A GRAMMAR-BASED APPROACH TO THE AUTOMATIC GENERATION OF USER-INTERFACE DIALOGUES

Michael L. Scott and Sue-Ken Yap

Department of Computer Science,
University of Rochester,
Rochester, NY 14627.
scott@cs.rochester.edu, ken@cs.rochester.edu

ABSTRACT

An effective user interface requires a dialogue layer that can handle multiple threads of interaction simultaneously. We propose a notation for specifying dialogues based on context-free attributed grammars with two extensions: *fork* operators for specifying sub-dialogues and *context* attributes for dispatching tokens. The notation is useful both as a means of communicating the behavior of the dialogue layer to designers and as input to a dialogue compiler that generates program code. In this paper we explain the motivation for our work and provide practical examples of the use of fork and context. In addition, we outline algorithms for parsing and for generating parser tables.

KEYWORDS: User interfaces, human factors, interaction techniques, grammars, parsing.

INTRODUCTION

The proliferation of computers among unsophisticated users, together with the availability of high-quality graphic displays, has focused increasing attention in recent years on the human factors of computing. Otherwise excellent programs can be rendered unsuccessful, or even useless, by the lack of an effective user interface. Unfortunately, high-quality ad-hoc interfaces tend to be among the most complex, subtle, and tedious portions of traditional programs. Like other researchers in User Interface Management Systems, we believe strongly in the need for formal specification of user interfaces, both as an aid to understanding program behavior, and as a technique to permit automatic generation of large amounts of code.

Our work is based on the Seeheim Model [20], which divides programs into an application layer, a dialogue layer, and a presentation layer. The application layer handles the "real work" of the program. The presentation layer hides device dependencies and performs low-level input and output processing. The dialogue layer is the heart of the user interface. It controls the flow of information between the outer two layers, ensuring its consistency and determining its structure and timing.

In this model the dialogue plays both an imperative and a declarative role. In addition to providing the algorithm for handling input and output tokens as they arrive, the dialogue also *describes* the valid sequences of such tokens from both the application and presentation points of view. We use grammars to specify this dialogue layer. In contrast to such notations as augmented transition diagrams and systems of event handlers, grammars are comparatively concise and abstract. They can enhance the intelligibility of an interface significantly by separating the specification of what constitutes a valid conversation between program and user from the details of how such a conversation is implemented. In addition, the large difference in abstraction level between grammar and programming language means that automatic generation of code constitutes a significant savings of programmer effort. Experience with programming language compiler compilers suggests that a user interface dialogue compiler could be a very useful tool.

Unfortunately, ordinary context-free grammars are

inadequate for the interactive, multi-threaded dialogues of modern interfaces. Two principal problems arise. First, users will often wish to conduct several concurrent conversations with an application, conversations that may or may not be nested. The dialogue's parsing algorithm must be able to handle arbitrary interleavings of concurrent conversations, dispatching each token to the appropriate production. Second, input must often be parsed differently depending on the window, or *context* in which it was entered. Since the number of contexts cannot generally be predicted at dialogue-generation time, a notation for specifying dialogues requires a mechanism for expanding the input alphabet in a regular way at run time.

The need for multi-threading arises not only in concurrent conversations, but also in the management of input from multiple devices. Buxton and Myers [3] found two-handed input for user tasks to be an improvement over single-handed input. Projects such as the Sassafras UIMS [11,12] already provide support for both types of concurrency, but existing work has yet to combine such support with the conceptual advantages of grammar-based notation.

We address the shortcomings of context-free grammars with a pair of additional concepts. For concurrent conversations, we allow productions to *fork* off sub-parsers that continue in parallel. For multiple contexts, we introduce a special synthetic attribute for tokens that can assist in driving the parse. Fork productions and context attributes are related in the sense that separate conversations will usually take their input from separate contexts. However, a single-threaded conversation may use several contexts, and a multi-threaded conversation may only require a single context. For this reason, we present the notions of fork and context as essentially independent.

FORK OPERATORS

In order to support concurrent, interleaved conversations, we augment context-free grammars[1] with two new operators: the parallel and operator: **&&**, and the parallel or operator: **||**. Both operators cause the creation of sub-parsers which work in parallel.

