CTS: An Interactive Technique for Manipulating Structured Text

A. Michael Burbidge
Computer Science Department
Brigham Young University
Provo, Utah 84602
mike@cs.byu.edu
(801) 378-4009

Dan R. Olsen
Computer Science Department
Brigham Young University
Provo, Utah 84602
olsen@bunsen.byu.edu
(801) 378-2225

Abstract

This paper describes Complex Textual Strings (CTS), a technique for manipulating structured text strings and an underlying application data structure by creating a two-way mapping between the text and the data structure. An editable specification, called unification grammars describes the structure and data mapping characteristics of a particular instance of CTS. We also describe an interpreter for parsing a complex textual string to produce a data structure, and pretty-printing a data structure to produce a complex textual string.

Keywords: Unification Grammar, Text Editing, Backtracking, Lisp, Structured Text, Athena Widget, Syntax Directed.

Introduction

A recently proposed model for UIMS, referred to as the Browse/Edit model [Olsen 88], suggests viewing user interface software as a mechanism for manipulating data or information from a particular application domain. To be effective, such a UIMS must provide a wide variety of interactive techniques for manipulating various types of application data. An overlooked technique, that can be used to manipulate certain kinds of complex data structures is a structured text string. That is, any text string that has a nontrivial syntactic and/or semantic structure. In this paper, we present a technique for specifying and manipulating complex textual strings (CTS), and their corresponding application data structures, in the context of a Browse/Edit type UIMS.

A Browse/Edit UIMS consists of three components: 1) The presentation, which is a set of techniques for interactively editing and viewing application data. This includes some method for specifying which techniques comprise a specific presentation. 2) A data structure shared by both the UIMS and the application. 3) A two-way mapping that describes how the presentation and the application data structure are related. i.e. How does the presentation change, when the application modifies the shared data structure, and vice versa. It is important that this description is editable, since a user interface is constructed by composing a set of these descriptions, that specify how generic interaction techniques are used to manipulate the shared data structure. A simple view of the Browse/Edit model is pictured in figure 1.

Cousin [Hayes 84] and ADG [DeSoi 89] are examples of systems based on the Browse/Edit UIMS model. In Cousin, the presentation is a set of named slots, a slot is a method of communication between the user interface and the application. The application data structure, consists of a set of Lisp primitives, associated with the slots. The primitives associated with a specific slot are called when the slot's value is modified. The presentation and presentation-to-application data mapping are described in a layout. A layout specifies all the slots and their attributes for a given presentation. One of the slot attributes is a set of lisp primitives.

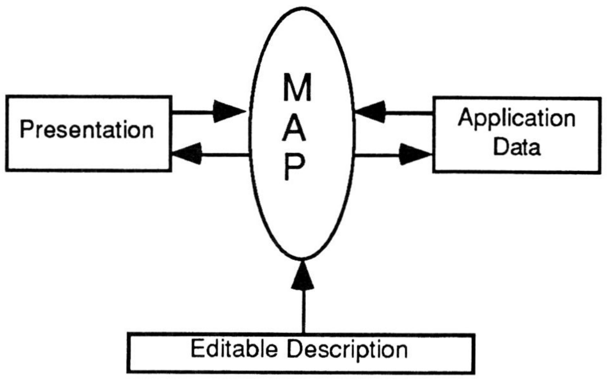

Figure 1 - Simple Browse/Edit Model

In ADG, the presentation is a set of instantiated behavior abstractions. A behavior abstraction is an interaction concept such as a slider. A slider can have many graphical presentations, such as a scroll-bar, or volume control. The behavior abstractions combined with a graphical presentation is an instantiated behavior. The application data structure, is composed from Lisp primitives and/or lists. A data flow diagram is used to describe the two-way relation between an instantiated behavior abstraction and the application data structure.

Complex Text Strings

Both Cousin and ADG handle text by treating each instance as an indivisible unit. Any slot in Cousin can be used to edit a free form text string. ADG provides a behavior abstraction called a Text Field, which can be combined with a graphical presentation to create an instantiated behavior abstraction that can be used to edit free form text strings. Most of the widget sets built with the X11 toolkit (Xt) contain a Text Widget, which can be used to create and edit simple multiline text string objects. Though these systems provide convenient facilities for editing text strings, none of them provide a method for dealing with the structure present in many kinds of text. For example, consider the textual representation of a phone number. Treating the text as an indivisible unit, makes it impossible to recognize meaningful substrings, such as the area code, and extension. It is also difficult to deal with multiple syntactic forms of the same string, in a meaningful way.

