. Annotations in the CAS are maintained separately from the document itself to allow greater flexibility than inline markup. There is an XML specification for the CAS and it is possible to develop analysis engines that operate on and output data in this XML format, which also (like GATE and NLTK) uses stand-off markup.

The Natural Language Toolkit (NLTK) is a suite of modules, data sets and tutorials (Bird, 2002). It supports many NLP data types and can process many NLP tasks. It has a rich collection of educational material (such as animated algorithms) for those who are learning NLP. It can also be used as a platform for prototyping of research systems.

SSF and the Shakti Analyzer are similar to the above three but have a major difference when compared with them. SSF is a "powerful" notation for representing the NLP analysis, at all stages, whether morphological, part-of-speech level, chunk level, or sentence level parse. The notation is so designed that it is flexible, as well as readable. The notation can be read by human beings and can also be loaded in memory, so that it can be used efficiently. It also allows the architecture to consist of modules which can be configured easily under different settings. The power of the notation and the flexibility of the resulting architecture gives enormous power to the system framework.

The readability of the format allows it to be used directly with any plain text editors, without requiring the use of any special tools or editors. Many users prefer the data in plain text format as it allows them to use the editors they are familiar with. Such readability and simplicity has turned out, in our experience, to be an advantage even for experts like software developers and (computer savvy) linguists.

It would be an interesting exercise to marry SSF notation and the Shakti way of doing things with the GATE and UIMA architecture. Our own feeling is that the resulting system/framework with a powerful notation like SSF and the comprehensive framework like UIMA/GATE would lead to a new even more powerful framework with a principled notation.

3 Shakti Standard Format

Shakti Standard Format (SSF) is a representation scheme (along with a corresponding format) that can be used for most kinds of linguistically analyzed data. It allows information in a sentence to be represented in the form of one or more *trees* together with a set of *attribute-value* pairs with nodes of the trees. The attribute-value pairs allow features or properties to be specified with every node. *Relations* of different types across nodes can also be specified using an attribute-value like representation. The representation is specially designed to allow different levels and kinds of linguistic analyses to be stored. The developers use APIs to store or access information regarding structure of trees and attribute-value pairs.

If a module is successful in its task, it adds a new analysis using trees and attribute values to the representation. Thus, even though the format is fixed, it is extensible in terms of attributes or analyses. This approach allows ready-made packages (such as, POS tagger, chunker, and parser) to be incorporated easily using a wrapper (or a pair of converters). In order to interface such pre-existing packages to the system, all that is required is to convert from (input) SSF to the input format required by that package and, the output of the package to SSF. The rest of the modules of the system continue to operate seamlessly.

The format allows both in-memory representation as well as stream (or text) representation. They are inter-convertible using a *reader* (stream to memory) and *printer* (memory to stream). The in-memory representation is good in speed of processing, while the stream is good for portability, heterogenous machines, and flexibility, in general.

SSF promotes the dictum: "Simplify globally, and if unavoidable, complicate only locally." Even if the number of modules is large and each module does a small job, the local complexity (of individual modules) remains under tight control for most of the modules. At worst, complexity is introduced only locally, without affecting the global simplicity.

3.1 Text Level SSF

In SSF, a text or a document has a sequence of sentences with some structure such as paragraphs and headings. It also includes meta information related to title, author, publisher, year and other information related to the origin of the text or the document. Usually, there is also the information related to encoding, and version number of the tagging scheme, etc. The text level SSF has two parts, header and body:

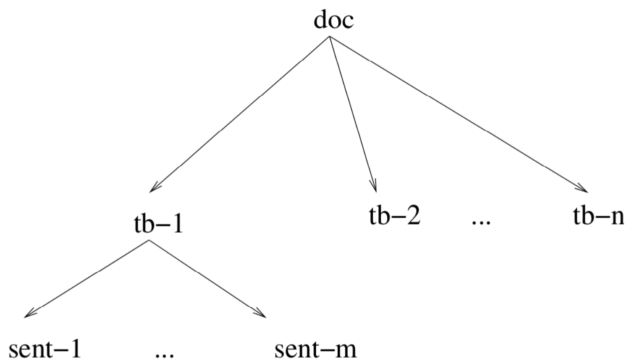

Figure 1: Document Structure in SSF

```
<document docid="..." docnumber="...">
<header>
...
</header>
<body>
...
</body>
```

The header contains meta information about the title, author, publisher, etc. as contained in the CML (Corpus Markup Language) input[4]. The body contains sentences, each in SSF. The body of a text in SSF contains text blocks given by the tag *tb*.

```
<body encode= ... >
<tb>
...
</tb>
...
</body>
```

A text block (tb) contains a sequence of sentences. Each sentence can be marked as a *segment* (to indicate a heading, a partial sentence, etc.) or not a segment (to indicate a normal sentence).

