Adaptive Forms: An Interaction Technique for Entering Structured Data

Martin R. Frank, Pedro Szekely
University of Southern California - Information Sciences Institute
4676 Admiralty Way, Marina del Rey, California 90292
Martin.R.Frank@acm.org szekely@isi.edu

ABSTRACT

Many software applications solicit input from the user via a "forms" paradigm that emulates their paper equivalent. It exploits the users' familiarity with these and is well suited for the input of simple attribute-value data (name, phone number, ...). The paper-forms paradigm starts breaking down when there is user input that may or may not be applicable depending on previous user input. In paper-based forms, this manifests itself by sections marked "fill out only if you entered yes in question 8a above", and simple electronic forms suffer from the same problem - much space is taken up for input fields that are not applicable.

One possible approach to making only relevant sections appear is to hand-write program fragments to hide and show them. As an alternative, we have developed a form specification language based on a context-free grammar that encodes data dependencies of the input, together with an accompanying run-time interpreter that uses novel layout techniques for collapsing already-entered input fields, for "blending" input fields possibly yet to come, and for showing only the applicable sections of the form.

Keywords

Data entry, automatic layout, parsing, user interface software, human-computer interaction

INTRODUCTION

Adaptive Forms is a tool for producing context-sensitive form-based interfaces. The system initially displays an overview of the main sections of a form, and an initial set of fields for the user to fill in. Depending on the values that the user enters, Adaptive Forms progressively adds new fields to the form. For example, a form for entering household information would show the user fields for entering the spouse's name only if the user had entered "married" in the "marital status" field.

The main design goal for Adaptive Forms is in entering structured information rapidly and without errors. One of our target applications was the specification of air campaign objectives, which are structured objects consisting of a verb (e.g., deny, gain), an aspect (e.g., what to deny or gain), an actor (e.g., country, a branch of armed forces), a location (e.g., a country or a region) and a time period. Each of the parts is itself a structured object whose sub-structure and possible values depend on the values specified for the other parts. For example, the aspects that can be gained are different from the aspects that can be denied, so the interface needs to compute the menus for the "aspect" field dynamically based on the fillers of other fields. Similar requirements arise in virtually any other application domain.

This paper is a revision of [3] that incorporates progress we have since made (towards the end of the article).

EXAMPLE

The use of an Adaptive Form is best explained with an example. We have written a grammar for operational objectives of a fictional Southern California Emergency Response Center. Figure 1 shows the initial screen. The field labelled "evacuate/ensure" is the currently-active field, highlighted with a thick border (and with a red-brick background color that is hard to see in this black-and-white screenshot). The remaining fields are computed by "looking ahead" in the input grammar to the possible completions of the current sentence. Required fields are indicated by a solid border while optional fields have a dotted border. The captions beneath the fields give an indication of what goes into them.

The first row tells us that the current field is required and may contain either evacuate or ensure, that the next field is also required but that its possible contents cannot be determined until we have filled in the current field, and that the third field is optional. The second row is required, and it always starts with using the. The third and fourth rows are entirely optional.

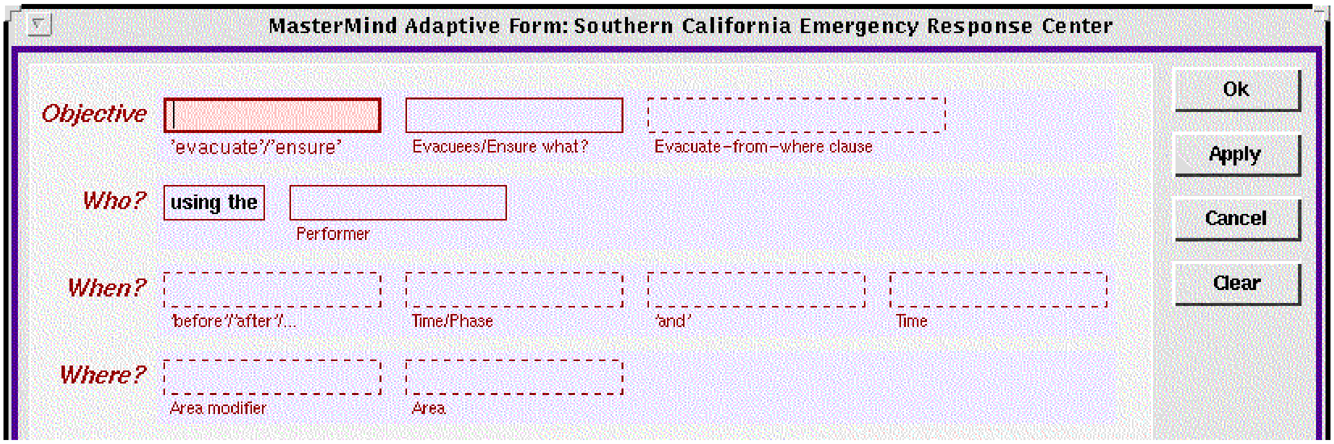

Figure 1. Emergency Response Center Example, Initial Screen

Figure 2 shows the interface after the user has typed ev<TAB> (the character 'e', the character 'v', and the TAB character). The Adaptive Form auto-completes all input so that only disambiguating input has to be entered, similar to the auto-completion in the Intuit Quicken personal finance program and in the Emacs editor [6]. In response, the cursor advances to the next field and the interface shows one more nesting level ahead (the Evacuate-from-where clause field from Figure 1 has expanded into two fields in Figure 2 - 'from' and Evacuate from where?). The lower pane shows the possible choices for the current field.

