10 An XML-Schema enabled advanced template model for structured documents

10.1 Introduction

The spatial model based template model is designed mainly for semi-structured documents, in order to solve conflicts carried by NE-Agent. For structured documents, the spatial relationship is strictly defined. That is, the probabilities of spatial relations are always 100%. Hence, the spatial model does not play important role here. On the other hand, for some structured documents, the template model is more complicated than simple flat tree. Model elements or part model can have complex cardinality property (see also case study 1, Section 4.1). For such complicated situation, spatial model is not able to be used to perform template filling.

Furthermore, some structured documents are described not informally, but in formal ways. This type of documents is widely used in the industry area such as log files, configuration files, message exchange protocols etc. Typical structured documents are written with some self-defined grammars and stored with ASCII format. The structures in documents are represented normally with predefined tokens and data fields. To read and understand such documents, parsers are developed. Because of the individuality of the definitions of data structures, parsers are implemented individually. One parser is only for one defined data structure. The individual implementing of parsers is expensive. A idea solution to structured documents is a generic parser, which can be configured to parse (extract) different types of documents.

Also, with the growing of Internet/Intranet, the challenge of data integration becomes emergent. That is, data must be represented in standard format and can be exchange with each other. Moreover, the requirement to extracting performance is very high, namely 100% precision and recall, because the data is read by machine automatically. To extract such data and export them with a standard format (e.g. XML), a new type of template model has to be developed.

XML is becoming a standard description language for structured information. Various structured data described by XML can be parsed, validated, processed with standard techniques. However, XML is relative new technique. That is, documents created in the early times are not XML-based. Many documents created today also are not XML complaint. Currently, one parser is only for one defined data structure.

One question is, could is it possible, to develop a generic parser for all different structured documents, or to convert the old style structured
documents to well-formed XML format? To go to the deep analyse, following two points are to take into account:

- Each structured document, whatever in ASCII or XML style, has its own structure definition. When such structure is shown in a tree style model, the contribution is shown as the same. Hence, an abstract document structure definition can be used for different structured documents regardless of their representation forms. In XML, such definition is named DTD (Document Type Definition) or XML-Schema. If we consider XML-Schema only as a definition for structured data, it can then be used for any structured documents, not only those that are written with XML-grammars. That is, we can use XML-Schema to define the structure model of all existing structured document. In Information Extraction, the XML-Schema becomes language of template definition.

- Once we have a document type definition, the next question is how to parse the document based on the definition. Normally, a stand-alone parser has hard-coded rules to recognise tokens fields and to extract underlying data. Because structured documents are designed for machine understanding and automatic parsing, underlying data is normally easily extracted with help of significant context strings (such as field name etc.) If recognition and extracting rules are not hard-coded but configurable, advanced Information Extraction techniques could extract almost all useful information using corresponded rules. Hence, a new approach is, that we can combine XML-Schema, which supports structure modelling, and information extraction technique, which acts as tokenizer, to contribute a universal parser for structured documents.

In the following we will introduce firstly system design, which defines the framework of system. Afterwards, algorithm is described in detail. Experiments, conclusion will be added after implementation and evaluation.

**10.2 System Design**

Based on the observation discussed above, we developed the system of generic parser for structured free text documents. Figure 40 shows the architecture of the system for structured document
There are two main modules in the system: TE-Agent and NE-Agent. Actually, this contribution is similar as the system architecture of CapturePlus. But it works a little different:

- The NE-Agent is the same to the NE-Agent in CapturePlus. Actually, they are the same component. But the working step is a little different. In CapturePlus, NE-Agent extracts at first all instances for all elements defined in a template. The result of NE-Agent is a list of instances. This list is then fed to TE-Agent, the task of NE-Agent is therefore finished. In the XML-enabled template model, NE-Agent works with TE-Agent interactively. That is, these two components cooperate with each other. NE-Agent extracts only part of all instances and passes them to TE-Agent. TE-Agent fill the instances received from NE-Agent and then tell the NE-Agent to extract some other instances. This interactivity is performed until all instances are extracted and filled into template.
• The TE-Agent is totally different than the spatial model in CapturePlus. Here the template model is defined by XML-Schema, which has strict cordiality constraints. Furthermore, with help of XML-Schema, template model can be defined more complicated than simple flat tree structure. various cardinality of elements and duplication of part models become possible. The TE-Agent works as a parser by using stack automata and deterministic finite state automata.