3.2 Sentence Level SSF

Several formalisms have been developed for such descriptions, but the two main ones in the field of NLP are Phrase Structure Grammar (PSG) (Chomsky, 1957) and Dependency Grammar (DG) (Tesniere, 1959). In PSG, a set of phrase structure rules are given for the grammar of a language. It is constituency based and order of elements are a part of the grammar, and the resulting tree. DG, on the other hand, is relational and shows relations between words or elements of a sentence. It, usually, tries to capture the syntactico-semantic relations of the elements in a sentence. The resulting dependency tree is a tree with nodes and edges being labelled.

The difference in the two approaches are shown below with the help of the following English example:

```
Ram ate the banana.
```

The phrase structure tree is drawn in Fig. 2 using a set of phrase structure rules. Fig. 3 shows the dependency tree representation for this sentence. SSF can represent both these formats.

[4]Thus SSF becomes a part of CML.

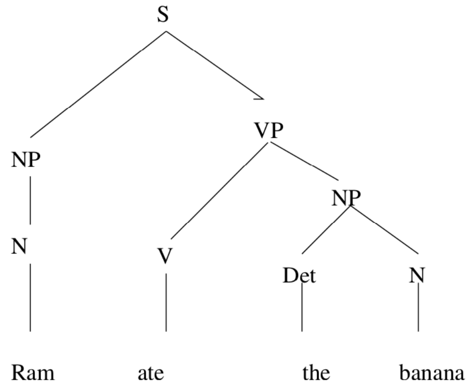

Figure 2: Phrase structure tree

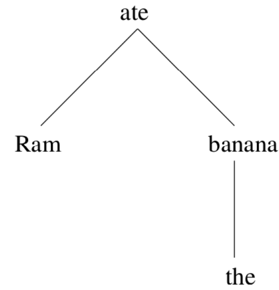

Figure 3: Dependency tree

Sentence level SSF is used to store the analysis of a sentence. It occurs as part of text level SSF. The analysis of a sentence may mark any or all of the following kinds of information as appropriate: part of speech of the words in the sentence; morphological analysis of the words including properties such as root, gender, number, person, tense, aspect, modality; phrase-structure or dependency structure of the sentence; and properties of units such as chunks, phrases, local word groups, bags, etc. Note that SSF is theory neutral and allows both phrase structure as well as dependency structure to be coded, and even mixed in well defined ways.

Though the format in SSF is fixed, it is extensible to handle new features. It also has a text representation, which makes it easy to read the output. The following example illustrates the SSF. For example, the following English sentence,

```
Children are watching some programmes on television in the house. -- (1)
```

The representation for the above sentence is shown in SSF in Fig. 4. As shown in this figure, each line represents a word/token or a group (except for lines with '))' which only indicate the end of a group). For each group, the symbol used is '(('. Each word or group has 3 parts. The first part stores the tree address of each word or group, and is for human readability only. The word or group is in the second part, with part of speech tag or group/phrase category in the third part.

```
Address    Token          Category       Attribute-value pairs
  -----------------------------------------------
  1        ((               NP
  1.1      children         NNS      <fs af=child,n,m,p,3,0,,>
           ))
  2        ((               VG
  2.1      are              VBP      <fs af=be,v,m,p,3,0,,>
  2.2      watching         VBG      <fs af='watch,v,m,s,3,0,,' aspect=PROG>
           ))
  3        ((               NP
  3.1      some             DT       <fs af=some,det,m,s,3,0,,>
  3.2      programmes       NNS      <fs af=programme,n,m,p,3,0,,>
           ))
  4        ((               PP
  4.1      on               IN       <fs af=on,p,m,s,3,0,,>
  4.1.1    ((               NP
  4.1.2    television       NN       <fs af=television,n,m,s,3,0,,>
           ))
           ))
  5        ((               PP
  5.1      in               IN       <fs af=in,p,m,s,3,0,,>
  5.2      ((               NP
  5.2.1    the              DT       <fs af=the,det,m,s,3,0,,>
  5.2.2    house            NN       <fs af=house,n,m,s,3,0,,>
           ))
           ))
  -----------------------------------------------
```