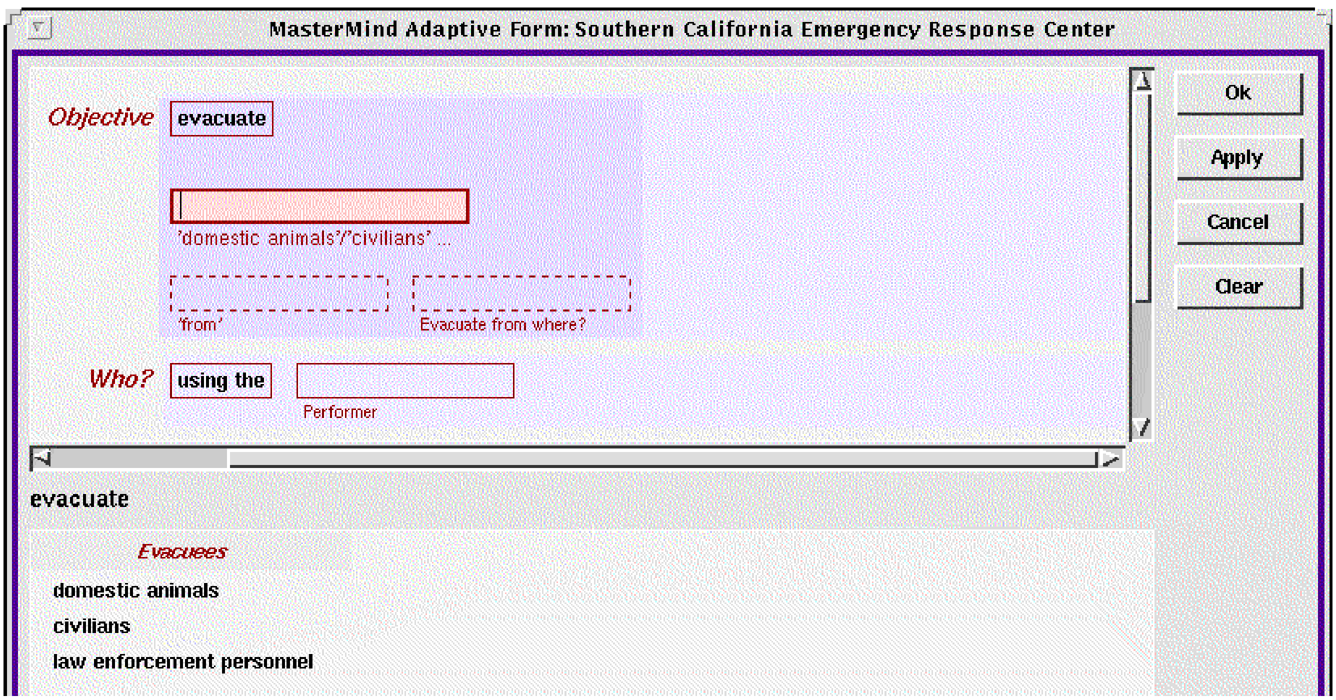

Figure 2. Emergency Response Center Example, Screen Two

Figure 3 shows the state after the user has typed d<TAB>M. Malibu and Marina del Rey are highlighted in yellow as they are the possible choices for the character M. Per the input grammar, an evacuation is possible from either a city or a county - the interface uses a multi-column format with headers to convey this to the user.

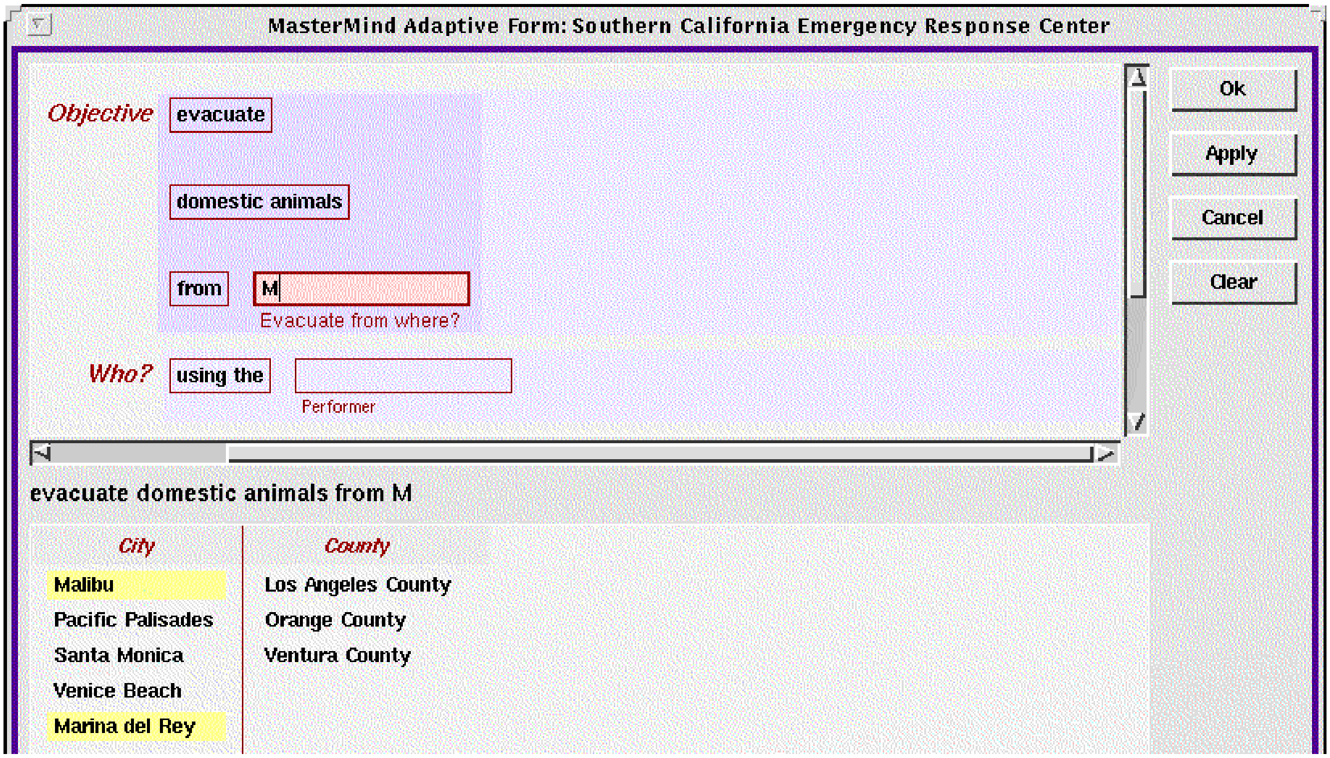

Figure 3. Emergency Response Center Example, Screen Three

The order of the possible values for the current field is defined by the forms designer, and may be in alphabetical order or in a domain-specific order (in Figure 3, the cities shown line the coast near Los Angeles, and are listed from North to South).

