NoDoSE - A tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents.

Brad Adelberg[*]

Abstract

Often interesting structured or semistructured data is not in database systems but in HTML pages, text files, or on paper. The data in these formats is not usable by standard query processing engines and hence users need a way of extracting data from these sources into a DBMS or of writing wrappers around the sources. This paper describes NoDoSE, the Northwestern Document Structure Extractor, which is an interactive tool for semi-automatically determining the structure of such documents and then extracting their data. Using a GUI, the user hierarchically decomposes the file, outlining its interesting regions and then describing their semantics. This task is expedited by a mining component that attempts to infer the grammar of the file from the information the user has input so far. Once the format of a document has been determined, its data can be extracted into a number of useful forms. This paper describes both the NoDoSE architecture, which can be used as a test bed for structure mining algorithms in general, and the mining algorithms that have been developed by the author. The prototype, which is written in Java, is described and experiences parsing a variety of documents are reported.

Keywords: data extraction, semistructured data, structure mining, wrapper induction.

1 Introduction

The amount of useful semistructured data [Abi97] on the web continues to grow at a torrid pace. Users would like to gain conventional database system functionality on this data such as sophisticated querying and reporting. This has spurred a recent flurry of work [KWD97,AK97a,HGMC$^+$97, DEW97] on generating wrappers around such sources, either manually or with software assistance, to bring the new data within the reach of general query tools. It is important to note, however, that a vast quantity of semistructured data stored in electronic form is not in highly formatted HTML pages but in text files on local file systems. Examples are mail files, code and code documentation, configuration files, logs of program activity, phone lists, etc. Further, there is a huge collection of semistructured information in print, that when scanned and OCRed will be in a plain text file, not one with HTML tags. Thus if we want to extract all of the data that's important to users through a query interface, we need to focus on something more general than HTML files — plain text files. Since HTML files are a special case of text files, a tool that handles text files will also handle HTML files.

Extracting information from text files is harder than for HTML files for three reasons:

1. Since text files do not generally contain markup tags, there are usually fewer structural clues and those that are present are not known a priori.

[*]Northwestern University Computer Science Department. Email: adelberg@cs.nwu.edu. Address: 1890 Maple Avenue, Evanston IL 60201. Telephone: (847) 467-2129. Fax: (847) 491-5228.

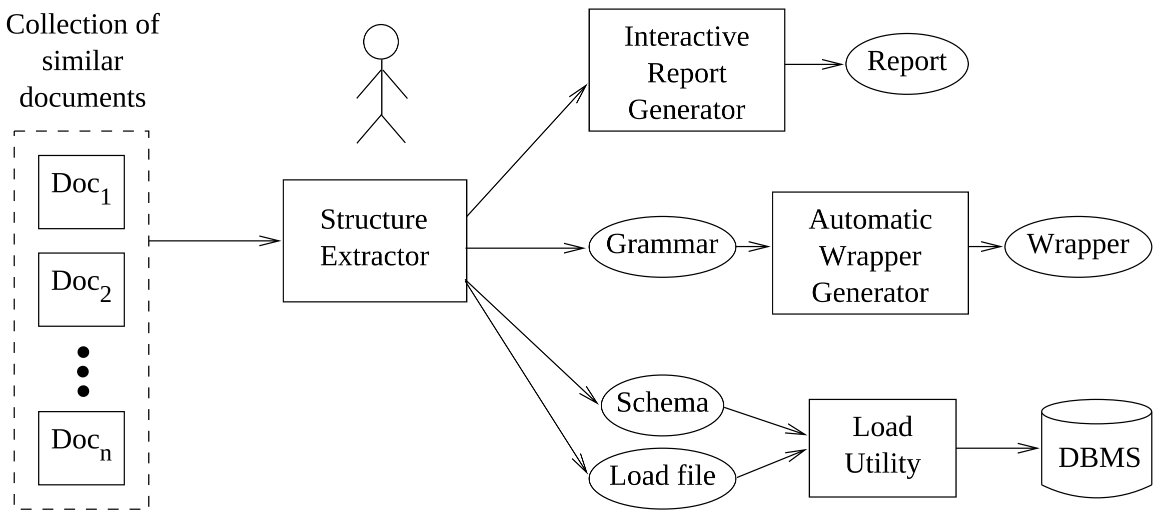


Figure 1: User level architecture.

2. Not all of the structural elements will be separated by markers (such as tags). Thus an extraction tool may have to consider parsing rules such as "list elements are exactly three lines long" as opposed to HTML where they begin after <li> and end before another list tag or an end list tag (i.e. </ul>).

3. Now that most HTML pages are composed using authoring tools, their format (e.g. spacing, capitalization, etc.) is very regular. Many text files, in contrast, are typed by humans and are thus less regular and may contain errors. Also, text files scanned from documents will almost undoubtedly contain misrecognized characters.

Given the complexity of parsing general text files, the likelihood of a fully automatic extraction system like those proposed for HTML files [KWD97] seems remote. With no clues about the format of a document, the system will not be able to differentiate useful data from junk. We note, however, that a few clues can go a long way: If a user indicates a few of the regions of a document that are interesting it may be possible to identify similar regions automatically.

Thus the approach we have taken is to build a *semi-automatic* system, one that cooperates with the user to extract the data. Figure 1 illustrates how the process would work. The input to the extractor is a set of text files (documents) that are instances of the same document type. For example, the files might be reports generated by a weekly file backup program for the past year. Using a GUI, the user hierarchically decomposes the files, outlining their interesting regions and then describing their semantics. This task is expedited by a mining component that attempts to infer the grammar of the file using the information the user has input so far. A detailed example of this process is presented in Section 2.

Once the format of the document type has been determined and verified by successfully parsing all of the input documents, the extractor should be able to generate different types of output:

1. Using a simple interactive report generator, the user could filter the data from the input documents and write it out in a new format. For example, the user may want to convert the

input into a comma delimited file suitable for importation into a data analysis program or spreadsheet.

2. If the extracted data is to be stored in a database, a schema file (if the data is structured) and a load file suitable for the load utility that comes with the DBMS can be generated.

3. If the input documents are to be exposed through a query interface, the Lex and Yacc code needed by a wrapper can be generated.

In this paper we describe the design and implementation of this extraction tool — NoDoSE, the Northwestern Document Structure Extractor. We cover both the NoDoSE architecture (in Section 3), which can be used as a test bed for structure mining algorithms in general, and the algorithms for inferring parsing rules that we have developed (in Section 4). A first version of NoDoSE has been implemented and is described in Section 5 along with a description of our experiences using it to extract data from a variety of documents.

2 Example

To illustrate the use of NoDoSE we consider the problem of analyzing the results of simulation experiments. Although recent work on data extraction [KWD97,AK97a,HGMC$^+$97] has focused on web pages, we choose simulation output here since it is meant to be human readable and not program readable. Thus it has fewer structural clues (such as tags) and is more difficult to determine parsing rules for. Also, we anticipate that a system capable of inferring parsing rules for plain text documents will be able to handle HTML documents as well.

An example of human readable simulator output (which was generated by DeNet [Liv90]) is shown in Figure 2[1]. Most tools for storing and analyzing data, such as database systems, spreadsheets, and plotting programs, cannot handle files this complicated. Instead, the output file must be converted into a more regular file (i.e. tabular) before it can be processed. Typically, this conversion is performed by a hand-coded program in awk, sed, perl, or some other scripting language. Using NoDoSE, the conversion can be performed quickly and without any coding expertise.

There are three steps to the conversion process:

1. Decide on how to model the data in the documents.

2. Hierarchically decompose the files, mapping regions of the files into components of the chosen model.

3. Specify how the extracted data is to be output.

We describe each of the three steps in the sections below.

[1]The output as shown is slightly modified from the original. For illustration purposed, node results relating to confidence intervals were removed since they cannot be represented as a <average,stdev,num> triple. NoDoSE can parse the original files if a different data model is chosen.

```
DeNet(V1.6) simulation started on Mon Sep  5 14:41:31 1994
...
Attributes of node # 0 (Principal)
    simTime - 5.000000E+02
Attributes of node # 1 (usource)
    arrivalRate - 2.000000E+02
    meanSkewL - 1.000000E-01
    meanSkewH - 1.000000E-01
...
Attributes of node # 5 (tsink)
    batchSize - 100
    confidenceLevel - 9.500000E-01
    confidenceInterval - 1.000000E+00
Sample Results Node # 1 (usource)
  1  numUpdates - (avg) 1.000000E+00 - (std) 0.000000E+00 - (num) 100431
Sample Results Node # 2 (tsource)
  2  numJobs - (avg) 1.000000E+00 - (std) 0.000000E+00 - (num) 2455
...
Sample Results Node # 5 (tsink)
  5  misDL - (avg) 9.295315E-01 - (std) 2.559869E-01 - (num) 2455
  5  missedStale - (avg) 9.295315E-01 - (std) 2.559869E-01 - (num) 2455
DeNet(V1.6) simulation terminated on Mon Sep  5 14:42:24 1994
 Total CPU usage     0.896 Minutes. (user     0.894 ; system     0.002 )
```

Figure 2: Simulation output example.