Figure 4: Shakti Standard Format

The example below shows the SSF for the first noun phrase where feature information is also shown, as the fourth part on each line. Some frequently occurring attributes (such as root, cat, gend, etc.) may be abbreviated using a special attribute called ‘af’ or abbreviated attributes, as follows:

```
1       ((              NP
1.1     children        NNS     <fs af='child,n,m,p,3,0,,' >
                                         |    | | | | |
                                         |    | | | |  \
                                        root  | | |pers |
                                              | | |    case
                                       category | number
                                                |
                                              gender
```

The field for each attribute is at a fixed position, and a comma is used as a separater. Thus, in case no value is given for a particular attribute, the field is left blank, e.g. last two fields in the above example.

Corresponding to the above SSF text stream, an in-memory data structure may be created using the APIs. (However, note that value of the property *Address* is not stored in the in-memory data structure explicitly. It is for human reference and readability only, and is computed when needed. A unique name, however can be assigned to a node and saved in the memory, as mentioned later.)

There are two types of attributes: user defined or system defined. The convention that is used is that a user defined attribute should not have an underscore at the end. System attribute may have a single underscore at its end.

Values are of two types: simple and structured. Simple values are represented by alphanumeric strings, with a possible underscore. Structured values have progressively more refined values separated by double underscores. For example, if a value is:

```
vmod__varg__k1
```

it shows the value as ‘vmod’ (modifier of a verb), which is further refined as ‘varg’ (argument of the verb) of type ‘k1’ (karta karaka).

3.3 Interlinking of Nodes

Nodes might be interlinked with each other through directed edges. Usually, these edges have nothing to do with phrase structure tree, and are concerned with dependency structure, thematic structure, etc. These are specified using the attribute value syntax, however, they do not specify a property for a node, rather a relation between two nodes.

For example, if a node is karta karaka of another node named ‘play1’ in the dependency structure (in other words, if there is a directed edge from the latter to the former) it can be represented as follows:

1	children	NN	$< fs\, drel =' k1 : play1' >$
2	played	VB	$< fs\, name = play1 >$

The above says that there is an edge labelled with ‘k1’ from ‘played’ to ‘children’ in the ‘drel’ tree (dependency relation tree). The node with token ‘played’ is named as ‘play1’ using a special attribute called ‘name’.

So the syntax is as follows: if you associate an arc with a node C as follows:

```
<treename>=<edgelabel>:<nodename>
```

it means that there is an edge from $< nodename >$ to C, and the edge is labelled with $< edgelabel >$. Name of a node may be declared with the attribute ‘name’:

```
name=<nodename>
```

3.4 Cross Linking across Sentences

There is a need to relate elements across sentences. A common case is that of co-reference of pronouns. For example, in the following sentences:

```
Sita saw Ram in the house. He had come all by himself.  -- (2)
```

the pronoun ‘he’ in the second sentence refers to the same person as referred to by ‘Ram’. Similarly ‘himself’ refers to same person as ‘he’ refers to. This is show by means of a co-reference link from ‘he’ to ‘Ram’, and from ‘himself’ to ’he’. SSF allows such cross-links to be marked.

The above text of two sentences is shown in SSF below.

```
<document docid="gandhi-324" docnumber="2">
<header> ...  </header>
<body>
<tb>
  <sentence num=1>
  ...
  2 Ram             <fs name=R>
  ...
  </sentence>
  <sentence num=2>
  1 He               <fs coref="..%R"  name=he>
  ...
  6 himself          <fs coref=he>
  7 .
  </sentence>
</tb>
```

Note that ‘himself’ in sentence 2 co-refers to ‘he’ in the same sentence. This is shown using attribute ‘coref’ and value ‘he’. To show co-reference across sentences, a notation is used with ‘%’. It is explained next.