Figure 4 shows the interface after the user has clicked on Malibu (selecting a current choice with the mouse is equivalent to typing a disambiguating prefix of it and pressing the Tab key). After a top-level clause is completed it reverts to a single-row layout to conserve screen space (see the Objective clause in Figure 4). One more nesting level is shown for the Who? clause as it became the active clause, and the Sheriffs Department choice is highlighted in green as it is the only choice matching the user input “S” (meaning that hitting the Tab key now will select it). In addition, the auto-completion “heriffs Department” is shown in low contrast in the field itself.

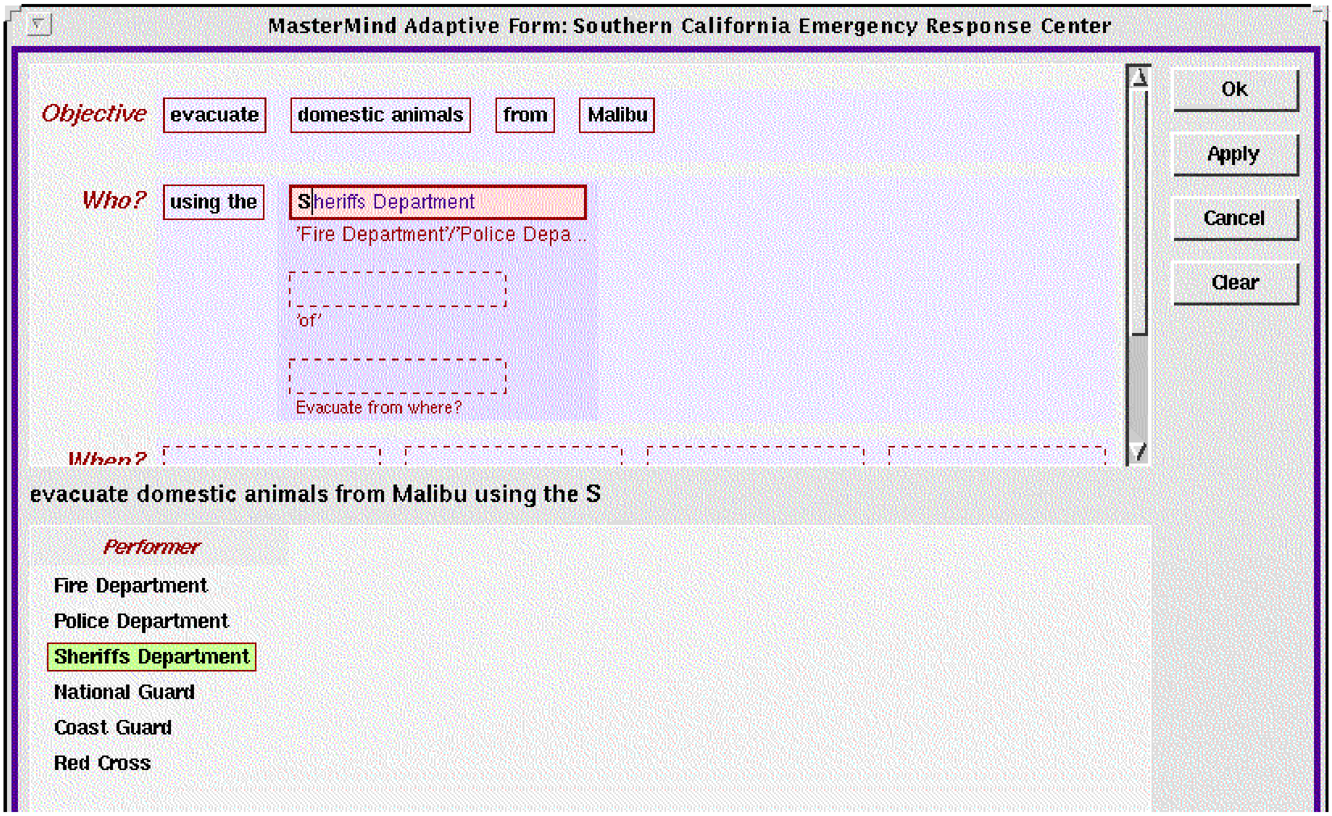

Figure 4. Emergency Response Center Example, Screen Four

Not shown in this example is the entry of user-defined text, such as of a custom city. There are two possible cases: (1) the input of user-defined text is expected, but some often-used choices are presented for the user's convenience, or (2) the users are strongly encouraged to stay with the pre-defined choices. In the former case, the interpreter accepts the input without question while in the latter case users have to explicitly press a "Type my own" button to deviate from the standard choices.

FORM SPECIFICATION

The specification language for Adaptive Forms is based on context free grammars. Our notation is similar to Backus-Naur Form, but augmented with labels that can be attached to the non-terminal on the left-hand side of the production. These labels are used to generate prompts for the fields and headers for the menus. The following three grammar rules illustrate our notation:

```
evacuateObjective "Evacuate":
  'evacuate' evacuateEvacuee 'from' evacuateFromClause;
evacuateEvacuee "Evacuees":
  'domestic animals' |
  'civilians' |
  'law enforcement personnel';
evacuateFromClause "From": city | county;
```

Our convention for writing the grammar specification is the following: symbols without quotes (e.g., evacuateObjective) represent non-terminal symbols in the grammar; symbols in single quotes (e.g., 'domestic animals') represent terminals in the grammar, and symbols in double quotes are labels attached to non-terminals.

Developers can think of a grammar as defining the set of sentences that user can enter (language parsing view), or as defining the data users can enter (structured editor view). We encourage the structured-editor view, where non-terminals represent object types, and terminals represent literal values of a given type.