2.1 Modeling the documents

Before extracting data from the documents the user must decide how to model the data. One possibility for the data, and the one that will be used in this example, is shown in Figure 3. Documents are of type SimulationRun and contain three top-level components: a timestamp, a list of input parameters for each simulation node, and a list of measured results for each node. The parameters for each node (part of NodeParams) are represented as a list of <name,value> pairs. This has the benefit of a very regular structure but the disadvantage that the type information about parameters is being lost since every value is modeled as a string. The structure is so regular, in fact, that the output of any simulator written in DeNet can be modeled using this schema.

Instead of representing all of the different simulator nodes in a generic way, we could also create a new class for each. This solution has the benefit that we can model the particular parameters of each node and their real types. It also has two problems: any change to the simulator will require a change to the model, and the grammar derived for the output of one simulator cannot be used on the output of another DeNet simulator. Thus because it is more general and because it is more difficult to parse, we will use the model from Figure 3 in this example. NoDoSE, however, can work with either.

```
interface NodeParams {
attribute int node_number;
attribute string node_name;
attribute List<Struct OneParam {string name, string value}> parameters;
}
interface NodeResults {
attribute int node_number;
attribute string node_name;
attribute List<Struct OneResult {string name, real average, real std, int num}> results;
}
interface SimulationRun {
attribute String timestamp;
attribute List<NodeParams> node_params;
attribute List<NodeResults> node_results;
}
```

Figure 3: Example schema for the simulation output.

2.2 Decomposing the documents

The decomposition process begins by loading a single document into NoDoSE. The user then hierarchically decomposes the document using a GUI. Next additional documents of the same type are loaded in to the system and automatically parsed. Any errors are corrected by using the GUI and reparsing. The process is complete when all of the documents have been successfully parsed.

The first step in decomposing a document is indicating its top level structure, in this case a record of type SimulationRun. Next, we add each of its three fields (timestamp, node_params, and node_results) by selecting the relevant portion of the text in the document window and clicking on the add structure button in the tool bar (Figure 4). The type, type name, and label of each field can be entered using the controls on the bottom portion of the window. Since node_params and node_results fields are complex types (lists), the decomposition process must continue.

Suppose the user chooses to decompose the list of node results next. Double-clicking on that node in the tree view panel will display only the portion of the document mapped to the node_results list. The user then selects the text of the first element of the list (the first two lines) and adds this as a structure. Next, the second element of the list is added. Figure 4 shows a snapshot of the interface at this point. The user could continue to add every element in this manner but this would become tedious if there are many elements. Instead, the user can ask NoDoSE to try to infer the remaining elements by mining the text. If the tool mistakenly identifies elements the user can correct a few of the errors and ask that the text be remined. In this way, the correct grammar for the component will eventually be learned. Once it is, NoDoSE is able to identify all of the other elements of the list correctly.

The decomposition process must continue since each element of the node results list is a record of type NodeResults. Any of the list elements can be selected and its fields added. Figure 5 shows the screen after the third element has been decomposed. As before, the user does not have to decompose every element by hand. Once a few elements (in this case, one) have been decomposed, the miner can be again invoked to decompose every record of type NodeResults. Since the parameters field of NodeResults is itself a list, the process must continue (node_params must also be decomposed). The process continues until all of the leaves of the document tree are atomic types.

After the grammar for a particular file has been determined, NoDoSE is loaded with all of the other files of the same type. These are automatically parsed using the grammar inferred from the first file. It's possible, though, that parsing fails on one of the additional files for one of two reasons. First, the additional file may contain an error such as a mistyped field name or an OCR error. In such a case, the user can correct the error through the GUI but the grammar does not need to be updated. The second reason why parsing may fail is that the additional files contain something that was not present in the first file parsed. For example, suppose that the files come from two different versions of the simulator and that the newer version measures and outputs an additional value. If a file from the old version was used for the initial parsing process, NoDoSE will fail to recognize the new field. In this case, the user must correct the parsed tree for the new file, describing the new field using the GUI as before. The extractor will then update its grammar to account for the new field. After this, any of the files with the newly described field will be automatically reparsed. When all of the files of the same document type have been successfully parsed, the first step of the conversion process is complete.

2.3 Outputting the extracted data

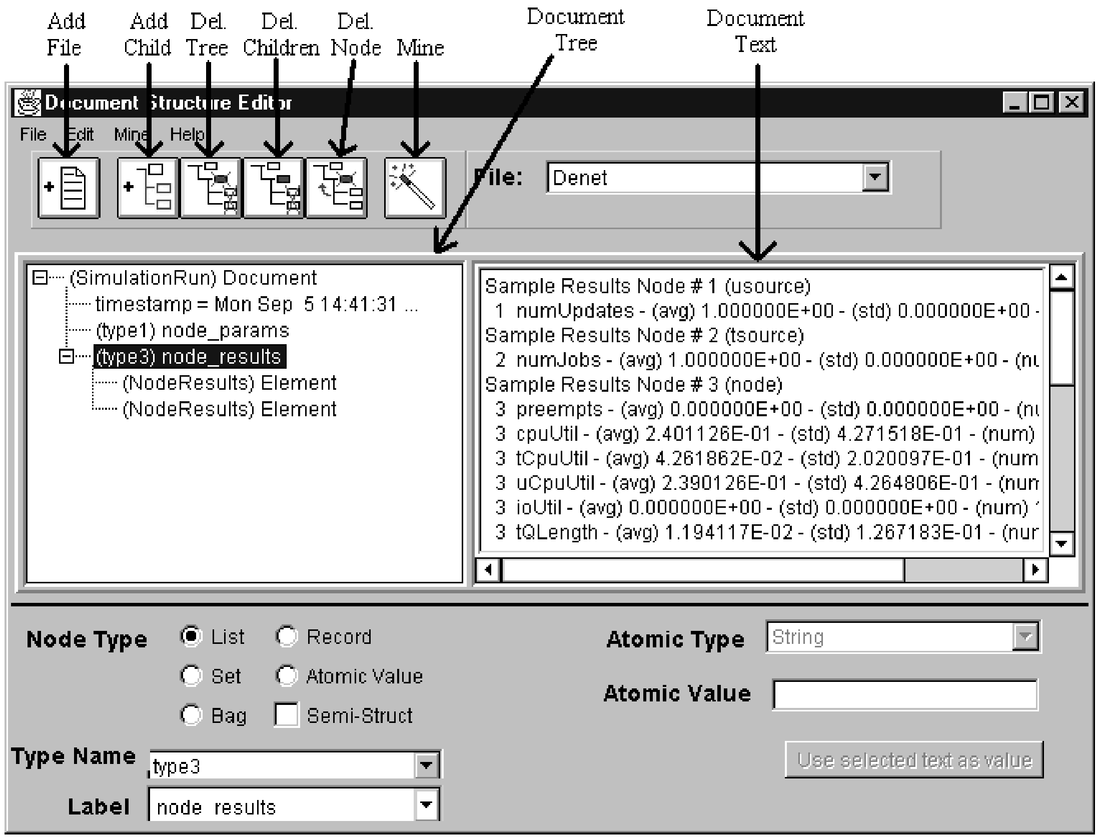
The final step of the conversion process is to specify how to output the data that has been extracted from the parsed files. As shown in Figure 1, different options are supported. One option is to write the data into a text file. The format of the file and which data to be output is specified using a simple GUI-based report generator. The intent of this component is not to replace the querying and reporting functions of a DBMS but to provide a quick means of writing simple files, such as comma or tab delimited tabular data for input to spreadsheets and the like. For users who need to perform more complex operations on the data, NoDoSE can generate a schema file and a load file for use by a load utility provided by a third party DBMS. At the present, the generated schema file is ODL-like and the load file is in a generic format of our design. Additional formats can be added either by using the report generator or by coding an additional report component.

3 System Architecture

This section describes the internal architecture of NoDoSE in two parts: it first describes how the structure of documents is represented and then gives an overview of the components that comprise the system.