Actually, this template model works very similar to a parser. The TE-Agent acts as parser and the NE-Agent as tokenizer. The main differences between our system and common XML parser are described as following:

• In an XML document, the next coming content can be exactly extracted by tokenizer, because it always enclosed by begin- and end tags. Then, XML parser can use XML-Schema to validate the extracted content. Unlike XML-documents, various structured free text documents do not have uniform structure definition. So that such validation is impossible because content can not be extracted with help of unique formatted tags. Therefore, in our system, the TE-Agent can not use XML-Schema for validation of current element. The clue is that XML-Schema is used for prediction of the next wanted element. This approach plays a key role in the system. For a common XML parser, XML-Schema is only used for validation. The validation is processed as following: the parse contains current information of element. According to XML-Schema, the next wanted possible elements can be retrieved. Then, the tokenizer extracts the next element, the validator can the check if the newly extracted element is one of next wanted possible elements. In our system, the XML-Schema is used to generate the next wanted possible elements. Once the next wanted element is known, NE-Agent can extract content for such element using corresponding extracting rules. Normally, the following element is not always unique (e.g. element with choice or mixed property, sub-element with cardinality etc.), a decision for selection of the right subsequence element is to be made.

• NE-Agent is much more complicated than a hard-coded tokenizer. For XML document, the task of tokenizer is simple: recognize the begin and end tags and then extracts the underlying content. But for structured free documents, there is no uniform definition for tags and contents. Therefore, extracting rules have to be developed for recognising tokens (similar as tags) and contents. For each element defined in XML-Schema, one or more corresponding extracting rules are to be defined.

The TE-Agent consists of a stack automaton, which performs context sensitive parsing, and a set of Deterministic Finite Automata (DFA) for all elements defined in XML-Schema, which are used to predict next wanted
element after current parsed element. The Information Extraction Module implements various extracting methods to find out tokens and underlying data.

System extracts and exports documents in three phases: design phase, loading phase and extracting phase. The working step is described as following:

- In the design phase, knowledge engineer develops XML-Schema and extraction rules for the structured documents. For each element in XML-Schema, a set of rules is developed. Such set of rules consists of extracting rule for recognizing token fields, for extracting underlying contents and for some constraints conditions etc. Because normally structured documents are rich data documents, regular expressions are suitable basic extracting technique. Combined with constraints and context information, such rules can extract the tokens and contents very effectively and efficiently. Rules are also written with XML. The developed XML-Schema and extracting rules can be saved in separated files but also be integrated in one XML document.

- In the loading phase, the TE-Agent loads at first the XML-Schema. Based on the defined XML-Schema, the TE-Agent constructs inner automates for parsing and prediction. For total parsing, a Stack Automata is generated with simple push and pop operation. For each element defined in XML-Schema, corresponding DFA is constructed. The DFA defines next possible coming elements after current element and their transition conditions. Simultaneously, NE-Agent loads total extracting rules and stores them in an inner table for future extracting.

- In the extracting phase, TE-Agent begins to parse the total document from the root. At first the next wanted element or a set of candidates of such elements based on current element is predicted. The prediction is made by evaluation current DFA. The information about predicted elements is then forwarded to NE-Agent. The NE-Agent loads then corresponding rules for each element and extracts its content from the document in the not consumed part of document. Extracting results are sent back to the TE-Agent after finishing extracting. TE-Agent makes a decision to choose the most suitable element instance as following element. The decision is made very simply: the element instance, which appears at first, is assigned as the following element. Note this element must conform to the XML-Schema because the “validation” is made before extraction already. Depending on the type of next element (sub-element, sibling element or parent-element), TE-Agent pushes/pops elements into/from Stack Automata and changes state in DFAs. The detailed algorithm will be

---

8 Note this simple decision could cause potential weakness of the system: the NE-Agent must have 100% precision. Deep discussion of this point is made in the section future discussion.
introduced in next section. Such processing is repeated until unrecovered errors occurred or the document is completely parsed. The output of processing is an XML file, which exactly conforms to the defined XML-Schema.