The first production shown above illustrates the specification of a structured object containing two attributes (`evacuateEvacuee` and `evacuateFromClause`). The terminal '`evacuate`' can be thought of as marker for the structured object, which can be used to distinguish it from other objects having attributes with the same types. The terminal `'from'` can be thought of as a marker for the following attribute, as it is used to distinguish between objects with the same structure. The second production defines a primitive type (`evacuateEvacuee`) and specifies the values or instances of this type. The third production illustrates the definition of sub-types. The type `evacuateFromClause` is defined as having two subtypes, `city` and `county`. This means that any attribute that is declared to hold a `evacuateFromClause` can be filled with a `city` or a `county`.

The form for the example specification above will initially have four fields, corresponding to the four symbols in the expansion of evacuateObjective. The first and third fields will be filled by literals ('evacuate' and 'from'), and they will behave as labels (they will be tabbed over automatically). The choices for the second field (evacuate whom?) will contain all the literals in the expansion of the who non-terminal ('domestic animals', etc.) The choices for the fourth field (evacuate from where?) will have two columns, one for cities and one for counties.

An important aspect of Adaptive Forms is that it is trivial to extend the specification to include new data. For example, suppose that we want to extend the form to allow evacuations from the intersection of two streets (e.g., from a building at Lincoln Boulevard and Mindanao Way). We would modify the rule for `evacuateFromClause` to include a new element called `evacuateLocation`, and we would define a new rule for `evacuateLocation`:

```
evacuateFromClause "From":
  city | county | evacuateLocation;
evacuateLocation "Location":
  'building at' road 'and' road;
```

The Adaptive Forms interpreter constructs the forms as appropriate. If the system is set to produce short forms, the initial form would look as described before (four fields). Now, once the user selects “building at” for the fourth field, the interpreter determines that the user is entering a location, and adds three new fields to the form, two to enter roads, and one pre-filled with `'and'`. If the system is set to produce expanded forms, the initial form would contain seven fields: the four fields of the original example, plus the three new fields required for evacuation from an intersection. The three new fields would be marked as optional because the user only has to fill them if he or she enters `'building at'` in the first field for location (where a city or county name can also be entered). When this happens, the system knows that the user is entering a `evacuateLocation`, so it converts the fields from optional to required. If the user does not enter `'building at'`, the three optional fields are removed.

Below we show the complete grammar used to generate the screen shots for our running example. We require that the grammar be unambiguous (more precisely, that for all right-hand sides of a non-terminal their FIRST sets do not intersect [1]). This restrictions facilitates writing a recursive-descent parser that requires no backtracking.

The first section of the grammar provides for user-defined symbols, if any. In this particular grammar, we let users input their own city names if the grammar does not already contain them. The definition consists of a name for the symbol, a regular expression for the allowed character sequence, and a quoted string that can be shown to the user.

```
// Regular expressions for user input.
UserDefinedCity ([a-zA-Z][a-zA-Z ]+) "Custom City"
Cardinal ([0-9]+) "Positive Number"
ArbitraryText () "Free-form Text"
```

Note that we do not allow two user-defined symbols to be active at the same time as that would introduce ambiguity. For example, if UserDefinedCity and ArbitraryText were active at the same time, then we could not tell which one was selected by the input “Los Angeles”. This restriction is somewhat draconian as there are situations where we could tell them apart (e.g. UserDefinedCity and Cardinal); in practice, we currently disambiguate by pre-fixing user-defined symbols with other symbols that make them unique (e.g. ‘City’ ‘of’ <UserDefinedCity> | ‘County’ ‘of’ <UserDefinedCounty> rather than just <UserDefinedCity> | <UserDefinedCounty>). The next section of the grammar contains the actual rules.

```
z "Southern California Emergency Response Center": objective who when where;

city "City":
  ‘Malibu’|’Pacific Palisades’|
  ‘Santa Monica’|’Venice Beach’|
  ‘Marina del Rey’|’Playa del Rey’|
  ‘El Segundo’|’Manhattan Beach’|
  ‘Redondo Beach’|’Palos Verdes’|
  UserDefinedCity;
county "County":
  ‘Los Angeles County’|
  ‘Orange County’|
  ‘Ventura County’;

objective "Objective":
  evacuateObjective|ensureObjective;
evacuateObjective "Evacuate":
  ‘evacuate’ evacuateEvacuee
  evacuateFromClause;
evacuateEvacuee "Evacuees":
  ‘domestic animals’|
  ‘civilians’|
  ‘law enforcement personnel’;
```

```
evacuateFromClause "Evacuate-from-where clause”: | ‘from’ evacuateFromWhat;
evacuateFromWhat "Evacuate from where? ": city | county;
ensureObjective "Ensure":
  ‘ensure’ ensureWhat;
ensureWhat "Ensure what? ":
  ‘water supply’ | ‘use of roads’;

who "Who? ": ‘using the’ performer;
performer "Performer":
  ‘Fire Department’ ‘of’ city|
  ‘Police Department’ ‘of’ city|
  ‘Sheriffs Department’ ‘of’ county|
  ‘National Guard’|’Coast Guard’|
  ‘Red Cross’;

when "When? ":
  ‘before’ time|’after’ time|
  ‘between’ time ‘and’ time|’during’ phase|;
time "Time": timeDay timePlusMinusPhrase;
timeDay "Event" :
  ‘occurence-of-disaster day’ |
  ‘abandon-search-for-survivors day’;
timePlusMinusPhrase "Plus-or-minus clause": | timePlusMinus timeNumber;
timePlusMinus "+/-" : ‘+’ | ‘-’ ;
timeNumber "#" : ‘1’|’2’|’3’|’4’|’5’|‘6’|’7’|’8’|’9’|’10’|Cardinal;
phase "Phase":’Phase I’|’Phase II’|’Phase III’|ArbitraryText;

where "Where? ": areaModifier area|;
areaModifier "Area modifier": ‘in’|’around’|’over’|’at’;
area "Area":city|county;
```

RUN-TIME INTERPRETATION AND LAYOUT

The Adaptive Forms interpreter is a stand-alone X-Windows application that is called by a client application via the UNIX pipe mechanism. The Adaptive Forms interpreter then notifies the client application with the data (the parse tree) whenever the user submits a form (the Ok button submits the data and quits the Adaptive Forms interpreter, the Apply button just submits).