3.1 Document Model

Externally, documents are represented as flat files which serve as the input to NoDoSE. Internally, however, we need to be able to store information about the structure of the documents. Hence for every file that is loaded by the user, NoDoSE maintains a tree that maps the structural elements of the document to the text of the file (Figure 6(a)). Each node of the tree represents one of the



File: Denet

```
⊟ (SimulationRun) Document
    timestamp = Mon Sep  5 14:41:31 ...
    (type1) node_params
  ⊟ (type3) node_results
      (NodeResults) Element
      (NodeResults) Element
```

```
Sample Results Node # 1 (usource)
 1  numUpdates - (avg) 1.000000E+00 - (std) 0.000000E+00 -
Sample Results Node # 2 (tsource)
 2  numJobs - (avg) 1.000000E+00 - (std) 0.000000E+00 - (nu
Sample Results Node # 3 (node)
 3  preempts - (avg) 0.000000E+00 - (std) 0.000000E+00 - (nu
 3  cpuUtil - (avg) 2.401126E-01 - (std) 4.271518E-01 - (num)
 3  tCpuUtil - (avg) 4.261862E-02 - (std) 2.020097E-01 - (num
 3  uCpuUtil - (avg) 2.390126E-01 - (std) 4.264806E-01 - (num
 3  ioUtil - (avg) 0.000000E+00 - (std) 0.000000E+00 - (num)
 3  tQLength - (avg) 1.194117E-02 - (std) 1.267183E-01 - (nur
```

Node Type: [✓ List] Record Set Atomic Value Bag Semi-Struct

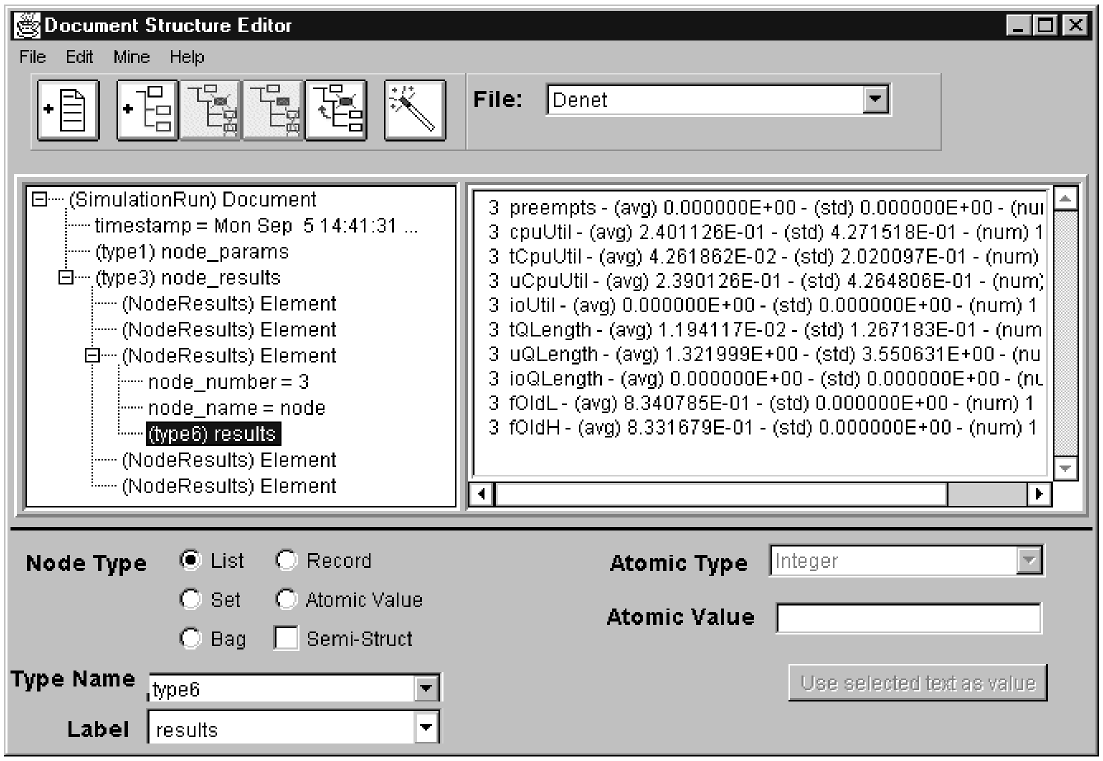
Atomic Type: String

Atomic Value:

Type Name: type3

Label: node_results

Figure 4: Screen shot of NoDoSE after a few steps.



File: Denet

```
⊟ (SimulationRun) Document
    timestamp = Mon Sep  5 14:41:31 ...
    (type1) node_params
  ⊟ (type3) node_results
      (NodeResults) Element
      (NodeResults) Element
    ⊟ (NodeResults) Element
        node_number = 3
        node_name = node
        (type6) results
      (NodeResults) Element
      (NodeResults) Element
```

```
 3  preempts - (avg) 0.000000E+00 - (std) 0.000000E+00 - (nu
 3  cpuUtil - (avg) 2.401126E-01 - (std) 4.271518E-01 - (num) 1
 3  tCpuUtil - (avg) 4.261862E-02 - (std) 2.020097E-01 - (num)
 3  uCpuUtil - (avg) 2.390126E-01 - (std) 4.264806E-01 - (num)
 3  ioUtil - (avg) 0.000000E+00 - (std) 0.000000E+00 - (num) 1
 3  tQLength - (avg) 1.194117E-02 - (std) 1.267183E-01 - (num
 3  uQLength - (avg) 1.321999E+00 - (std) 3.550631E+00 - (nu
 3  ioQLength - (avg) 0.000000E+00 - (std) 0.000000E+00 - (nu
 3  fOldL - (avg) 8.340785E-01 - (std) 0.000000E+00 - (num) 1
 3  fOldH - (avg) 8.331679E-01 - (std) 0.000000E+00 - (num) 1
```

Node Type: [✓ List] Record Set Atomic Value Bag Semi-Struct

Atomic Type: Integer

Atomic Value:

Type Name: type6

Label: results

Figure 5: Screen shot of NoDoSE later in the process.

structural components of the document such as an element of a list or a field in a record. The following values are stored in each node:

typeName - Every node in the tree, and thus every component of a document, must be of either an atomic type or a named composite type. Details of the type system supported by NoDoSE are described below.

startOffset, endOffset - These two values indicate which portion of the file corresponds to the structural component. For non-root nodes, the offsets are relative to the start of the parent node's region.

label - The only required use of this field is in the children of record nodes to indicate which field the node represents. Labels can also be used to represent data in a schema-less model such as OEM [CGMH$^+$97].

authorId - This identifies the creator of the node, the user or one of the mining components. Maintaining the originator of a node is useful when mining structure since user identified regions can usually be given greater credence than regions identified by the mining components.

confidenceValue - This is a value between 0 and 1 indicating how confident the author is that this node is correct. It is typically set to 1, meaning complete confidence, for nodes added by the user and is set to a lower value for nodes inferred by one of the mining components. One practical use of the confidence value is to alert the user about nodes that may not have been parsed correctly (see Section 4.1.1).

To clarify the mapping process, consider the file shown in Figure 6(a) that is composed of just a single line. We can view this file as a list of names, each name being composed of a first and last name. This structure would be represented by the tree shown in Figure 6(b). The root of the tree represents the whole document and is mapped to the entire file. The root is of type *Doc* which is a list of objects of type *Name*. The root has three children, each corresponding to one of the names in the list, and in the same order as the corresponding names appear in the document. Each child is of type *Name*, which is a structure with fields for the first and last name. Of course each node has a different pair of offset values to indicate where in the text the element is.

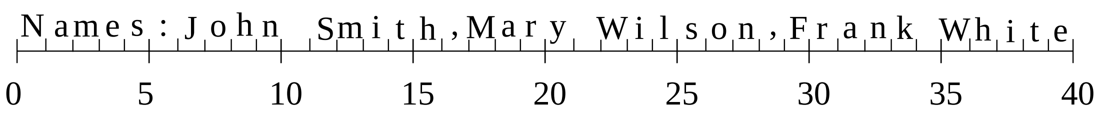
Finally, each node of type *Name* has two children corresponding to the two fields in the record. Each child is of type *String* which is atomic and hence they have no children themselves. Also, each child node has a label that identifies which field of the parent node it contains the value for. Note that all of the firstName nodes have a start offset of 0. This is because offset are relative to the parent node's text (and, in this case, the first name value begins every list element).