Name labels are defined at the level of a sentence: Scope of any name label is a sentence. It should be unique within a sentence, and can be referred to within the sentence by using it directly.

To refer to a name label in another sentence in the same text block (paragraph), path has to be specified:

```
..%R
```

To refer to a name label R in a sentence in another text block numbered 3, refer to it as:

```
..%..%3%1%R
```

4 Shakti Natural Language Analyzer

Shakti Analyzer has been designed for analyzing natural languages. Originally, it was available for analyzing English as part of the Shakti[5] English-Hindi machine translation system. It has now been extended for analyzing a number of Indian languages as mentioned later (Section-6.1).

The Shakti Analyzer can incorporate new modules as black boxes or as open-source software. The simplicity of the overall architecture makes it easy to do so. Different available English parsers have been extensively adapted, and the version used by Shakti system runs using Collins parser.

Shakti analyzer combines rule-based approach with statistical approach. The SSF representation is designed to keep both kinds of information. The rules are mostly linguistic in nature, and the statistical approach tries to infer or use linguistic information. For example, statistical POS tagger tries to infer linguistic (part-of-speech) tags, whereas WSD module uses grammatical relations together with statistics to disambiguate the word sense.

The system has a number of innovative design principles which are described below.

4.1 System Organization Principles

A number of system organization principles have been used which have led to the rapid development of the system. While the principles by themselves might not appear to be new, their application is perhaps new.

4.1.1 Modularity

The system consists of a large number of modules, each one of which typically performs a small logical task. This allows the overall machine translation task to be broken up into a large number of small subtasks, each of which can be accomplished separately. Currently the system (as used in the Shakti system)

[5]http:/shakti.iiit.ac.in

has 69 different modules. About 9 modules are used for analyzing the source language (English), 24 modules are used for performing bilingual tasks such as substituting target language roots and reordering etc., and the remaining modules are used for generating target language.

4.1.2 Simplicity of Organization

The overall system architecture is kept extremely simple. All modules operate on data in SSF . They communicate with each other via SSF.

The attribute value pairs allow features or properties to be specified with every node. Relations of different types across nodes can also be specified using an attribute-value like representation. The representation is specially designed to allow different levels and kinds of linguistic analyses to be stored. The developer uses APIs to store or access information regarding structure of trees and attribute value pairs.

4.1.3 Designed to Deal with Failure

NLP analysis modules are known to have limited coverage. They are not always able to produce an output. They fail to produce output either because of limits of the best known algorithms or incompleteness of data or rules. For example, a sentential parser might fail to parse either because it does not know how to deal with a construction or because a dictionary entry is missing. Similarly, a chunker or part of speech tagger might fail, at times, to produce an analysis. The system is designed to deal with failure at every step in the pipeline. This is facilitated by a common representation for the outputs of the POS tagger, chunker and parser (all in SSF). The downstream modules continue to operate on the data stream, albeit less effectively, when a more detailed analysis is not available. (If all modules were to fail, a default rule of no-reordering and dictionary lookup would still be applied.)

As another example, if the word sense disambiguation (WSD) module fails to identify the sense of a word in the input sentence, it does not put in the sense feature for the word. This only means that the module which substitutes the target language root from the available equivalents from dictionary, will use a default rule for selecting the sense because the detailed WSD was not successful (say, due to lack of training data).

The SSF is designed to represent partial information, routinely. Appropriate modules know what to do when their desired information is available and use defaults when it is not available. In fact, for many modules, there are not just two but several levels at which they operate, depending on availability of information corresponding to that level. Each level represents a graceful degradation of output quality.

The above flexibility is achieved by using two kinds of representation: constituent level representation and feature-structure level representation. The former is used to store phrase level analysis (and partial parse etc.) and the latter for outputs of many kinds of other tasks such as WSD, TAM computation, case computation, dependency relations, etc.

4.1.4 Transparency for Developers

An extremely important characteristic for the successful development of complex software such as a machine translation system is to expose the input and output produced by every module. This transparency becomes even more important in a research environment where new ideas are constantly being tried with a high turnover of student developers.