We will use the following terminology in this section. A grammar consists of regular expressions and rules (there are 3 regular expressions and 22 rules in our example grammar above). A rule has a name, a label, and one or more expansions (the “evacuateFromWhat” rule has two expansions, “city” and “county”). An expansion consists of zero or more symbols. An individual symbol is either a non-terminal, or a literal, or a reference to a regular expression (the “who” rule above has a single expansion consisting of the literal ‘using the’ followed by the non-terminal performer, and the last expansion of the “phase” rule consists of a reference to the regular expression ArbitraryText).

We use a recursive-descent parsing technique (as opposed to a shift-reduce technique such as LALR parsing) [1] so that the parse tree is explicitly represented at all times. This is convenient because the algorithm for laying out fields based on the grammar and the current state can then be implemented as a recursive function on the parse tree.

At start-up, and after the user competes input for a field, we “trivially-expand” the parse tree. This is done by placing the initial symbol on top of the parse tree and by then pushing rules with a single expansion onto the parse tree in a left-most derivation until we encounter a rule with more than one expansion (for which we need user input to make the decision on which of its expansions to follow).

In our running example, the start symbol is z, which has only one right-hand-side expansion “objective who when where”, and is thus expanded. The rule for the objective non-terminal has more than one expansion, so that the trivial-expansion process stops (even though the who non-terminal could be trivially-expanded, the reason for not doing so will become clear shortly). Thus, the initial parse tree is as shown in Figure 5.

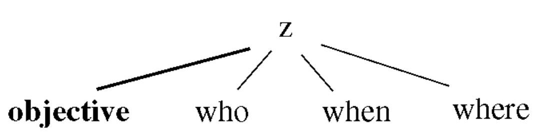

Figure 5. Trivially-expanded parse tree for the running example

In our terminology, “objective” is the active non-terminal (the one for which we currently need user input to decide which of its expansions to follow) and the path from the root to the active non-terminal is called the active path of the parse tree.

We control the screen layout by two “look-ahead” factors. The forward look-ahead determines how far ahead of the active non-terminal we are looking. The sideward look-ahead determines how far ahead of the leaf non-terminals not on the active path we are looking (Figure 6.)

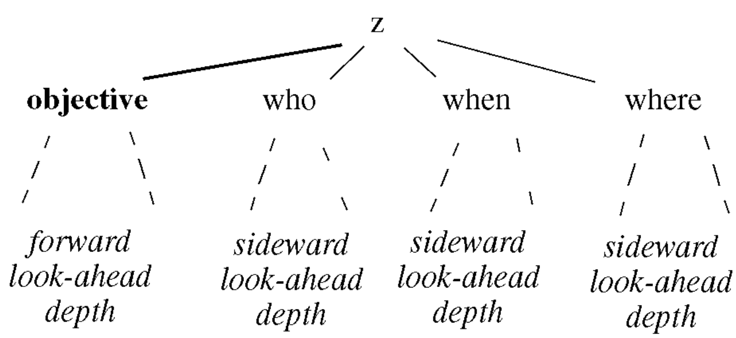

Figure 6. Forward look-ahead vs. sideward look-ahead

For example, the initial screen of our running example (Figure 1) had a forward look-ahead of one - that is, we go forward one from the active non-terminal objective so that we have the alternatives for evaluateObjective and ensureObjective to present. The labels for those two alternatives are (1) ‘evacuate’ Evacuees Evacuate-from-where-clause and (2) ‘ensure’ Ensure-what?. The fields on the screen are then produced by transparently overlaying the alternatives. The screen in Figure 1 also had a sideward look-ahead of one: we applied the same procedure of looking one ahead of who, when and where.

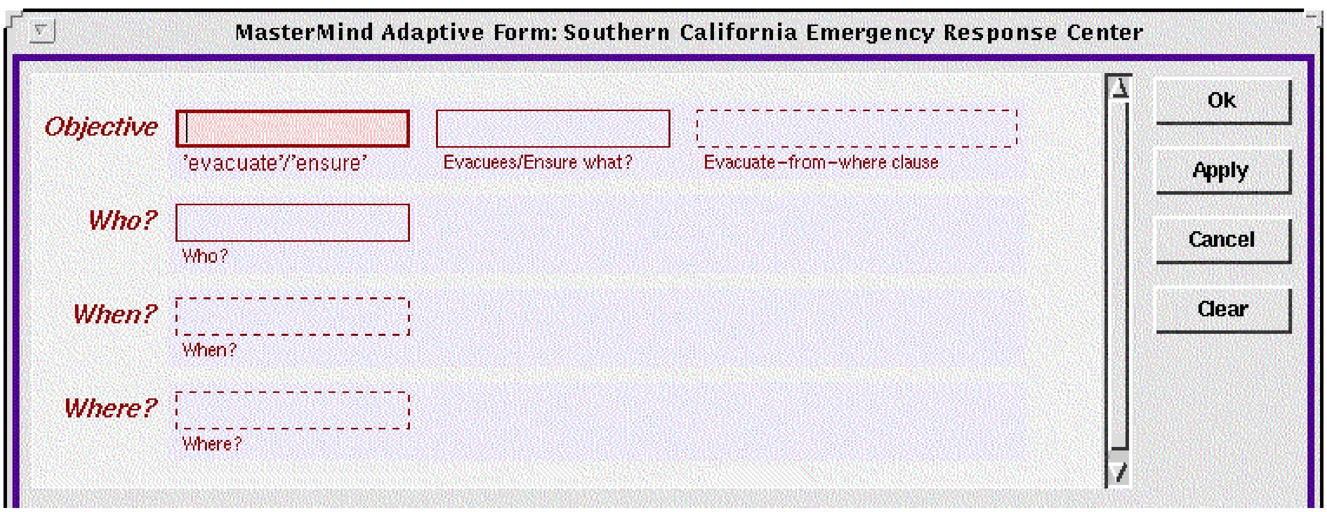

Figure 7. Forward look-ahead of one, sideward look-ahead of zero (compare to Figure 1)

Figure 7 shows the effect of setting the sideward look-ahead to zero, and Figure 8 shows the effect of setting the sideward look-ahead to two while setting the forward look-ahead to zero. Figure 7 also illustrates why we did not trivially-expand the who non-terminal in the parse-tree: doing so would make the who clause appear different from the when and where clause in Figure 7 because it would in effect force its sideward look-ahead to always be at least one.