Rather than deal with text strings as indivisible units, we have developed the concept of a complex textual string (CTS). A CTS is any text string that has meaningful syntactic and/or semantic structure that can be used for manipulating a corresponding application data structure. Viewing structured text in this manner, we can then use the Browse/Edit model and construct a two-way mapping, between the meaningful substrings in a CTS and an appropriate application data structure.

To demonstrate this, consider creating a two-way mapping between a textual representation of a phone number, and a data structure built using Lisp type lists. The significant substrings in a phone number are: area code, prefix, and suffix. A two-way mapping can be constructed using a list of three elements, as shown in the following examples:

```
(801) 378-4009     <==>     (801 (378 4009))
378-4009           <==>     (() (378 4009))
```

More complex mappings can easily be constructed. For example, text strings that represent proper names must be mapped to a variable length data structure. Using a CTS, a two-way mapping could be represented by a two element list, the first consisting of an arbitrarily long list of symbols, one for each given name, and the second a symbol representing the sir name. Some examples would be:

```
Hillman, Lee          <==>     ((Lee) Hillman)
Hillman, Gary Lee  <==>     ((Gary Lee) Hillman)
Sant, Bryan Chad Romney <==>
            (Bryan Chad Romney) Sant)
```

Text strings representing recursive kinds of things can also be handled. For example, a textual representation of a spread sheet formula could be mapped recursively to a postfix representation, as shown below:

```
C1 = A1 + 3              <==>  (assign C1 (add A1 3))
C1 = (A1 - B1) * 0.5  <==>
          (assign C1 (mult (sub A1 B1) 0.5))
```

Unification Grammars

A Unification Grammar (UG) is a method for specifying the presentation to application data mapping characteristics of a CTS. The presentation component of a CTS, is the syntactic structure of the text. This is specified using a context-free grammar. Since the presentation to application data mapping is dependent on the structure of the text, we augment the context-free grammar with patterns constructed from Lisp type lists, to specify the two-way map.

This is different than other syntax directed editing approaches in two ways: First, systems such as the Cornell Program Synthesizer [Reps 84] map text into a fixed data structure, that is actually hardwired into the structure of the program. And second, the text to data mapping function of the Synthesizer is one-way. Using unification grammars we get a two-way mapping.

A pattern is a list that contains constants, variables, or subpatterns. Variables are symbols that begin with a question mark and match any arbitrary list. A specific variable may occur more than once in a pattern, but it must represent the same list in all cases. An example of a simple pattern is:

```
((?first ?second) ?last)
```

Patterns are useful for qualifying a set of lists that have a certain characteristic. For example the above pattern could be used to match lists that represent the names of people that have two given names. The following pattern would match any three element lists whose first and last elements were equivalent:

```
(?A ?B ?A)
```

All lists representing phone numbers that have no area code would match the pattern:

```
(() (?prefix ?suffix))
```

or phone numbers of people living in Utah, will match the pattern:

```
(801 (?prefix ?suffix))
```

Using a context-free grammar we can specify the syntax of a phone number with the following single production grammar:

```
Phone ::= "(" Digits ")"  Digits "-" Digits ;
```

By augmenting this grammar with patterns we can specify a the two-way mapping between segments of the structured text and the application data structure:

```
<Phone:(?area-code (?prefix ?suffix))> ::=
          "(" Digits:?area-code ")"
          Digits:?prefix "-" Digits:?suffix;
```

It is important to note that the scope of a variable in a unification grammar, is the variable's surrounding production. This means, in the previous example, that ?suffix in the pattern associated with <Phone>, refers to the same variable as ?suffix associated with the last instance of Digits.

Given the grammar for phone number we can parse the text string (801) 378-4009 to produce the following bindings:

```
?area-code = 801, ?prefix = 378, ?suffix = 4009
```

Given these bindings, the list associated with <Phone> becomes (801 (378 4009)). Now, we can also go in the reverse direction. Given the grammar and the list (801 (378 4009)), by unifying the list and the pattern (?area-code (?prefix ?suffix)) we produce the same bindings as when parsing, and given the right-hand side of the production can now produce a syntactically correct phone number.