In the Shakti system, unprecedented transparency is achieved by using a highly readable textual notation for the SSF, and requiring every module to produce output in this format. In fact, the textual SSF output of a module is not only for the human consumption, but is used by the subsequent module in the data stream as its input. This ensures that no part of the resulting analysis is left hidden in some global variables; all analysis is represented in readable SSF (otherwise it is not processed at all by the subsequent modules).

Experience has shown that this methodology has made debugging as well as the development of the system convenient for programmers and linguists alike. In case an output is not as expected, one can quickly find out which module went wrong (that is, which module did not function as expected). In fact, linguists are using the system quite effectively to debug their linguistic data with ease.

5 Implementations

A considerable repository of implementations (in code) has evolved around SSF and the analyzer. In this section we consider two of the kinds of implementations that have accumulated so far.

5.1 SSF API

Application Programming Interfaces (APIs) have been implemented in multiple programming languages to allow programmers to transparently operate on any data stored in SSF. Of these, the better designed APIs, such as those in Perl and Java, allow all kinds of operations to be performed on the SSF data. These operation include basic operations such as reading, writing and modifying the data, as well as for advanced operations such as search and bulk transformation of the data. The Java API is a part of Sanchay [6], which is a collection of tools and APIs for language processing, specially tailored for the needs of Indian languages which were not (till very recently) well supported on computers and operating systems.

The availability of decently designed APIs for SSF allow programmers to use SSF for arbitrary purposes. And they have used it successfully to build natural language systems and tools as described below.

5.2 Sanchay Corpus Query Language

Trees have a quite constrained structure, whereas graphs have somewhat anarchic structure. Threaded trees (Ait-Mokhtar et al., 2002; Larchevelque, 2002) provided a middle ground between the two. They start with trees as the core structure, but they allow constrained links between the nodes of a tree that a pure tree would not allow. This overlaying of constrained links over the core trees allows multiple layers and/or types of annotation to be stored in the same structure. With a little more improvisation, we can even have links across sentences, i.e., at the discourse level (see section-3.3). It is possible, for example, to have a phrase structure tree (the core tree) overlaid with a dependency tree (via constrained links or 'threads'), just as it is possible to have POS tagged and chunked data to be overlaid with named entities and discourse relations.

The Sanchay Corpus Query Language (SCQL) (Singh, 2012) is a query language designed for threaded trees. It so turns out that SSF is also a representation that can be viewed as threaded trees. Thus, the SCQL can work over data in SSF. This language has a simple, intuitive and concise syntax and high expressive power. It allows not only to search for complicated patterns with short queries but also allows data manipulation and specification of arbitrary return values. Many of the commonly used tasks that otherwise require writing programs, can be performed with one or more queries.

6 Applications

6.1 Sampark Machine Translation Architecture

Overcoming the language barrier in the Indian sub-continent is a very challenging task[7]. Sampark[8] is an effort in this direction. Sampark has been developed as part of the consortium project called Indian Language to India Language Machine translation (ILMT) funded by TDIL program of Department of Information Technology, Government of India. Work on this project is contributed to by 11 major research centres across India working on Natural Language Processing.

Sampark, or the ILMT project, has developed language technology for 9 Indian languages resulting in MT for 18 language pairs. These are: 14 bi-directional systems between Hindi and Urdu / Punjabi / Telugu / Bengali / Tamil / Marathi / Kannada and 4 bi-directional systems between Tamil and Malayalam / Telugu. Out of these, 8 pairs have been exposed via a web interface. A REST API is also available to acess the machine translation system over the Internet.

[6] http://sanchay.co.in

[7] There are 22 constitutionally recognized languages in India, and many more which are not recognized. Hindi, Bengali, Telugu, Marathi, Tamil and Urdu are among the major languages of the world in terms of number of speakers, summing up to a total of 850 million.

[8] http://sampark.org.in

The Sampark system uses Computational Paninian Grammar (CPG) (Bharati et al., 1995), in combination with machine learning. Thus, it is a hybrid system using both rule-based and statistical approaches. There are 13 major modules that together form a hybrid system. The machine translation system is based on the analyze-transfer-generate paradigm. It starts with an analysis of the source language sentence. Then a transfer of structure and vocabulary to target language is carried out. Finally the target language is generated. One of the benefits of this approach is that the language analyzer for a particular language can be developed once and then be combined with generators for other languages, making it easier to build a machine translation system for new pairs of languages.