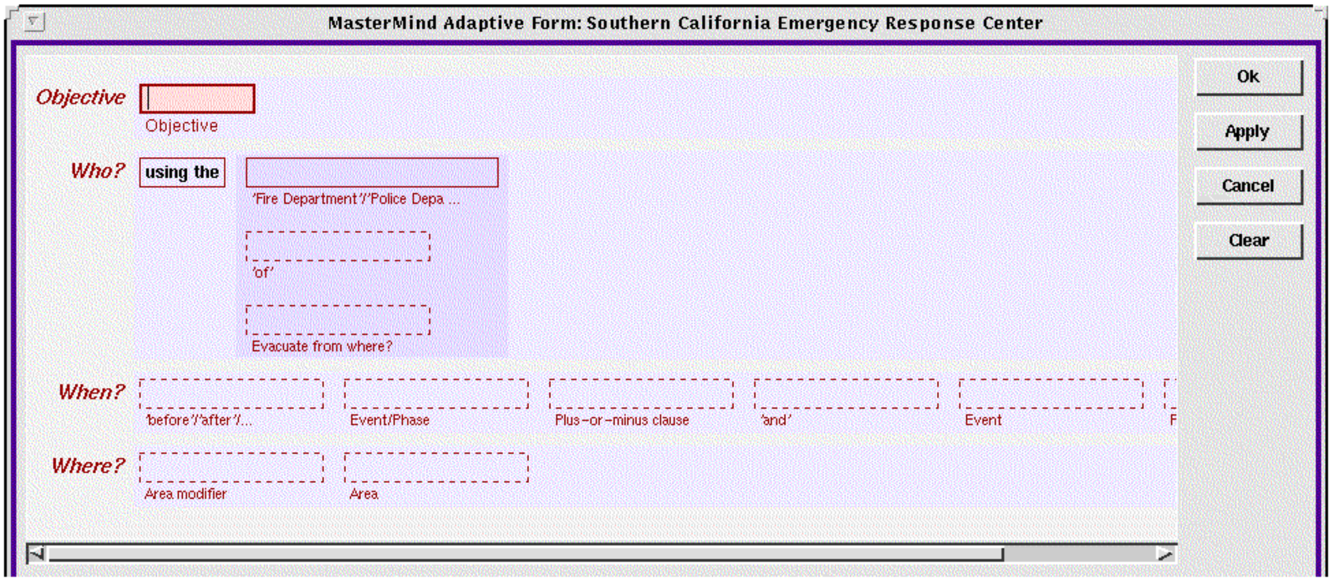

Figure 8. Forward look-ahead of zero, sideward look-ahead of two (compare to Figure 1)

In our experience, the optimal forward look-ahead is one or two, while the optimal sideward look-ahead is zero or one. The obvious trade-off is that an overly shallow look-ahead leaves the users in the dark while an overly deep look-ahead may confuse them because of the large number of fields. In addition, a deeper look-ahead demands more computational resources (while we have not yet encountered a situation where this was an issue).

RELATED WORK

The system most closely related to this work is NLMenu from Texas Instruments, which also implements an interactive parser for a grammar, albeit aimed at natural language understanding [8,9]. It used a simple tiled-columns layout for presenting the possible sentences, and it also did not collapse the already-entered part of a sentence (it keeps showing the choices for past fields, which consume a large amount of screen space). The type of grammars and the parsing algorithms used in these systems are also different. NLMenu accepts an ambiguous grammars and uses a parser based on push-down automata. Adaptive Forms are driven by unambiguous grammar, and uses a recursive descent parser.

Our research should not be confused with earlier work on syntax-directed editors (see [7] for a good overview) - our focus is on forms and simple languages, not on general-purpose programming languages.

There are several tools for building forms that support dynamic hiding and exposure of fields. ActiveForms [10] is a system written in Tcl/Tk that runs under SurfIt, a Web browser also written in Tcl/Tk. ActiveForms allows developers to write Tcl scripts that are executed each time a user enters a value in a field. ActiveForms allows these scripts to modify the form, by adding or deleting fields. To achieve this dynamic behavior the developer not only needs to attach a script to a field to create and define the layout of new fields, but also must be careful to write the scripts that remove fields when they are no longer needed. In our system, this work is done automatically. A nice aspect of ActiveForms is that the scripts can also be used for other purposes such as validating input, dynamically calling application procedures to generate menus, etc. We plan to extend Adaptive Forms with an application programming interface that will allow developers to define procedures that dynamically compute the set of terminals for a non-terminal, and also to perform input validation that is cumbersome or impossible to encode in a grammar.

Systems like Amulet [5], which use constraints, can also be used to specify relationships between fields so that the values and set of choices for one field can be computed based on the values of other fields. Amulet even allows the constraints to create new graphical objects, so it would be possible to write constraints that add and delete fields in a form. However, like in ActiveForms, developers must explicitly write programming language code for that purpose (in C++).

Dynamic Forms [4] is another system that provides some capabilities to hide and expose fields in a form. Dynamic Forms allows forms to be organized in a hierarchy, and the forms interpreter provides a facility to let users hide or expose branches of the hierarchy. This feature allows users to conveniently view large forms. The main difference with Adaptive Forms is that fields are exposed and hidden under user control and not based on the data that the user is entering. The two systems are complementary, and we envision adding the ability to open and close the top-level sections of a form. Dynamic Forms also provides capabilities to validate and propagate field values, but developers must write Java code to implement these features for their forms.