Every node in a document tree must be associated with a particular type. NoDoSE predefines six atomic types: *Integer*, *Float*, *String*, *Date*, *EmailAddress*, and *URL*. Additional atomic types can also be added — the user need only supply a name. Complex types can be defined as well using the following common type constructors:

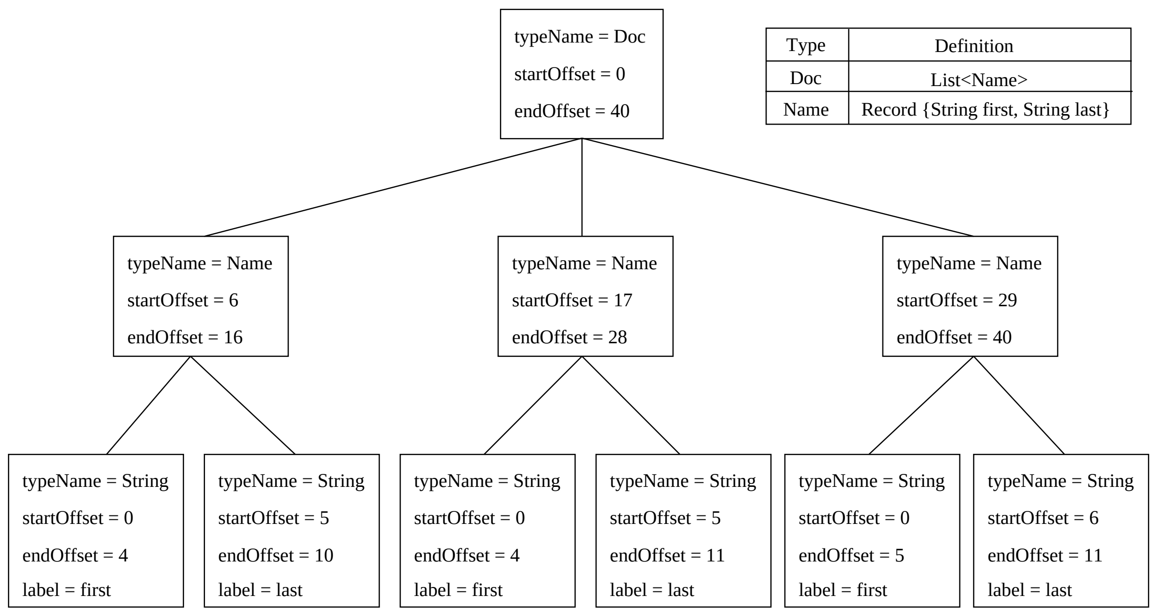
1. *NewType* = Set<*OldType*>,

2. *NewType* = Bag<*OldType*>,

3. *NewType* = List<*OldType*>,



(a) Example file.



Type	Definition
Doc	List<Name>
Name	Record {String first, String last}

(b) Document tree for example file.

Figure 6: Representation of a document.

4. $NewType$ = Record$\{OldType_1\ fieldName_1, OldType_2\ fieldName_2, ...\}$.

Note that unlike some type systems, only singly nested types can be defined in a single step. Thus defining the new type List<Record{String first, String last}> would require two new types, one for the record and one for the list.

In addition to the structured type constructors, NoDoSE provides an analogous set of type constructors for semistructured data: SemiSet<>, SemiBag<>, SemiList<>, and SemiRecord{ $fieldName_1$,$fieldName_2$,...}. These constructors do not restrict the type of their components so, for example, all of the elements of the list do not have to be of the same type. Note that many of the previously proposed models for semistructured data do not require all four constructors. For instance, OEM[CGMH$^+$97] objects can be represented using only atomic types and SemiList. We have added them, though, for cases where more semantic information is known.

Having covered the type system we can now define what constitutes a legal document tree. For a tree to be legal all of its nodes must be legal. A node n is legal if and only if all of the following conditions hold, the first four of which are related to type restrictions and the last of which is related to mapping:

1. if n is an instance of an atomic type it cannot have any children;

2. if n is an instance of a structured collection (List, Bag, or Set) all of the children of n must have the same type;

3. if n is an instance of a Record type defined as the set of fields $F = \{< t_1, f_1 >, < t_2, f_2 >, ..., < t_m, f_m >\}$ and n has the children $c_1, c_2, ..., c_k$:

   (a) $(\forall i)[< c_i.typeName, c_i.label > \in F]$,

   (b) $(\forall i, j)[(1 \leq i \leq k) \wedge (1 \leq j \leq k) \wedge (c_i.fieldName = c_j.fieldName) \rightarrow (i = j)]$.

4. if n is an instance of a SemiRecord type defined as the set of fields $F = \{f_1, f_2, ..., f_m\}$ and n has the children $c_1, c_2, ..., c_k$:

   (a) $(\forall i)[\{c_i.label\} \in F]$,

   (b) $(\forall i, j)[(1 \leq i \leq k) \wedge (1 \leq j \leq k) \wedge (c_i.label = c_j.label) \rightarrow (i = j)]$.

5. let p be the parent of n, l its left sibling, and r its right sibling. The following must hold:

   (a) $0 \leq n.startOffset < n.endOffset \leq parentLength$ where $parentLength$ is taken to be $p.endOffset - p.startOffset$ if p exists and the length of the document otherwise;

   (b) if l exists, $l.endOffset \leq n.startOffset$;

   (c) if r exists, $n.endOffset \leq r.startOffset$.

It is the responsibility of the *Instance Manager* and *Document Manager*, described below, to ensure that every tree instance is legal.

3.2 Components

NoDoSE is intended to be a test bed for studying the data extraction problem. Thus, rather than build a monolithic tool and force other researchers to wade through thousands of lines of source code, we've designed the system as a set of components that communicate through Java interfaces. Any of the components can be replaced independent of the others, and for certain types of components, more than one can be instantiated at any given time. At the present, changing components still involves changing a few lines of code in the top level NoDoSE class but with in the next version we plan to use the Java Reflection class to allow components to be changed or added dynamically without any code changes.

Figure 7 shows most of the components in NoDoSE and how they interact. The reporting component has not been shown in the interest of readability but will be discussed below. Most of the components provide the basic infrastructure for the system: reading files, maintaining document trees and type information, and supporting undo/redo. We expect that these components, which we collectively call the *support components*, will rarely be the subject of experimentation. The remaining three components are those most likely to be changed: the structure miners, the report generators, and the GUI.

Allowing third parties to build components that modify the data is dangerous since they may accidentally violate constraints. For example, a GUI may allow the user to add a child to an atomic type. To avoid these problems we have adopted the model-view-controller [Gol90,KGP88] paradigm. The model, which stores the data and enforces constraints, is maintained by the support

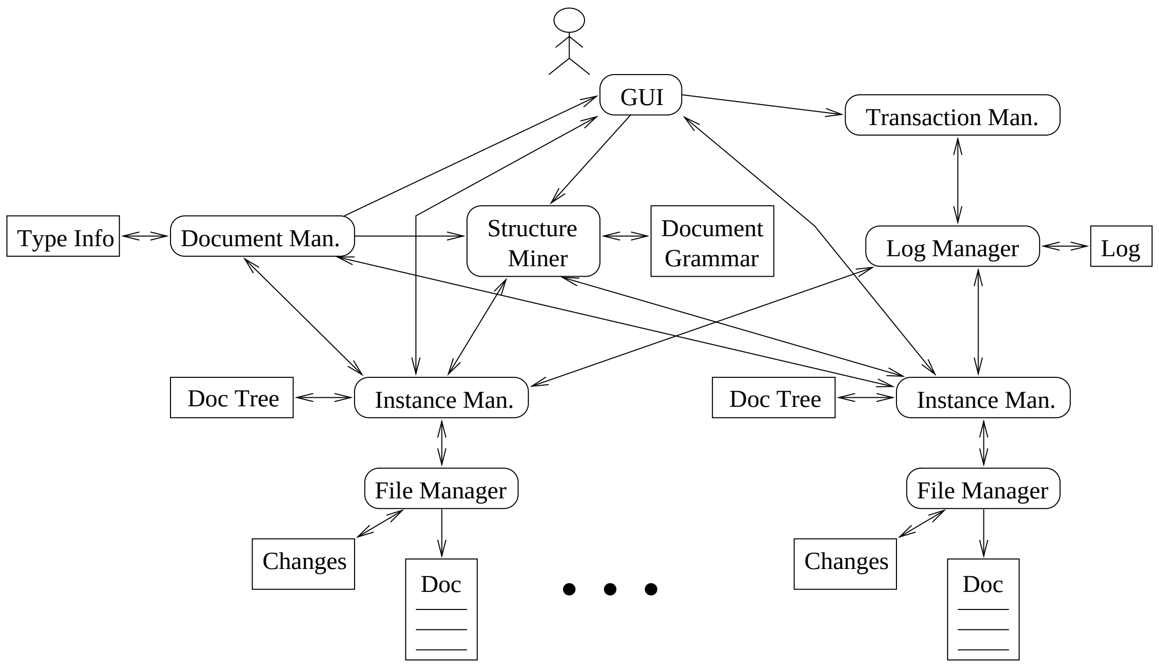


Figure 7: Internal architecture.

components. The other three components types, Reporters, Miners, and GUIs, all provide views of the data in the model. In addition, the miner and GUI both serve as controllers since they modify the data. This must be done through the model, however, which guarantees that no constraints can be violated. If a controller performs an an action that would violate a constraint, an exception is raised by the model which can be trapped and handled by the controller.