10.3 Algorithm

10.3.1 TE-Agent

We construct TE-Agent as a Stack Automata. It works like other common parsers with simple push and pop operations. Current element is pushed into the stack, after processing it is popped from the stack again. To predict wanted element after one element, we construct Deterministic Finite State Automata (DFS) to describe the possible following elements of current element. Actually, for each element a DFS is to be developed. There are totally one Stack Automata for parsing and N DFSs for prediction (similar as validation), where N is the number of elements in defined XML-Schema. The construction of Automata is processed in the loading phase. For Stack Automata, stack is initialised. For DFAs, states and transitions are investigated according to the definition of current element. Type of element (such as Choice, Sequences, Mixed. PDATA etc.) and its cardinality are analysed. The number of states is just the number of the sub-elements. The transition conditions are depended on the element type and cardinalities of sub-elements. Following example shows a sample DFAs for one element.

```
(ElementType name="AccountMovement" content="eltOnly" order="seq">
  <element type="Amount" minOccurs="1" maxOccurs="1"/>
  <element type="Source" minOccurs="1" maxOccurs="1"/>
  <element type="Target" minOccurs="1" maxOccurs="1"/>
  <element type="Commentar" minOccurs="0" maxOccurs="1"/>
</ElementType>
```

For this example, because the order is defined sequence, DFA can be constructed very easily: automata begin from a start state and go to the state “Amount”, and then “Source”, “Target”, “Comment” and finally to end state. Because the minimal occur of state “Comment” is 0, state From state “Target” DFA can also directly go to the end state. The description of DFA is shown in Figure 41.
Figure 41: An example of construction of DFA

Figure 42 gives the algorithm for construction the DFA according to XML-Schema.

<table>
<thead>
<tr>
<th>Load XML-Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all element type definitions</td>
</tr>
<tr>
<td>Get next element type definition</td>
</tr>
<tr>
<td>If contentType = “textOnly”</td>
</tr>
<tr>
<td>Construct DFS for text node</td>
</tr>
<tr>
<td>Else</td>
</tr>
<tr>
<td>If order is “seq” construct DFS with sequence property</td>
</tr>
<tr>
<td>If order is “choice” construct DFS with choice property</td>
</tr>
<tr>
<td>If order is “mixed” construct DFS with all possible links</td>
</tr>
<tr>
<td>Define the transition condition according to the cardinality</td>
</tr>
<tr>
<td>Link states, which have cardinality 0</td>
</tr>
<tr>
<td>End IF</td>
</tr>
<tr>
<td>End For</td>
</tr>
</tbody>
</table>

Figure 42: Algorithm for construction DFS according to XML-Schema

Once all DFAs are constructed, Schema Controller begins to parse. It works similar to a common parser with a stack automaton. The parsing algorithm is showed in Figure 43.

Note that, for each element instance, there is a corresponding DFA instance. To get the next possible element, the DFA instance for current element on the top of stack is at first investigated. Transition condition to next state is the cardinality of the state. Cardinality is defined by minimal and maximal occurs. If the current state has been visited more than minimal occurs but less than maximal occurs, next state can be subsequence elements or itself. Has the current state been visited more than maximal occurs, DFA has to left current state and go to one of next subsequence elements. If DFA reaches the end state, the next possible states must be investigated by parent DFA. This routine is called then recursively until all possible next elements are investigated. Figure 44 shows the function GetNextPossibleStates.
Load XML-Schema and Extraction Rules
Initialize Stack Automata
Construct Deterministic Finite State Automata for each element
Initialize DFA for root element
Push DFA for root element into stack
While stack is not empty and no parsing error
   Get DFA of current element on the top of stack
   If DFA is not in End state
      Copy a stack automata
      GetNextPossibleStates(CopyStack,ListElementName )
      Extracts all instances for possible elements
      Select the best instance as next element
      Update the current DFA in stack
      Create new DFA for new element
      push new DFA into stack
   Else
      Pop current element from the stack
   End If
End While