CURRENT WORK

Since we wrote the original paper [3], we have completely abandoned our C++ implementation and moved to Java. We have also implemented two major applications using Adaptive Forms, one for maintaining business rules and one for entering air campaign plans (Figure 9).

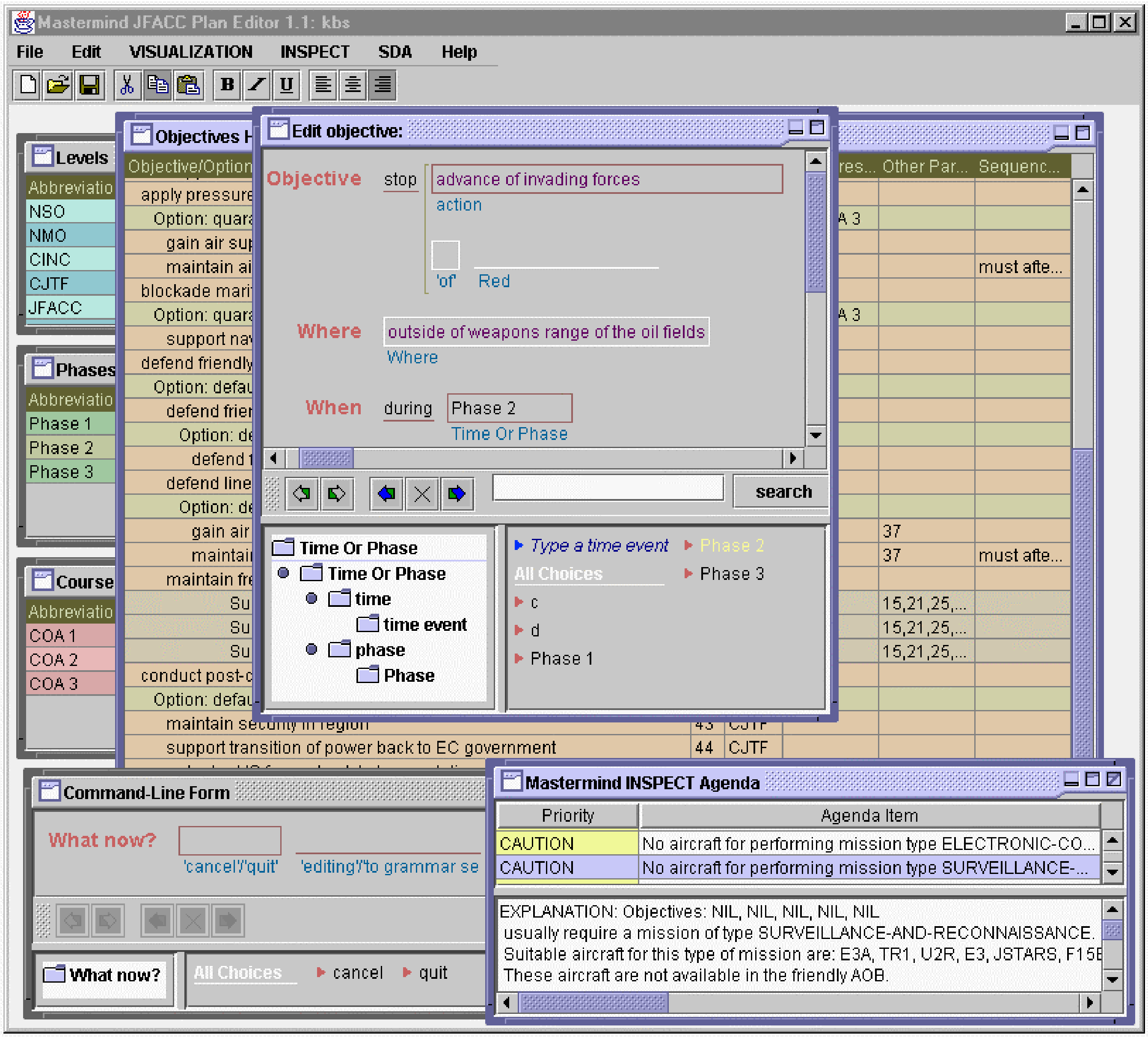

Figure 9. The Air Campaign Planning application (as a standalone Java application)

In the application shown in Figure 9, we make use of Adaptive Forms for two different purposes: for entering structured objectives (centered sub-window) and for providing a command-line style interaction with the editor in addition to direct manipulation (lower left sub-window). The central motivation for developing Adaptive Forms was that Artificial Intelligence applications need to receive input (such as objectives) in a structured form while ordinary humans tend to resist having to use an overly formal specification language; the Adaptive Forms mechanism seems to be a great compromise between those conflicting goals for certain applications. The lower-right sub-window shows such an Artificial Intelligence application (not built by us) that can inspect air campaign plans by consulting a domain-specific knowledge base.

We have now also fully implemented the Adaptive Forms mechanism as pure HTML pages based on Java servlets. Figure 10 shows the exact same air campaign planning application, but in its World-Wide Web incarnation – it can be used by anyone with a forms-enabled Web browser (no Java or plug-ins required). A nice side-effect of using Adaptive Forms for user input is that domain-specific application code can be written at a level of abstraction slightly higher than that of toolkits (by making use of colored and indented tables for output over 90% of our code for the two versions is identical).