Indian languages have a lot of similarities in grammatical structures, so only shallow parsing was found to be adequate for the purposes of building a machine translation system. Transfer grammar component has also been kept simple. Domain dictionaries are used to cover domain specific aspects.

At the core of the Sampark architecture is an enhanced version of the Shakti Natural Language Analyzer. The individual modules may, of course, be different for different language pairs, but the pipelined architecture bears close resemblance to the Shakti machine translation system. And it uses the Shakti Standard Format as the blackboard (Erman et al., 1980) on which the different modules (POS taggers, chunkers, named entity recognzier, transfer grammar module etc.) operate, that is, read from and write to. SSF thus becomes the glue that ties together all the modules in all the MT systems for the various language pairs. The modules are not only written in different programming languages, some of them are rule-based, whereas others are statistical.

The use of SSF as the underlying default representation helps to control the complexity of the overall system. It also helps to achieve unprecedented transparency for input and output for every module. Readability of SSF helps in development and debugging because the input and output of any module can be easily seen and read by humans, whether linguists or programmers. Even if a module fails, SSF helps to run the modules without any effect on normal operation of system. In such a case, the output SSF would have unfilled value of an attribute and downstream modules continue to operate on the data stream.

6.2 Annotation Interfaces and Other Tools

Sanchay, mentioned above, has a syntactic annotation interface that has been used for development of treebanks for Indian languages (Begum et al., 2008). These treebanks have been one of the primary sources of information for the development the Sampark machine translation systems, among other things. This syntactic annotation interface provides facilities for everything that is required to be done to transform the selected data in the raw text format to the final annotated treebank. The usual stages of annotation include POS tagging, morphological annotation, chunking and dependency annotation. This interface has evolved over a period of several years based on the feedback received from the annotators and other users. There are plans to use the interface for similar annotation for even more languages.

The underlying default format used in the above interface is SSF. The advantages of using SSF for this purpose are similar to those mentioned earlier for purposes such as building machine translation systems. The complete process of annotation required to build a full-fledged treebank is complicated and there are numerous issues that have to be taken care of. The blackboard-like nature of SSF allows for a smooth shifts between different stages of annotation, even going back to an earlier stage, if necessary, to correct mistakes. It allows all the annotation information to be situated in one contiguous place.

The interface uses the Java API for SSF, which is perhaps the most developed among the different APIs for SSF. The API (a part of Sanchay) again allows transparency for the programmer as far as manipulating the data is concerned. It also ensures that there are fewer bugs when new programmers work on any part of the system where SSF data is being used. One recent addition to the interface was a GUI to correct mistakes in treebanks (Agarwal et al., 2012).

The syntactic annotation interface is not the only interface in Sanchay that uses SSF. Some other interfaces do that too. For example, there are sentence alignment and word alignment interfaces, which also use the same format for similar reasons. Thus, it is even possible to build parallel treebanks in SSF using the Sanchay interfaces.

Then there are other tools in Sanchay such as the integrated tool for accessing language resources (Singh and Ambati, 2010). This tool allows various kinds of language resources, including those in SSF, to be accessed, searched and manipulated through the inter-connected annotation interfaces and the SSF API. There is also a text editor in Sanchay that is specially tailored for Indian languages and it can validate SSF (Singh, 2008).

The availability of a corpus query language (section-5.2) that is implemented in Sanchay and that can be used for data in SSF is another big facilitator for anyone who wants to build new tools for language processing and wants to operate on linguistic data.

Apart from these, a number of research projects have used SSF (the representation or the analyzer) directly or indirectly, that is, either for theoretical frameworks or as part of the implementation (Bharati et al., 2009; Gadde et al., 2010; Husain et al., 2011).