Figure 43: Algorithm for Parsing

GetNextPossibleStates(CopyStack,ListElementName )
   Get current DFA from top of CopyStack
   Get number of visited for current state in current DFA
   If number of visited < minimal occurs
      ListElementName.add (name of current state)
   Else if number of visited < minimal occurs
      ListElementName.add (name of current and names of all possible
      subsequence states)
   Else
      ListElementName.add (names of all possible subsequence states)
   End IF
If ListElementName contains End state
   CopyStack.pop
   GetNextPossibleStates(CopyStack,ListElementName )
End If

Figure 44: Function for getting next states
10.3.2 Examples

In following we give an example to illustrate how the XML-Schema enabled advanced template model works. Give is a simple template of bank transfer. The SML-Schema is defined as following:

```xml
<ElementType name="Kontobewegung" content="eltOnly" order="seq">
  <elementType type="Betrag" minOccurs="1" maxOccurs="1"/>
  <elementType type="Quelle" minOccurs="1" maxOccurs="1"/>
  <elementType type="Ziel" minOccurs="1" maxOccurs="1"/>
  <elementType type="Kommentar" minOccurs="0" maxOccurs="1"/>
</ElementType>
<ElementType name="Quelle" content="eltOnly" order="seq">
  <elementType type="Name" minOccurs="1" maxOccurs="1"/>
  <elementType type="KontoNr" minOccurs="1" maxOccurs="1"/>
  <elementType type="BLZ" minOccurs="1" maxOccurs="1"/>
</ElementType>
<ElementType name="Ziel" content="eltOnly" order="seq">
  <elementType type="Name" minOccurs="1" maxOccurs="1"/>
  <elementType type="KontoNr" minOccurs="1" maxOccurs="1"/>
  <elementType type="BLZ" minOccurs="1" maxOccurs="1"/>
</ElementType>
<ElementType name="Betrag" content="textOnly">...
<ElementType name="Name" content="textOnly">...
<ElementType name="KontoNr" content="textOnly">...
<ElementType name="BLZ" content="textOnly">...
<ElementType name="Kommentar" content="textOnly">...
```

In the loading phase, DFAs are generated. The corresponding DFAs, which illustrated in Figure 45, are contributed for each element defined in the XML-Schema. In the simple example, Only the element “Kommentar” is optional, so that the DFA for “Kontobewegung” has two ways after the element “Ziel”. The one is to element “Kommentar”, the other is to the end-element.
In the extracting phase, the IE-system loads document and perform the steps described above. Assumed that the document is parsed at the position “BLZ”. The document and state of IE-system is showed in following figures.
Element “BLZ” is at the top of stack automata. The next element in DFA “BLZ” is the end-element. So the stack automata pops up this element.

Figure 46: Worked example, step 1

Element “Ziel” is at the top of stack automata. The next element in DFA “Ziel” is the end-element. So the stack automata pops up this element.

Figure 47: Worked example, step 2
10.4 Experiments and Discussion

As a preliminary trial of the XML-Schema/IE hybrid parser, it was applied to three document collections of varying informality, the CIA Factbook, the Microsoft Excel comma delimited file (CSV) and Siemens product SIMATIC Hardware Configuration file.

The CIA Factbook provides information about countries and political organizations (see also Section 4.1). The XML-Schema for part of CIA Factbook is shown in Figure 49. In CIA Factbook, there are totally 248 elements defined in the complete XML-Schema.