Because there can be many views on the model, the core components also support the observer/observable paradigm [GEH+94]. For example, the type manager allows other components to register as observers of a particular type. Whenever a new instance is added to that type or an instance deleted, the observers are notified. The mining component described in Section 4 uses this notification to incrementally maintain its statistics about a given type.

Below we describe all of the components of the system. Details of the version 1.0 implementation can be found in Section 5.

File Manager - Enables sections of a file to be read or modified by the other components. Modifications are not performed directly on the original file; a separate file of changes is maintained. This ensures that NoDoSE cannot corrupt an input file and that it can work with read-only files or files residing on remote machines. There is a one to one mapping between file managers and files.

Instance Manager - Maintains the document tree for a file, providing all of the basic tree manipulation operations, such as node insertion and deletion. It also provides methods that map the tree to the file or vice versa. For example, when a user double clicks in the document text panel, the GUI can use the instance manager to find the tightest bounding node for the point in the file so that it can display its type information. A particular instance manager stores only a single tree so every file must have its own.

Document Manager - Maintains information about a class of document (i.e. the output of a particular simulator). In particular, it stores a list of all of the files of a particular document class, as well as information on all of the types used in the files. The document manager stores four pieces of information for each type in the system: its name, its definition, a list of all of the nodes of the type, and a list of *observers*, which are the components that want to be notified when any of the type information changes.

Log Manager/Transaction Manager - Supports user undo/redo.

Structure Miner - Attempts to automatically determine how to parse a given node type. This component is described in much more detail in Section 4.

GUI - Unlike in many systems, the GUI in NoDoSE is truly a replaceable component. In fact, we are currently working on an alternate user interface for structured documents through which the user first specifies the schema of the document in ODL and then maps the text of documents to the schema. In a sense, this is the opposite of the current approach.

Reporter - Outputs extracted information, usually as a report, a load file, or as information needed by a wrapper generator. Multiple Reporter components can be active within one system to give the user different output options.

4 Mining for Structure

This section describes the two mining/parsing components that have been implemented so far: one that mines text files and one that parses HTML code. Both components are limited in scope; The text miner only handles structured types and the HTML parser does not handle frames or other advanced features. Despite their limitations, however, we have been able to extract data from an interesting set of documents. Further, building two different mining components has forced us to ensure that the interfaces exposed to the mining components are powerful enough and clean enough to support different algorithms. Details of both mining components appear below.

4.1 Plain text miner

The component described in this section attempts to determine the parsing rule for instances of a type. The particular type being mined in any invocation is called the *target type*. For this first version of NoDoSE, we chose to concentrate on mining structured types (Set, Bag, List and Record) since the type of the children nodes are known by definition. After we develop robust algorithms for this mining problem we plan to study the mining of semistructured types.

Another simplification in the current version is that we use the same algorithm for all of the collection types (Set, Bag, and List) since the semantic differences between them are rarely noticeable at the level of the format of the text file. Thus, for the remainder of the section, when we discuss mining Lists our comments will be equally valid for mining any collection type.

The algorithms for mining lists and records are both based on the same overall three step strategy:

Meaning	*Notation*
Begin with marker	[**With Marker** "marker"]
Begin after marker	[**After Marker** "marker"]
Begin at fixed offset	[**Offset** offset]
End with marker	[**With Marker** "marker"]
End before marker	[**Before Marker** "marker"]
End after a fixed # of lines	[**After Lines** num_lines]
End at fixed offset	[**Offset** offset]

Table 1: Parse rule components for the plain text miner.

1. **Theory generation -** Create a set of theories for how to parse the instances of the target type. For a list type, a simple theory is "each element will be separated by a comma".
2. **Theory evaluation -** For each theory under consideration,
   (a) Parse every node that is an instance of the target type (a list of which can be retrieved from the Document Manager) to generate a list of predicted children nodes. Note that if multiple documents are loaded, not all of the nodes will necessarily be from the same document.
   (b) Compare the predicted nodes to the nodes that are actually present in the document tree. Count the number of nodes in the document trees that were not predicted, which we call *false negatives*, and the number of predicted nodes that cannot possibly be correct, which we call *false positives*.
3. **Theory application -** If one or more of the theories has no false positives or negatives, pick one of them and add its predicted nodes to the document tree (or trees, if more than one document is loaded).

Although we will need different types of theories for parsing lists and records, the two share common elements: In each case, we are trying to subdivide the portion of the file corresponding to a given node, which we call the *node text*, into smaller units, each either a list element or a record field. To find the boundaries of the units we will need two theories: a theory about how the beginning of a unit is determined, called a *start theory*, and a theory about how the end of a unit is determined, called an *end theory*. We call the combination of a start theory and an end theory a *unit theory*.

Table 1 lists all of the start and end theories used in the current implementation of NoDoSE's text mining component. Most include a variable that must be instantiated, often a marker which is a string that separates units. For example, in the file from Figure 6, the best start theory would be [**After Marker** ","]. We could use something similar for the best end theory: [**Before Marker** ","]. To represent the resultant unit theory, which is the combination of the start and end theory, we will write <[**Before Marker** ","],[**Before Marker** ","]>. Rather than explain the meaning of the rest of the theories here, we introduce the concrete problem of parsing lists to give the discussion more context.

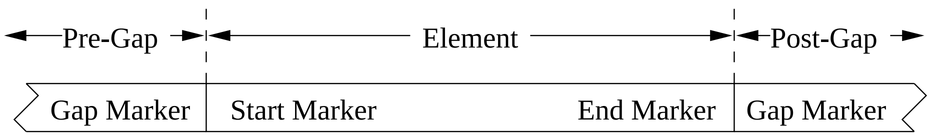


Figure 8: Presumed format of a list element.

4.1.1 Mining lists

Mining a list type entails two tasks: find a parsing rule that identifies every element of a list of the target type and then parsing every instance of the target type using the rule. Figure 9 shows the three instances of a target type, Roster = List<*String*>, that will serve as a running example for this section. The lists is meant to represent the rosters of basketball teams[2]. The boxes around parts of the lists indicate the elements that the user has already identified.

The mining algorithm for list types depends on three assumptions:

1. Every element of the list will have the same type. Since this algorithm will only run on structured collection types, this assumption must hold.

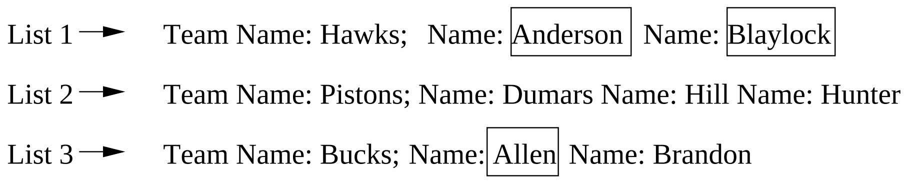
2. Every element of the list will have the same format. This assumption does not necessarily hold but seems reasonable given assumption 1 and simplifies the grammar induction.

3. If k elements of a list have been identified by the user, they will be the first k elements in the list. This is a very powerful assumption because it gives the miner a way to identify theories that generate false positives — no predicted unit can appear before any preexisting unit in a list. It does, however, impose restrictions on how structure information must be input by the user.

Of course all three assumptions hold for our example. The first holds by definition: all of the elements are of type String. The second holds as well since each element has the basic format "Player Name: *name*". The third also holds since list 1 has all of its elements specified, list 2 has none of its elements specified, and list 3 has only one element specified but its the first element. An example of a violation of assumption 3 would be if list 2 had the player named Hill specified without having Dumars specified as well.

Lists can be viewed in general terms as a header followed by the elements of the list separated by gaps. Of course, specific list types may not have headers or gaps at all. Figure 10 shows how list 1 of our example fits this pattern. For any list with at least one element defined, we know the boundaries of its header — everything to the left of the first element by assumption 3 above. Also, everything between two defined elements is necessarily a gap since no other element could exist between the two according to assumption 3. Thus even if some of the list instances have no elements defined and others have only some of their elements defined, the miner will usually still be able to identify a few headers and gaps (if the list text has them). In our example, the headers of lists 1 and 3 are known as is the gap between elements 1 and 2 in list 1.