7 Conclusion

We described a readable representation scheme called Shakti Standard Format (SSF). We showed how this scheme (an instance of the blackboard architectural model), which is based on certain organizational principles such as modularity, simplicity, robustness and transparency, can be used to create not only a linguistic analysis engine (Shakti Natural Language Analyzer), but can be used for arbitrary other purposes wherever linguistic analysis is one of the tasks. We briefly described the machine translation systems (Shakti and Sampark) which use this scheme at their core level. Similarly, we described how it can be used for creation of language resources (such as treebanks) and the annotation interfaces used to create these resources. It has also figured in several research projects so far. We mentioned one query language (Sanchay Corpus Query Language) that operates on this representation scheme and has been integrated with the annotation interfaces. Overall, the representation scheme has been successful at building infrastructure for language technology over the last more than a decade. The scheme is theory neutral and can be used for both phrase structure grammar and for dependency grammar.

References

Rahul Agarwal, Bharat Ram Ambati, and Anil Kumar Singh. 2012. A GUI to Detect and Correct Errors in Hindi Dependency Treebank. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC)*, Instanbul, Turkey. ELRA.

S. Ait-Mokhtar, J.P. Chanod, and C. Roux. 2002. Robustness beyond shallowness: incremental deep parsing. *Natural Language Engineering*, 8(2-3):121144, January.

Alessandro Di Bari, Alessandro Faraotti, Carmela Gambardella, and Guido Vetere. 2013. A Model-driven approach to NLP programming with UIMA. In *UIMA@GSCL*, pages 2–9.

Rafiya Begum, Samar Husain, Arun Dhwaj, Dipti Misra Sharma, Lakshmi Bai, and Rajeev Sangal. 2008. Dependency Annotation Scheme for Indian Languages. In *Proceedings of The Third International Joint Conference on Natural Language Processing (IJCNLP)*, Hyderabad, India.

Ashkar Bharati, Vineet Chaitanya, and Rajeev Sangal. 1995. *Natural Language Processing: A Paninian Perspective*. Prentice-Hall of India Pvt. Ltd.

Akshar Bharati, Samar Husain, Phani Gadde, Bharat Ambati, Dipti M Sharma, and Rajeev Sangal. 2009. A Modular Cascaded Approach to Complete Parsing. In *Proceedings of the COLIPS International Conference on Asian Language Processing 2009 (IALP)*, Singapore.

Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Philadelphia: Association for Computational Linguistics*.

Noam Chomsky. 1957. *Syntactic Structures*. The Hague/Paris: Mouton.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. 2011. *Text Processing with GATE (Version 6)*.

Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. 1980. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Comput. Surv.*, 12(2):213–253, June.

D. Ferrucci and A. Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348.

Phani Gadde, Karan Jindal, Samar Husain, Dipti Misra Sharma, and Rajeev Sangal. 2010. Improving Data Driven Dependency Parsing using Clausal Information. In *Proceedings of 11th Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, Los Angeles.

Samar Husain, Phani Gadde, Joakim Nivre, and Rajeev Sangal. 2011. Clausal Parsing Helps Data-driven Dependency Parsing: Experiments with Hindi. In *Proceedings of Fifth International Joint Conference on Natural Language Processing (IJCNLP)*, Thailand.

J.M. Larchevelque. 2002. Optimal Incremental Parsing. *ACM Transactions on Programing Languages and Systems*, 17(1):115, January.

Yaoyong Li, Kalina Bontcheva, and Hamish Cunningham. 2009. Adapting SVM for Data Sparseness and Imbalance: A Case Study on Information Extraction. *Natural Language Engineering*, 15(2):241–271.

Tae-Gil Noh and Sebastian Padó. 2013. Using UIMA to Structure An Open Platform for Textual Entailment. In *UIMA@GSCL*, pages 26–33.

Anil Kumar Singh and Bharat Ambati. 2010. An Integrated Digital Tool for Accessing Language Resources. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC)*, Malta. ELRA.

Anil Kumar Singh. 2008. A Mechanism to Provide Language-Encoding Support and an NLP Friendly Editor. In *Proceedings of the Third International Joint Conference on Natural Language Processing (IJCNLP)*, Hyderabad, India. AFNLP.

Anil Kumar Singh. 2012. A Concise Query Language with Search and Transform Operations for Corpora with Multiple Levels of Annotation. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC)*, Instanbul, Turkey. ELRA.

L. Tesniere. 1959. *Elements de syntaxe structurale*. Paris: Klincksieck.