The process of taking a unification grammar and a text string and producing the corresponding list, can be called parsing, although it is more powerful than a simple parse, since it is really driven by the data and the grammar, rather than just the grammar. The reverse operation, that of taking a list and a unification grammar and producing the text string, we call pretty-printing.

A parse would proceed in the following manner: Given a production, begin examining, in order, the syntactic symbols on the right-hand side of the production. If a symbol is a constant such as "(", we make sure that the next token in the text string matches. Otherwise, we simply bind the token in the text string to the variable associated with the current symbol. A parse of the string (801) 378-4009 would consist of the following steps:

1) Match the constant "(", from the production against the first token, "(" in the string.
2) Get the next syntactic symbol, Digits:?area-code. It is not a constant, so bind 801 to the variable ?area-code.
3) Match ")" from production against ")" from text string.
4) Bind 378 to ?prefix
5) Match "-" from production against "-" from text string.
6) Bind 4009 to ?suffix.

We end up with the bindings ?area-code=801, ?prefix=378, and ?suffix=4009. And since the scope of a variable is its surrounding production, the resulting application data is the list (801 (378 4009)).

The pretty-print would proceed in a similar fashion, except that instead of examining the syntactic symbols, we begin by unifying the given list and the pattern associated with the production. This should give a binding to all variables in the production. We then examine the syntactic symbols, in order. If a symbol is a constant we output the constant to the text string, otherwise we output the value of the pattern associated with the syntactic symbol.

By adding multiple productions we can allow alternate syntax for one type text string. For example, the following production does not require an area code, but maps the local area code into the application list:

```
<Phone:(801 (?prefix ?suffix)) ::=
        Digits:?prefix "-" Digits:?suffix;
```

Using this production, the text 378-4009 maps to (801 (378 4009)) and vice versa, and combining this with the previous production, we have a grammar representing a CTS that is flexible enough to manipulate phone numbers with or without area codes.

So far, our examples have been very simple. When specifying a unification grammar for any reasonably complex string it is useful to break the string into sub-strings and specify productions for each of the sub-strings. Then using productions for sub-strings we can specify a production for complete string. For example, a unification grammar equivalent to the two productions given above would be:

```
<Phone:(?area-code ?number)> ::=
        <AreaCode:?area-code> <Number:?number>;
<Phone:(801 ?number)> ::= <Number:?number>;
<AreaCode:?area-code> ::= "(" Digits:?area-code ")";
<Number:(?prefix ?suffix)> ::=
        Digits:?prefix "-:" Digits:?suffix;
```

Some types of structured strings have a recursive nature. Any string, such as the given name example used earlier, that contains a variable length list of sub-strings can be described recursively. Following is a recursive unification grammar describing a variable number of given names:

```
<Name:(?gnames | ?last)> ::=
        Name:?last "," <GName:?gnames>;
<GName:(?new_name | ?name_list)> ::=
        Name:?new_name <GName:?name_list>;
<GName:(?name)> ::= Name:?name;
```

In this example we used the concatenation operator |, as in Prolog [Sterling 86], for constructing a list. The element preceding the | is the CAR of the list, and the element following the | is the CDR of the list.

By combining recursion and multiple nonterminals a fairly complex textual string and mapping can be described. The following unification grammar describes an arithmetic expression, such as could be used in a spread sheet formula, that is mapped to a Lisp list postfix notation:

```
1) <Exp:(add ?op1 ?op2)> ::=
        <Operand:?op1> "+" <Exp:?op2>;
2) <Exp:(sub ?op1 ?op2)> ::=
        <Operand:?op1> "-" <Exp:?op2>;
3) <Exp:?op> ::= <Operand:?op>;
4) <Operand:?op> ::= Integer:?op;
5) <Operand:?op> ::= Real:?op;
6) <Operand:?op> ::= CellName:?op;
```

Given the above grammar and some example expressions, the following two-way mappings are established:

```
C1 + 5          <==>      (add C1 5)
A1 + 5 - C1     <==>      (add A1 (sub 5 C1))
```

Parsing/pretty-printing a CTS specified by a unification grammar containing multiple nonterminals proceeds in much the same manner as described earlier, with the exception of what happens when we encounter a nonterminal while examining the right-hand side syntactic symbols. In the above example, we would be given Exp as a starting symbol.When encountering a nonterminal on the right-hand side of a production, we first save the position in the current production, and then choose the appropriate production for the nonterminal just encountered. We then proceed with the new nonterminal just as in the single nonterminal case, except that when we're finished, we return to finish processing the remaining symbols in the original production.