[2]If the user were interested in capturing the name of the team as well he would probably not define the example strings to be Lists at all. Instead, they could be defined (in two steps) as Record{String teamName, List<*String*> playerNames}.



List 1
Team Name: Hawks; Name: Anderson Name: Blaylock
List 2
Team Name: Pistons; Name: Dumars Name: Hill Name: Hunter
List 3
Team Name: Bucks; Name: Allen Name: Brandon

Figure 9: Example lists.

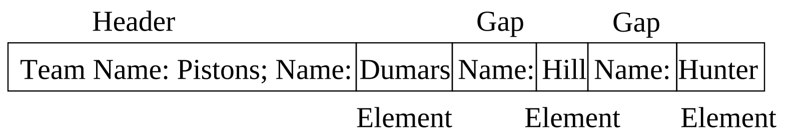


Figure 10: Presumed format of a list.

The task of the miner is to generalize from these examples to discover how to identify the headers and gaps in all of the lists. This will require two types of theory: an end theory for finding the end of the header (if present), which we will call the *header theory*, and a unit theory for finding the beginning and end of the elements of the list, which we will call the *element theory*.

Simplified psuedo-code for mining algorithm used in NoDoSE is shown in Figure 11. The code employs the three steps described in the preceding section: theory generation (lines 1-2), theory evaluation (lines 3-28), and theory application (lines 29-32). Each of the steps is discussed in detail below.

Theory generation For lists, we have two types of theories to generate, header and element theories. Let us first consider how to generate the set of header theories, T_H. We begin with the end theories from Table 1. Next, we instantiate the placeholders in the theories which means choosing markers, offsets, and numbers of lines. For example, consider instantiating the marker in the theory [**With Marker** "marker"]. To do so, we need to examine all of the known headers and find their longest common suffix. In the running example, the longest common suffix is the string "; Name: " and so the instantiated theory, [**With Marker** "; Name: "] is added to T_H. (In fact, the prototype actually adds two almost identical version of this theory, one in which the marker is case sensitive and one in which it is not, but this does not affect the algorithms in any fundamental way and hence will not be further mentioned.)

Often there will not be any consistent value with which to instantiate a theory. For instance, there may be no common suffix at all or the headers may not all have the same number of lines in them. In this case, the theory is not added to H_T.

Generating element theories is similar to generating header theories except that element theories are composed of both a start theory and an end theory — header theories do not require start theories since they always begin at offset 0. With the possible exception of the first and last elements in a list, an element can be viewed as shown in Figure 10. A pre-gap is the gap between the element and the preceding element and a post-gap the gap between the element and the following element. The theories for parsing list elements are based on trying to find a common suffix in the pre-gaps, a common prefix in the elements, a common suffix in the elements, or a common prefix in the post-gaps. In the example lists, the common prefix and suffix of the gaps are both " Name: "

and the elements do not have a common start or end marker. Thus the set of valid start theories, T_{start}, is {[**After Marker** " Name: "]} and the set of valid end theories, T_{end} is {[**Before Marker** " Name:"]}. The set of candidate element theories, T_E, is computed as $T_{start} \times T_{end}$, which in this case is just {[**After Marker** " Name: "],[**Before Marker** " Name:"]}.

Theory evaluation This step represents the majority of the code in Figure 11. Conceptually, the code chooses the best header theory and then uses that theory in trying to find the best unit theory. This is a bit of a short cut since the algorithm should really consider all possible pairs of header and element theories. Unfortunately, this is very expensive with even a moderate number of different theories. Hence we choose to separate the two tasks, realizing that in some cases we may miss the best pairing.

The best header theory is determined in Lines 3 through 10. The algorithm tries each theory, using it to predict the headers of all of the list instances. For those lists that have at least one element defined, the header is known and can thus be checked against the predicted header. If one of the theories correctly identifies all of headers it is assumed to be correct and saved for use in element parsing. If no theory is correct, we assume that headers do not have to be specially handled for the target list type.

In the example from Figure 9, it is critical that the headers are handled. By skipping past the header, the element theory <[**After Marker** " Name: "],[**Before Marker** " Name: "]> can be used to correctly identify every element. If the header is not skipped, the same element theory will predict that the first element of the list starts with the " Name: " marker that is part of "Team Name: " and thus mining will fail. If the headers were "Team: " instead of "Team Name: ", however, the element theory would work even if headers were not skipped. Thus many lists do not require a header to be found at all and therefore the mining algorithm continues even if no consistent header theory can be found.

The code from Lines 11 to 28 uses each element theory to predict the elements in all of the lists. For each list, the search for elements starts at the first character in the unit text unless a header is present in which case the search begins immediately after it (line 17). The start theory is used to find the predicted beginning of the next element (line 19), and if its successful, the end theory is used to find the predicted ending of the element (line 21). If a new element is found it is added to the predicted set. The search continues in the same unit text until no more elements are found. At this point the predicted elements are compared again the elements defined by the user. The function $findFalseNegatives(predictedSet, actualSet)$ counts the number of user defined elements that were not predicted by the theory, $|actualSet - predictedSet|$. The function $findFalsePositives(predictedSet, actualSet)$ counts the number of predicted elements that must be incorrect, which are the new predicted elements that start before the last user defined element ends (by assumption 3). For example, in Figure 9 if a theory predicted that an element other than "Allen" started prior to the space after "Allen" in list 3, it would have to be incorrect. For list 2 which has no user defined elements, however, no predicted element can be eliminated.

Theory application As long as at least one consistent element theory has been found, the elements that it predicted and that were not in the original document trees are added using the Instance Manager (lines 29-32). The new elements are of the same type as all of the other children

```
(1)  Create and instantiate the set of header theories, $T_H$.
(2)  Create and instantiate the set of element theories, $T_E$.
(3)  $t_{h_{best}}$ = null
(4)  for each header theory, $t_h$, in $T_H$
(5)      for each list $l$ in $L$
(6)          $offset = findHeaderEnd(t_h, l)$
(7)          if ($l$ has a header and $offset \neq l.headerEnd$) then
(8)              $t_h.errors{+}{+}$
(9)      if ($t.errors == 0$) then
(10)         $t_{h_{best}} = t$
(11) $t_{e_{best}}$ = null
(12  for each element theory $t_e$ in $T_E$
(13)     for each list $l$ in $L$
(14)         $offset = 0$
(15)         $predicted = \{\}$
(16)         if ($t_{h_{best}} \neq$ null) then
(17)             $offset = findHeaderEnd(t_{h_{best}}, l)$
(18)         while ($offset < l.length$)
(19)             $start = offset = findElementStart(t_e, l, offset)$
(20)             if ($start \neq -1$)
(21)                 $end = offset = findElementEnd(t_e, l, offset)$
(22)                 if ($end \neq -1$)
(23)                     $predicted = predicted \cup \{< l, start, end >\}$
(24)         $t_e.predicted = t_e.predicted \cup predicted$
(25)         $t_e.falsePos \mathrel{+}= findFalsePositives(predicted, l.elements)$
(26)         $t_e.falseNeg \mathrel{+}= findFalseNegatives(predicted, l.elements)$
(27)     if ($t_e.falsePos == 0 \wedge t_e.falseNeg == 0$) then
(28)         $t_{e_{best}} = t_e$
(29) if ($t_{e_{best}} \neq$ null) then
(30)     For each element $< l, start, end >$ in $t_{e_{best}}.predicted$
(31)         if $< start, end > \notin l.elements$
(32)             $l.elements = l.elements \cup \{< start, end >\}$
```

Figure 11: The (simplified) algorithm for mining and parsing list types.

of the target type, the author ids identify the text miner as the author, and a confidence factor can be assigned. In this version the confidence factor of the new elements is set to 0.5, but in the future we plan to compare each element against statistics gathered on all of the elements to try to identify questionable predictions. For instance, if all of the elements are 40 characters long except for one which is 80, it is likely that due to a an error in the parsing rule or a typo in the document, two consecutive elements have mistakenly been parsed as one.

4.1.2 Mining records

The mining algorithm for record fields depends on four assumptions:

1. Every field in a record has a unique name. This assumption is enforced by the Instance Manager.

2. If the fields of two different records of the same type have the same name, the two fields themselves will have the same type. Such fields are called *corresponding* fields. For example, if two record of the same type both have fields named *phoneNumber*, the two phoneNumber fields should have the same type. This assumption is also enforced by the Instance Manager.