The second experiment is made for MS Excel comma delimited file. One reason to parse such documents is that many product data processed by MS Excel. To exchange the Excel document with other applications, Excel documents are normally exported as comma delimited file (CSV). The CSV file separates data fields with user-defined comma. Parsing CSV file is very simple: define an XML-Schema for each head field and primitive rule with respect to comma separator. We tested various types of CSV files such as product catalogues and project schedulers etc.
Figure 49: XML-Schema for CIA Factbook

The last test document comes from Siemens Automation system SIMATIC. The hardware configuration software exports total hardware information to a configuration file (see Figure 50). Such data has to be imported into a database now. For this type of document, 60 elements are defined in XML-Schema. A real hardware configuration file is very big, normally more than 3000 lines.

The generic parser has parsed all documents described above successfully. The example sets for these three types of documents are at each case 100. The precisions and recalls of all of these three documents reached 100%.
In this section we introduced a new methodology for parsing and extracting information from structured documents. A new approach for XML-Schema-enabled generic parser is introduced. With combination XML parser and Information Extraction techniques various structured free text documents can be parsed and converted as XML-documents in a generic way. From the evaluation we showed that this technique can be used to parse a large variability of different structured documents. This technique enable engineer to convert different types of structured documents to XML by using configuration instead of programming.

10.5 Further discussion

The XML-Schema enabled template model is only implemented as a prototype and evaluated mainly for structured industrial documents. As a prototype there are some restrictions to XML-Schema. For example, nested model is not supported. Another restriction is ignoring all attributes. Actually, attributes are scarcely used in normal no-XML structured documents. In the future we will improve the model to support nested model and attributes. One of the most important weaknesses of this prototype is the strong assumption of a perfect NE-Agent. The TE-Agent does not implement any recovery to drop false instances extracted by NE-Agent (Tokenizer).

However, this idea can be extended to develop a more sophisticated Information Extraction system with combination of existing techniques used in CapturePlus. The advantage of XML-Schema enable template model is
obviously. The only limitation is, that currently XML-Schema is only used for strictly structured documents. Many application domains, which are in semi-structured gene, can not be processed in this approach directly. On the other hand, the spatial model works effectively to solve conflicts occurring in semi-structured document. Hence, the new template model should be able to combine these two techniques and provides a more powerful template filling ability. Figure 51 presents a design approach of new software architecture.

**Figure 51**: System Framework for generic XML-Schema-enabled template
Above is presented the main internal components of the Annotation Engine. These are briefly described below:

**Structure Reader** – this component is capable of reading in the hierarchical structure and constraints defined in XML-Schema and can initialize the Internal Data Structures that represent this information. The Structure Reader will also (optionally) read in the Validation Model, which is a meta-schema for the template model, in order to validate any template model that is imported into the IE-system. In the future, additional external formats (e.g. the representation of an NLP grammar) might be read in to define the structural constraints for the Annotation Engine, but this is left as future work for now. The Structure Reader has the (potentially complex) task of generating the Internal Model Data Structures (i.e. DFAs), which are based around the concept of a Finite State Transition graph.

**NE-Agent Recogniser** – this component is capable of reading the XML format of CapturePlus to extract out a set of recognisers i.e. different IE-Agents each specialized to recognize a specific named entity. In fact, most of the work in building a set of Recognisers, including the interpretation of the various search options, is carried out by the NE-Agent-Manager and all that needs to be maintained in the Recogniser Manager are the names of each Recogniser and how they are associated with specific parts of the Information Extraction Model.

**Internal Model Data Structures** – This data structure is specific to the new IE-system and reflects the structure and constraints of the template model, as well as being the representation of the instances of the content that are extracted from the input documents. This data structure is based around the concept of a Finite State Transition (FST) Graph.

**Parser** – this is the main component of the TE-Agent. It must:

- Interpret the structural constraints that have been imported from the Document Content Model, for example deciding when a particular part of the model should be replicated to represent the content of the current input document

- Apply all currently active Recognisers to the next region in the current input document

- Assess the reliability of results returned from the Recognisers –
  - Valid results are added to the Internal Model Data Structures
  - Ambiguous or erroneous results are dealt with by the Conflict Resolution Manager
  - When values are missed by the recognisers, this must also be detected and handled
**Conflict Resolution Manager** – this component receives the current state of a partially instantiated template model (XML-Schema) and the sequences of the most recent results returned by Recognisers. Some conflicts may arise for a number of reasons (E.g. more than one mutually exclusive recognisers returns results, mandatory values are missing, etc.) The Conflict Resolution Manager must assume that Recogniser performance is sub-optimal and must select the minimal local repair that resolves the conflict. This will typically be a heuristic reasoning based on the structural constraints that are known and/or the reliability of the recognisers. Note that some back-tracking for the Conflict Resolution Manager may also be needed.