Consider the steps for parsing "C1 + A1 - 5":

1) Chose production 1, and begin processing its syntactic symbols.
2) The first symbol is the nonterminal symbol Operand, since it is a nonterminal we save our position and begin processing a production for Operand.
3) Choose production 6, for Operand.
4) Begin processing its symbols. There is only one, the terminal symbol CellName, it is not a constant ,so set ?op = C1, the next token in the text string.
5) Return to the saved position in the original production, Exp. Note the result of processing the Operand nonterminal is that ?op1 = C1.
6) The next syntactic symbol is the constant "+", we check it for a match against the next token.
7) The next symbol is a nonterminal for Exp, it is processed

in a manner similar to the Operand nonterminal, except it will recurs through two nonterminals instead of one. When we return from processing Exp, ?op2=(sub A1 - 5), and we are finished. The resulting data structure is (add C1 (sub A1 5)).

Algorithms

Both the CTS parser and pretty-printer are implemented using the standard backtracking and unification algorithm found in backward-chaining production systems. One of the unique concepts of the CTS algorithm is its technique for mapping the terminal tokens of a text string onto the productions in a unification grammar. This is also the only difference between the parser and pretty-printer algorithms. When the parser encounters a terminal token, it is mapped to the appropriate location in the shared data structure. When the pretty-printer encounters a terminal token, the value in the shared data is output to the text string.

There are many efficient deterministic algorithms for parsing text and mapping it onto an intermediate data structure However, most of them require some form of preprocessing that excludes them from being used interactively. Using a backtracking and unification algorithm we get an editable specification that is easily modifiable without a time consuming preprocessing or compilation step, which is one of the requirements for interactive techniques in a Browse/Edit UIMS.

We have encountered a few problems with the algorithmic approach we have taken. First, certain kinds of complex recursive strings do not perform at interactive speeds, do to heavy backtracking. We have not observed the problem when dealing with just complexity, or just recursion. A prime example, is a string that represents a list of program statements. A program statement can be quite complex, and at the same time be composed of recursive type strings, such as arithmetic expressions. We feel that performance could be improved a great deal by implementing a Cut type operator, as in Prolog. This would limit the amount of backtracking for certain productions.

Another problem, is that it is possible to specify a unification grammar that has a unique kind of ambiguity in the way that the parser and the pretty-printer interact. A simple example best describes the ambiguity. Consider the following abbreviated unification grammar:

```
<Exp:(add ?op1 ?op2)> ::=
          <Operand:?op1> "+" <Exp:?op2>;
<Exp:?op> ::= <Operand:?op>;
<Operand:?op> ::= Integer:?op;
<Operand:?op> ::= "(" <Exp:?op> ")";
```

Given the string "a + b" and the above grammar, parsing produces the list (add a b), but parsing the string "(a + b)" produces the same list. The problem comes in trying to pretty-print the list (add a b). The pretty-printer encounters an ambiguity in trying to decide which of the previously given strings should be produced. This is manifested by infinite recursion. (hang) Fortunately, there is a simple solution. Replace the last production by:

```
<Operand:(precedence ?op)> ::= "(" <Exp:?op> ")";
```

Now parsing the string, "(a + b)" produces the list (precedence (add a b)), thus removing the ambiguity encountered by the pretty-printer.

Using Structured Text Strings

So far we have described, rather abstractly, how to specify the structure of a CTS and the two-way mapping between it and the application data structure. We now describe how we have implemented these concepts using the Athena text widget [Swick 88] as a means of interactively manipulating the actual text. Although, the Athena widget set is not a Browse/Edit UIMS, building a miniature Browse/Edit system that deals strictly with CTS, using the Athena text widget has provided an environment for us to easily experiment with these ideas. We hope to integrate these ideas into other research we are currently doing in our laboratory.