3. All corresponding fields will have the same format. This assumption is not forced on the mining component but it seems reasonable given assumption 2 and simplifies the grammar induction.

4. The fields in a record instance are either completely identified by the user or not identified at all. Thus if k fields of a record instance have been identified by the user, they will be the only k fields in that instance. This is a very powerful assumption because it gives the miner a way to detect a parsing theory that generates false positives — a predicted field must appear in a record if the user has identified any fields at all in that record.

Note the assumptions that this component does *not* make: every field is present in every record instance, and the order of fields within a record is fixed. Thus we are able to parse a limited but useful class of semistructured documents.

Mining lists and records are similar except for one important difference. For lists, we assume that the format of every element is the same. This assumption allows the list mining algorithm to consider only two sets of theories, one to skip past the header and one to identify elements. In contrast, every field in a record type may have a different format and thus every field requires its own set of theories. Further, the order in which the algorithm tries to parse the fields is important.

For example, consider the text of a record that contains, among other things, the string "Name: Smith, John.". Suppose the user chooses to model a name as two fields, *LastName* and *FirstName*. The best theory for identifying a *LastName* field might be <[**After Marker** "Name:"], [**Before Marker** ","]>. If we know that a *FirstName* field always follows *LastName*, its rule would be <[**After Marker** ","],[**Before Marker** "."]>. We must be careful, though, to only try to apply the unit theory for a first name immediately after parsing a last name. Otherwise, an unrelated comma anywhere is the text of the record would lead to the first name being falsely parsed.

To avoid problems of this sort, the mining algorithm tries to find an order for the fields in a record type that is consistent across all of its instances. This is difficult for two reasons:

```
(1)  Pick a consistent order for the fields of the target type.
(2)  $numFieldsMined = 0$
(3)  for $i = 1$ TO $N_f$
(4)      Create and instantiate the set of field theories, $T_{F_i}$.
(5)      $t_{best_i} = \text{null}$
(6)  for $j = 1$ TO $N_r$
(7)      $lastOffset_j = 0$
(8)  for $i = 1$ TO $N_f$
         — Try to find field $i$ in every record —
(9)      for $j = 1$ TO $N_r$
(10)         for each unit theory $t$ in $T_{F_i}$
(11)             $start = findFieldStart(t, r_j, lastOffset_j)$
(12)             if $(start \neq -1)$ then
(13)                 $end = findFieldEnd(t, r_j, start)$
(14)                 if $(end \neq -1)$
(15)                     $t.predicted = t.predicted \cup \{< r_j, start, end >\}$
(16)                     $t.falsePos += findFalsePositives(\{< r_j, start, end >\}, r_j.fields)$
(17)             if $(start = -1 \vee end = -1)$ then
(18)                 $t.falseNeg += findFalseNegatives(i, r_j.fields)$
         — Find the first theory that perfectly predicted field $i$ using the theories —
(19)     for each theory $t$ in $T_{F_i}$
(20)         if $(t.falsePos == 0 \wedge t.falseNeg == 0)$ then
(21)             $t_{best_i} = t$
(22)             $numFieldsMined{+}{+}$
(23)             for $j = 1$ TO $N_r$
(24)                 $newOffset_j = findFieldEnd(t, r_j, findFieldStart(t, r_j, lastOffset_j))$
(25)                 if $(newOffset_j \neq -1)$ then
(26)                     $lastOffset_j = newOffset_j$
(27)             break
(28) if $(numFieldsMined = N_f)$ then
(29)     for $i = 1$ TO $N_f$
(30)         For each element $< r, start, end >$ in $t_{best_i}.predicted$
(31)             if $(< start, end > \notin r.elements)$
(32)                 $r.elements = r.elements \cup \{< start, end >\}$
```

Figure 12: The simplified algorithm for mining and parsing record types.

1. Not all fields are present in each record instance and no single instance is guaranteed to have every field in it. Thus we may have to look at more than one record instance to determine the field order and may not be able to determine a unique ordering even if we look at every record instance.

2. An important ordering may only exist between subsets of the fields in the record and the other fields may exhibit an inconsistent ordering.

The miner uses a simple algorithm that computes for each field, the set of all fields that have preceded it in at least one record and the set of all fields that have followed it in at least one record. The two sets can be used to find a totally consistent ordering if one exists although it is not guaranteed to handle the second complication from above. So far, this has not been a problem in the documents we have mined.

The psuedo-code for the record mining algorithm is shown in Figure 12. Lines 1 through 7 do theory creation and general initialization. Theory variables are instantiated by comparing the corresponding fields in all of the records, looking for a common pre-gap marker, start marker, end marker, or post-gap marker as was done with list elements.

The for loop starting at line 8 performs the meat of the algorithm: All of the record instances of the target type are parsed in parallel, one field at a time. Each theory for the current field number is tried to see if it can identify the field in each of the record instances (lines 11-15). If so, the algorithm calls $findFalsePositive$ to see if the field should really have been found. Using assumption 4, a predicted field is a false positive if it has not already been defined and the user has defined at least one field in the record instance. If no field was found for a record instance, the algorithm checks that none was defined by the user (lines 17-18).

After all of the theories have been tried on all of the record instances, the first consistent theory is chosen (lines 19-26). In addition, the current offsets into all of the records are updated to account for the newly parsed fields. The main loop then repeats, starting from the new offsets and looking for the next field.

After all of the fields have been parsed, the algorithm checks that it has found a consistent theory for every field. If it has, all of the fields predicted by the consistent theories are added to their records.

4.2 HTML Parser

The HTML parser available as part of NoDoSE parses documents or subdocuments based completely on structural information using recursive descent. Unlike the plain text miner, the HTML parser does not store any internal information about a type since it parses based on the static grammar rules of HTML. The tags that are understood by the parser are listed in Table 2 with a brief description (due to space constraints) of how they are represented in the document tree. Other tags are just considered part of the text of the document. The following discussion is necessarily brief due to space constraints.

In practice, the HTML parser generates more structure than the user is interested in (i.e., information from meta tags). After parsing, he should delete the nodes that are not of interest and rename types and labels to be semantically meaningful. The changes are not recorded by the HTML parser, though, so the next instance of the same page type will require all the changes to

Tags	*Representation*
head,title,meta	Head is represented as a Record with two fields: title, which is a String extracted from the title tag, and meta, which is a List of strings extracted from the meta tags.
body	Body is represented as a SemiList. Typically its children will be first level headings.
h1,h2,h3, h4,h5,h6	Each heading is represented as a SemiList. Typically its children are paragraphs, represented as Strings, and sub-headings.
ul,ol,dir menu,li	All of the list formats are represented by a SemiList of list items. Lists can be arbitrarily nested.
table,tr,td	A table is represented as a SemiList of rows. Each row is a SemiList of the data in the cells.
anything else	Represented in the tree as a string.

Table 2: Translation of HTML tags into document structure.

be made again. To avoid this problem, the plain text miner can be run on the parsed document produced by the HTML miner. It will try to infer the format of the file just as if the structure had been entered using the GUI. If it is successful, future HTML files of the same type can be parsed automatically.

The current version of the plain text parser only works with structured types, however, so additional modification of the document tree produced by the HTML parser is necessary. Thus the typical use of the HTML parser follows these steps:

1. Use the GUI and plain text miner to decompose the document until the HTML portion is reached. If the entire document is HTML, this step can be skipped.

2. Run the HTML parser on the HTML portion. This creates a sub-tree below the current node that represents the HTML portion of the document parsed.

3. Use the editing capabilities of the GUI to convert the sub-tree to a form suitable for mining. For example, the user must delete the nodes that are not of interest and rename types and labels to be semantically meaningful. Also, the nodes created by the HTML parser will all be semistructured (except for the atomic types). The types of these nodes must be changed to a structured type before the miner described in the previous section can work on them.

4. If there are other documents of the same type, load them and use the plain text miner to mine them.

5. Perform any needed edit steps and repeat step 4 until the plain text miner get the correct results for all of the files.

The approach of running the miner on the tree produced by the parser is an attractive one since it lets the text miner benefit from the knowledge of the document syntax without requiring that the parser and miner communicate directly. The same approach can be used on other files types with known syntax such as latex files or mail files. For example, consider a mail file consisting of responses to a survey. Each message will have all of the normal header information plus a highly formatted message body with the survey answers. By parsing the file using the mail syntax and then mining the message bodies to structure the survey responses, a user can quickly export the survey data along with header data (send, time it was sent) to a DBMS.