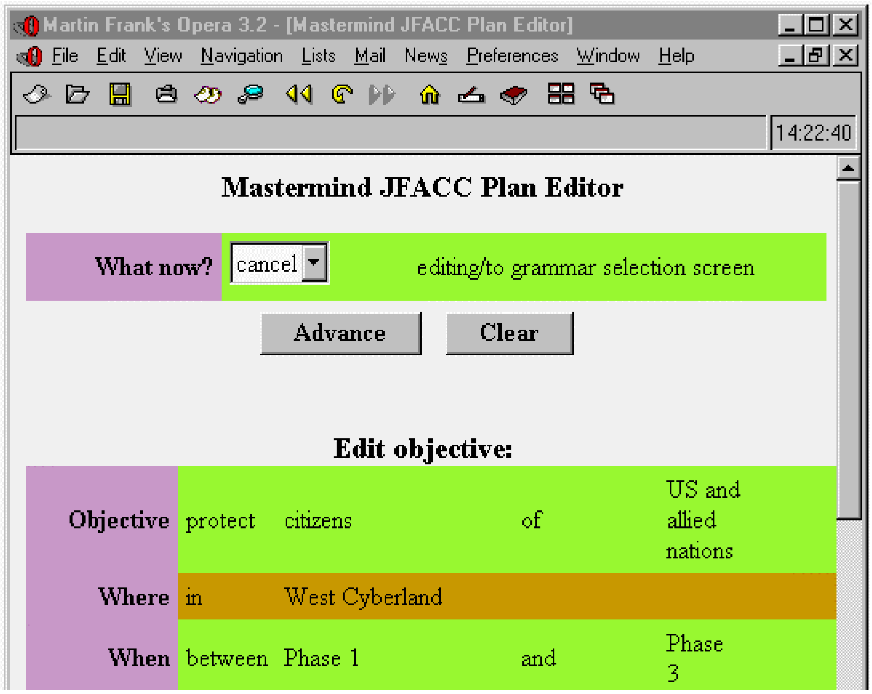

Figure 10. The Air Campaign Planning application (as HTML-based application)

FUTURE WORK

Recursion The grammar currently prohibits any form of recursion, so that it is not possible to specify that a non-terminal can appear a variable number of times. This limitation has already become apparent in a database query application of Adaptive Forms in which a possible sentence is "select * where a=1 and b=1 and c=1". The number of "and" clauses is variable in this application. Our current stop-gap measure is to have a fixed number of non-terminals called *and1* through *and5*, each of which can expand into epsilon (into the null clause). The envisioned solution is to allow recursion in the grammar, and to enhance the run-time interpreter for detecting cycles. It could then present an upcoming variable number of fields using an ellipsis and optional fields ('where' *and-clause* 'and' ... 'and' *and-clause*).

Application Programming Interface (API) In the same domain, we have also encountered the need for an application programming interface that lets the controlling application adapt the grammar as the user moves along. In our specific situation, imagine that there is one hundred tables in the database with ten attributes each and that the query language allows arbitrary joins between them. It is impossible to compute a static grammar encompassing all possibilities up-front because of the combinatorial explosion involved. Instead, only a skeleton grammar should be read in at first, and the further choices be computed from the database as the user goes along (this approach is currently being investigated by the SIMS project [2]).

CONCLUSIONS

Our experience with Adaptive Forms is limited but encouraging. We used the system for constructing an editor that allows air campaign planners to specify objectives (67 rules), and for constructing a database query interface (837 rules). The objectives editor allows users to enter thirty-two different kinds of objectives and was developed in consultation with domain experts. It has been used in several simulated exercises, but has not yet been deployed for routine use.

We have not performed any formal usability studies to determine whether users can in fact enter objectives faster using the editor than typing the paraphrase in English (one of the original design goals), but it appears so empirically. The grammar for the objectives editor was built by ourselves, and handed over to other developers who integrated it with other software. These developers were able, without any documentation or help from us, to enhance the grammar, and to also construct a new grammar for a different domain.

In conclusion, Adaptive Forms are attractive to developers because they can produce apparently custom-built, high-quality, domain-specific form-based user interfaces without ever having to deal with a user interface toolkit. With the enhanced capability of accepting dynamically computed grammars, Adaptive Forms can also double as easy-to-use, context sensitive query interfaces to databases.

ACKNOWLEDGEMENTS

We gratefully acknowledge DARPA funding of the MasterMind project as part of the Human-Computer Interaction Initiative and the JFACC Program.

REFERENCES

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Series in Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts, 1977.

[2] Y. Arens, C. A. Knoblock, and W.-M. Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems*, 6(2/3):99-130, June 1996.

[3] M. Frank and P. Szekely. Adaptive Forms: An interaction paradigm for entering structured data. In *Proceedings of the International Conference on Intelligent User Interfaces*, pages 153-160, (San Francisco, California, January 6-8) 1998.

[4] A. Girgensohn, B. Zimmermann, A. Lee, B. Burns, and M. E. Atwood. Dynamic forms: An enhanced interaction abstraction based on forms. In *Proceedings of Interact'95, Fifth IFIP Conference on Human-Computer Interaction*, pages 362-367. Chapman & Hall, (London, England) 1995.

[5] B. A. Myers, R. McDaniel, R. Miller, A. Ferrency, P. Doane, A. Faulring, E. Borison, A. Mickish, and A. Klimovitski. The amulet environment: New models for effective user interface software development. Technical Report CMU-CS-96-189, Carnegie Mellon University School of Computer Science, November 1996.

[6] R. M. Stallman. Emacs: The extensible, customizable, self-documenting display editor. Technical Report 519, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, August 1979.

[7] G. Szwillus and L. Neal, editors. *Structure-Based Editors and Environments*. Computers and People Series. Academic Press, San Diego, California, 1996.

[8] H. R. Tennant, K. M. Ross, R. M. Saenz, C. W. Thompson, and J. R. Miller. Menu-based natural language understanding. In *Proceedings of the 21st Annual Meeting of the Association for Computational Logistics*, pages 151-158, (Boston, Massachusetts, June 15-17) 1983.

[9] H. R. Tennant, K. M. Ross, and C. W. Thompson. Usable natural language interfaces through menu-based natural language understanding. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 154-160, (Boston, Massachusetts, December 12-15) 1983.

[10] P. Thistlewaite and S. Ball. Active FORMs. In *Fifth World Wide Web Conference*, 1996. http://www5conf.inria.fr/fich_html/papers/P40/Overview.html.