Currently, the CTS implementation consists of three components : 1) A parser which accepts a unification grammar and a text string, and returns a Lisp type list. 2) A pretty-printer which accepts a unification grammar and a list, and returns a text string. 3) A set of routines for manipulating the Lisp lists returned by the parser. This allows an application to inspect and modify the lists. Although, dealing with lists is inconvenient in C, it is not a limitation. Any arbitrary data structure can be represented with lists. We have plans to replace the Lisp lists with an object-oriented data model and application framework.

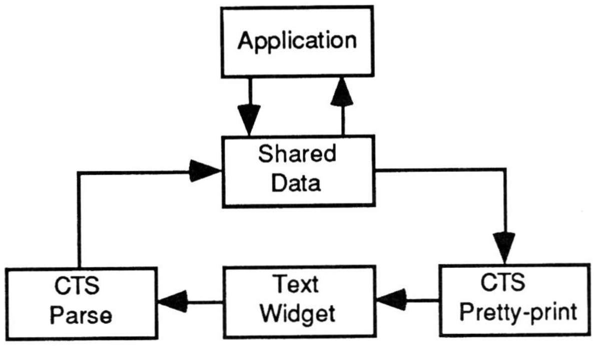

Figure 2 - Prototype CTS Architecture

Figure 2 depicts the overall architecture of the prototype implementation of CTS. We have added a callback resource to the Text Widget, that is activated on any specifiable key down event. The callback routine retrieves the text from the text widget and calls CTS_Parser, passing it a unification grammar and the text string. The CTS_Parser maps the text string to the appropriately structured list. Finally, the callback routine calls CTS_Pretty_Printer passing it the list and the unification grammar and then using the text widget redisplays the returned text string. In the case of multiline text strings formatting information such as indentation can be specified in a unification grammar. For example, the following production demonstrates the description of indenting for a generic If statement:

```
<IfStmt:(ifstmt ?cond ?block1 ?block2)> ::=
  "if" <CondExp:?cond> "then" Indent:"+2" "\n"
  <StmtBlock:?block1> Indent:"-2" "\n"
  "else" Indent:"+2" "\n"
  <StmtBlock:?block2> Indent:"-2" "\n"
  "endif";
```

It is also possible for the application to modify the list data structure, and in turn activate the CTS_Pretty_Printer and redisplay of the resulting text using the text widget.

Conclusions

Many kinds of textual interaction contain inherently meaningful structure. In the research described in this paper, we have tried to develop an interactive technique for manipulating structured textual strings and their underlying application data structure. Using ideas from the Browse/Edit model we have suggested that a two-way mapping can be constructed between structured text and application data. We have proposed unification grammars as a method to specify both the structure and data mapping characteristics of complex text strings. We have described an interpretive implementation based on unification and backtracking for parsing and pretty-printing these structured text strings and their corresponding lists. By using an interpretive approach, the specification of complex text strings can be edited at any point in the development/production cycle; no preprocess or compilation step is required. We have described an implementation running in our lab, that performs at acceptable interactive speeds for simple to moderately complex strings.

References

[DeSoi 89] DeSoi, J.F., Lively, W.M., and Sheppard, S.V., "Graphical Specification of User Interfaces with Behavior Abstraction." *CHI'89 Proceedings* (May 1989).

[Hayes 84] Hayes, P., "Executable Interface Definitions Using Form-Based Interface Abstractions." *In Advances in Computer-Human Interaction*, H.R. Hartson, E., Ablex, New Jersey, 1984.

[Olsen 88] Olsen, D.R., "A Browse/Edit Model for User Interface Management." *UIST'88 Proceedings* (November 1988).

[Reps 84] Reps, T., and Teitelbaum, T., "The Synthesizer Generator." *In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pa., Apr. 1984).

[Sterling 86] Sterling, L., In The Art of Prolog: Advanced Programming Techniques, E. Shapiro, Cambridge, Mass., MIT Press, 1986.

[Swick 88] Swick, R.R., and Weissman, T., In X Toolkit Athena Widgets -- C Language Interface, Digital Equipment Corporation and MIT Project Athena, 1988.