5 Implementation

A prototype that implements a subset of the components described in Section 3 has been implemented in approximately 5000 lines of Java code. Two of the components, the Transaction Manager and the Log Manager, have not been implemented at all although the interfaces have been designed; thus NoDoSE version 1.0 does not support undo. Also, the current version only contains one Reporting component that writes the extracted data in a generic format similar to OEM. It also outputs an ODL schema for the data if it is not semistructured.

We have run NoDoSE on many different files, including simulator output, mail files, c source code, OCRed documents, and many web pages. The results are difficult to quantify, although overall we have been pleased at the wide range of documents NoDoSE can extract data from. Part of this success, however, is the result of learning how to work around the quirks of the system. For example, the miner is currently very sensitive to where the user chooses the boundaries between list elements and record fields (i.e. whether to include the final carriage return in the selected text).

Also, the dependence of many of the theories on constant string markers causes problems. For instance, the records of type OneParam from the simulation output example (Figure 2) look like " 5 misDL - (avg) 9.295315E-01 - (std) 2.559869E-01 - (num) 2455" where the first value, 5 in this case, is the simulator node number that the measured variable is in. This value is redundant and is thus not part of the record type we defined for OneParam. The consistent pre-gap marker for the variable name (misDL in this case) is two spaces since OneParam records from other nodes start with their own node number, i.e., " 4 ". Unfortunately, identifying the beginning of the variable name by two spaces will fail since the node number, which precedes the variable name in the text, is also preceded by two spaces, and will thus be falsely identified as the variable name. This problem can be avoided by including the node number in the record even though it is redundant. Doing so "eats" the node number so that two spaces serves as an adequate pre-gap marker for the variable name field. To obviate the need for such workarounds in the future we're more developing more flexible markers based on regular expressions.

The performance of the system is fine for small files — the mining wait is never more than a second or two. We have not been able to deal with large files, however, due to the state of Java. One problem is with the TextArea component which is used to display portions of the file on the right side of the program window. The component in the toolkit uses the windows peer which does not support files over 32kb and we have not been able to find a pure Java component with adequate performance for files over 100kb. Luckily, many interesting document types, especially web pages, are well within this limit so this restriction has not significantly hampered our research, although it has made measuring scalability impossible. Given the commercial push for Java, it's reasonable to believe that such problems will be corrected in the near future.

6 Related Work

NoDoSE makes two major contributions to the data extraction problem: its open architecture for structural mining and the plain text mining component that has been implemented in version 1.0 of the system. To our knowledge, the former has not been proposed by any other researchers and hence we do not discuss it further in this section. Instead, we concentrate on the approaches others have taken to the latter problem: mining structure and extracting data from documents.

The three efforts that are most closely related to our own are [AK97a,AK97b], [HGMC$^+$97], and [KWD97]. The system built by Ashish and Knoblock [AK97a] is closest to NoDoSE in its approach: to infer the structure of a document by combining automatic analysis with user input. Their system is designed for web pages only: it uses font size information, HTML tags, and indentation to guess a page's structure. A user can then correct the guesses by instructing the system to ignore certain keywords and by identifying new keywords that the system missed. The advantage of this system is that certain types of pages can be parsed with very little user input since the system leverages its knowledge about HTML syntax and about how characteristics like font size are used to indicate nesting. The major disadvantages of the system is that because it depends on HTML tags, it is not useful for any other type of document. Also, it deals with only single instances of documents so it is unclear that it can be used in cases where no single instance of a document type has all of the features of the type.

Kushmerick, Weld, and Doorenbos describe a system [KWD97] that automatically extracts data from web pages although it will also work with plain text files. The extracted data must be representable as a set of tuples; no deep structure can be inferred. The advantage of the system is that no user interaction is required — the system infers the grammar of a document through a machine learning algorithm applied to many instances of the document type. The algorithm must be provided with domain knowledge, however, in the form of oracles that can identify interesting types of fields within a document. Further, if the algorithm fails, there is no information the user can provide to help it. The authors report a success rate of 48% on Internet information resources which is impressive for a fully automatic algorithm but not adequate for most applications. Still, their work contains interesting ideas for automatic parsing and their notion of corroborating recognizers is similar to way we evaluate theories against user input.

The final system we discuss, that of Hammer et. al. [HGMC$^+$97], is unlike the others in that it is fully manual — the user must code a wrapper for their document type using a toolkit. The toolkit provides many constructs, especially for HTML processing, that make it easier than writing a parser directly in Lex and Yacc [Joh75]. It also provides the most control over the output format of the extracted data as well as the best support for semi-structured data. The obvious disadvantage is that the user must be able to analyze their documents and then code their wrapper which limits the usefulness of this approach as a rapid data integration tool. We are considering it, however, as one of the output formats of NoDoSE to give users more control over the output format of their data.

For comparison, the salient features of the three related projects described above, as well of those of NoDoSE, are summarized in Table 3. In comparison to the other systems, NoDoSE has two primary advantages:

1. It is the only system that can infer the structure of text files and has support for HTML

System	*Grammar Generation*	*Nested structure*	*Semi-structured data*	*Text documents*	*Support for HTML*	*Open Architecture*
Ashish & Knoblock	Semi-automatic	Yes	Somewhat	No	Yes	No
Hammer et. al.	Manual	Yes	Yes	Yes	Yes	N/A
Kushmerick & Weld & Doorenbos	Automatic	No	No	Yes	No	No
NoDoSE	Semi-automatic	Yes	Somewhat	Yes	Yes	Yes

Table 3: Comparison of different data extraction tools.

documents.

2. It is the only system that can serve as a test bed for structure extraction experiments since the mining components are well separated from the rest of the system.

We note that except for the system of Hammer et. al. (which does not have a mining component), there is no reason that the other mining algorithms could not be integrated as mining components in NoDoSE. This would yield improved handling of HTML documents or with portions of documents that contain HTML while retaining the plain text capabilities.

Finally, we mention two additional studies that are related to this work. First, in [DEW97] the authors built a system, ShopBot, for the automatic extraction of product and pricing information from on-line shopping web sites. The system performs reasonably well since it is able to leverage its domain knowledge about shopping but is not applicable to other domains and it was not considered in the above comparison. Second, in [AJ97] the authors describe a fully manual GUI-based tool for converting structured files from one format to another. Unfortunately, not enough information is provided to compare it to the studies described above.

7 Conclusions

Given the amount of interesting data that is in HTML pages or text files rather than in database systems, users have a strong need for a tool to extract data from such sources. This paper described a tool, NoDoSE, designed explicitly for these needs. NoDoSE serves two purposes. First, it provides a general architecture for the exploration of the data extraction problem, allowing other researchers to plug in their own mining algorithms, user interfaces, or report generators, without having to build the entire framework themselves. Second, it contains a component that is capable of inferring the structure of a useful class of text files, allowing data to be quickly extracted without coding.

The results of using NoDoSE on the type of documents we were originally targeting, simulation output and web pages, are promising. We have also noticed that many documents with complex parsing rules, such as files of c code, that should be beyond NoDoSE's reach are not due to stylistic conventions like indentation and standard comment blocks. On the other hand, we have occasionally been surprised to find very simple and very regular looking documents that NoDoSE cannot handle.

Usually, the failure is due to the dependence of many of the parsing theories on constant markers that delimit list elements or record fields. Thus we are currently developing an alternate approach based on regular expressions. We are also finishing the documentation of the component interfaces and will release NoDoSE over the web soon thereafter.

References

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of ICDT (*invited talk*)*, 1997.

[AJ97] A. Aiken and M. Jacoby. Data format conversion using pointer tree automata. Technical report, University of California, Berkeley, 1997.

[AK97a] N. Ashish and C.A. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of cooperative information systems*, 1997.

[AK97b] N. Ashish and C.A. Knoblock. Wrapper generation for semi-structured internet sources. In *Workshop on management of semistructured data*, 1997.

[CGMH+97] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: integration of heterogeneous information sources. In *Proceedings of the meeting of the processing society of japan*, 1997.

[DEW97] R. Doorenbos, O. Etzioni, and D.S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the first international conference on autonomous agents*, 1997.

[GEH+94] Gamma, Erich, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of object-oriented software architecture.* Addison-Wesley, 1994.

[Gol90] A. Goldberg. Information models, views, and controllers. *Dr. Dobb's Journal*, July 1990.

[HGMC+97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *Workshop on management of semistructured data*, 1997.

[Joh75] S.C. Johnson. Yacc — yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[KGP88] Krasner, Glenn, and S. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-oriented programming*, August/September 1988.

[KWD97] N. Kushmerick, D.S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of IJCAI*, 1997.

[Liv90] M. Livny. **DeNet** user's guide. Technical report, University of Wisconsin-Madison, 1990.