This new approach can be considered as a new system design to contribute a commercial information extraction software (e.g. SemanticEdge).

**10.6 Related Work**

STALKER (Muslea 1999) considered that semi-structured information is often presented in the form of lists of tuples, with explicit separators used to distinguish the different elements. STALKER used embedded catalogue (EC) that describes the structure of a wide-range of semi-structured documents. The EC description of a page is a tree-like structure in which the leaves are the item of interest to the user. The internal nodes of the EC tree represents lists of k-tuples, where each item in the k-tuple can be either a leaf or another list (hierarchical representation). Actually, the EC-tree is very like the XML-Schema tree. However, EC does not support flexible ordering and cardinality setting. The lists in inner nodes define only sequence with cardinality one, while XML-Schema can model various ordering such as “sequence”, “choice” or “mixed type”. Likewise a flexible setting of cardinality is possible in XML-Schema.

Embley (1998) introduced his approach to extracting information from semi-structured documents is to uses so-called ontologies, which provide a kind of semantic description for supporting automatic information extraction. If the ontology is properly set up and properly applied to an HTML document that is suitable for the ontology, then fact extraction could works quite correctly and completely. That is, this approach works fine for unstructured documents that are data rich and narrow in ontological breadth. A document is data rich if it has a number of identifiable constants such as dates, names, account numbers, ID numbers, part numbers, times, currency values, and so forth. A document is narrow in ontological breadth if its application domain can be described with a relatively small ontological model. The input to the approach is an application ontology and an structured document, and the output is a filtered and structured data set in a database. Since all the processes and
intermediate file formats are fixed in advance, the framework constitutes a general procedure that takes as input any declared ontology for an application domain of interest and a structured document within the application's domain and produces as output structured data set, filtered with respect to the ontology.

There are three processes in framework: an ontology parser, a constant/keyword recognizer, and a structured-text generator. The input is an application ontology and a set of unstructured documents, and the output is a populated relational database. A main program invokes the parser, recognizer, and generator in the proper sequence. The ontology parser is invoked only once at the beginning of execution, while the recognizer and generator are repeatedly invoked in sequence for each unstructured document to be processed.

After invoking the parser, the main program invokes the constant/keyword recognizer and then the structured-text generator for each unstructured document. The recognizer applies each regular expression to the unstructured document. When the Perl program recognizes a string S according to a regular expression E with tag T, it emits T as the name, S as the string, and the beginning and ending character numbers in the document as the position, so called list the “name/string/position table.”

The structured-text generator uses the object/relationship/constraint list and the SQL schema to match attributes (object-set names in the ontology) with values (constants described in the name/string/position table). Then the generator forms tuples for relations in the generated SQL schema. The generator forms tuples according to five heuristics such as Keyword Proximity, Subsumed and Overlapping Constants, Functional Relationships, Nonfunctional Relationships and First Occurrence without Constraint Violation.

In comparison with Embley’s Ontology approach, our XML-Schema enabled template filling and Embley’s Ontology approach are both developed for structured documents. Both approaches use regular expressions and semantic constraints to extract contents. The outputs are both structured data set. In our approach the output data is represented as XML document, while the Embley’s system saves results in a database. However, there are some differences that can be described as following:

1. In Embley’s approach, the template is defined with Entity-Relationship, while in our approach the template is XM-Schema enabled. Both models support flexible cardinality. However, the entity-relationship is not able to define ordering property such as sequence, choice or mixed ordering.
2. The algorithm for extracting of XML-Schema approach is based on Stack Automata and Deterministic Finite State Automata. The extracting is here stepwise. That is, the template is filled during the extracting. To ensure right template filling system uses prediction of next wanted element by using DFA. The prediction can eliminate errors from the NE-Agent. For instance, if the next wanted element is a date, all other information before next coming date, which could be instances of some other elements, are passed out. The Ontology approach, on the other hand, extracts all instances of all elements in template at first and stores them in a “name/string/position table”. The template filling is processed after the extracting phase. In template filling, some heuristic methods are involved such as Keyword Proximity, Subsumed and Overlapping Constants etc. These heuristics can cause errors and the processing is less efficient.

Another system SMES developed by DFKI (Deutsche Forschungsinstitut für Künstliche Intelligenz GmbH.) is to provide a set of basic natural language components and generic linguistic knowledge sources which can easily be customized for processing different tasks in a flexible manner (Neumann 1997).- The main components are:

- A tokenizer based on regular expressions: it scans an ASCII text file for recognizing text structure, special tokens like date and time expressions, abbreviations and words.
- A German morphological component that performs morphological inflection and compound processing. POS tagging uses rules and Brill-based unsupervised tagger to solve the WSD problem.
- A declarative specification tool for expressing finite state grammars for handling word groups and phrasal entities (e.g., general NPs, PPs, or verb groups, complex time and date expressions, proper name expressions). A finite state grammar consists of a set of fragment extraction patterns defined as finite state transducers (FST), where modularity is achieved through a generic input/output device. FST are compiled to Lisp functions using an extended version of the compiler defined in (Krieger, 1987).
- A bidirectional lexical-driven shallow parser for the combination of extracted fragments.
- An interface to TDL, a type description language for constraint-based grammars (Krieger and Schäfer, 1994). TDL is used in SEMS for performing type-driven lexical retrieval, e.g., for concept-driven filtering, and for the evaluation of syntactic agreement tests during fragment processing and combination.
- A knowledge base, which collects different knowledge sources. According to (Neumann 1997), the knowledge base includes 120.000 lexical root entries, subgrammars for simple and complex date and time expressions, person names, company names, currency expressions, as well as shallow
grammars for general nominal phrases, prepositional phrases, and general verb-modifier expressions.

- A generic graphical editor for text items and an HTML interface to the Netscape browser, which performs marking of the relevant text parts by providing typed parentheses which also serve as links to the internal representation of the extracted information.

The template filling is performed in two steps: Fragment processing and Fragment combination/Template generation. The step Fragment processing is performed through fragment extraction patterns which are expressed as finite state transducers (FST) and which are compiled to Lisp functions using a compiler based on (Krieger, 1987). An FST consists of a unique name, the recognition part, the output description, and a set of compiler parameters. An FST operates on a stream of tokens. The recognition part of an FST is used for describing regular patterns over such token streams.

In order to obtain flexible control mechanisms for the matching phase it is possible to specify whether an exact match is requested or whether an FST should already succeed when the recognition part matches a prefix of the input string (or suffix, respectively).

The SEMS uses also some similar techniques as XML-Schema approach:

1. Both Stack Automata and Finite State Automata are used in SEMS in a similar way to the XML-Schema approach. To extract the useful contents, the Finite State Transducers (FST) are used. Next possible token is predicted based on FST in SEMS. Each variable of a basic edge denotes a stack, so that the current token is actually pushed onto the stack.

2. Both systems use searching constraints such as prefix and suffix (in XML-Schema approach the constraints are named as pre-token and post-token).

Two major differences between two systems are described as following:

1. SEMS is based on NLP technique, while XML-Schema approach uses syntactic rules (such as regular expressions with respect to context). Actually, the SEMS is a MUC system to extract information from unstructured free documents, while the system described here is industry oriented and faced to extract fine data from structured industry documents.

2. SMES’s template filling is a flat, non-hierarchical task. That is, the final goal of template filling in SEMS is to fill information into a scenario (such as a terrorist event). Hierarchical relationship between elements and cardinality of a element are not the interest of SMES.