> Two productions joined by **&&** are a production. The combined production succeeds when both of the sub-productions succeed. More formally, the combined production generates all interleavings of strings generated by the sub-productions.

> Two productions joined by **||** are a production. The combined production succeeds when either of the sub-productions succeeds. More formally, the combined production generates all interleavings of a string generated by one sub-production with a prefix of a string generated by the other sub-production.

The **&&** operator is useful when the order of input is immaterial. Filling in a form is one example:

```
form → A && B
```

Both **A** and **B** must appear in the input for **form** to succeed. The tokens that **A** and **B** derive may appear in any order in the input.

The **||** operator is useful when the completion of one sub-production obviates the need for the other. User-generated interrupts (e.g. from hitting the *DELETE* key) are one example:

```
getnumber → interrupt || read_digits
```

The completion of either **interrupt** or **read_digits** will cause **getnumber** to succeed. Different levels of interrupts can be used to back the parser out to arbitrary pre-arranged positions.

Together, the parallel *and* and *or* operators allow us to construct a hierarchy of sub-parsers to manage a conversation with multiple levels of aborts, nested parallel conversations (e.g. for interactive help), and other useful structure.[2] More detailed examples can be found in the section on **USAGE**.

We impose one additional rule on the use of fork operators: it must always be possible to predict their use without lookahead. In other words, if A is the left hand side of a fork production and αA is a prefix of a valid sentential form, then *every* valid sentential form beginning with α must in fact begin with αA, or be derivable from a sentential form that begins with αA. This amounts to insisting that concurrent conversations must be started explicitly by user or application action, and need not be detected in response to the arrival of input from one of the branches. This rule simplifies parsing considerably. We also believe it to be consistent with natural dialogue structure.

[1] We use a hopefully self-explanatory extended Backus-Naur Form notation.

[2] It is worth noting that our augmented grammars are not in general context free. The language generated by

```
A → B && C
B → a B c | ε
C → b C d | ε
```

is one example; its intersection with the regular set $a^*b^*c^*d^*$ is $a^n b^m c^n d^m$.

CONTEXT ATTRIBUTES

In a multi-window application, otherwise indistinguishable inputs originating from different windows of the presentation layer must generally be passed through to the dialogue as distinct tokens. If an unbounded number of windows can be created at run time, it becomes impossible to enumerate all tokens in a finite grammar at dialogue-generation time. We therefore propose that all tokens possess at least two synthetic attributes: *value* and *context*. Value captures the usual notion of token type. Context allows the dialogue to differentiate between tokens of the same value from different sources.

For managing the use of context we adopt a notation based on left-attributed LL(1) grammars in simple-assignment form [2,15]. The inherited attributes of a symbol X in a production of an L-attributed grammar depend only on attributes of RHS symbols to the left of X or on inherited attributes of the LHS of the production. In an LL(1) grammar, all attributes can be evaluated left-to-right, in the course of the parse itself. The simple-assignment property requires that all dependencies be copy rules; computation is performed solely in action routines. The restriction to one token of lookahead is consistent with intuitive behavior for interactive systems.

We augment the usual predictive parsing algorithm to use context to guide the parse. In addition to matching in value, each token must also match in context. Actual context values are not known at dialogue-generation time, but the locations in which those values will appear at run time can be predicted. Each parse table entry describes where to find, at run time, the context for which the entry is valid. Value attributes index into the parse tables; context attributes are used to dispatch tokens to the appropriate sub-parser.

Attributes that contain context values must be identified to the dialogue compiler. Rules for the dialogue notation ensure that whenever parallel sub-parsers are able to accept tokens with the same value, those tokens will appear in different contexts. First, action routines that return context values in synthetic attributes are required to create new, unique values for each call. Second, the copy rules for a given production are not permitted to assign the same context value into two different inherited attributes of a non-terminal on the RHS or two different synthetic attributes of the non-terminal on the LHS. Furthermore, no context value may be copied into both branches of a fork unless the value alphabets of the two sub-productions are disjoint. Simply put, the branches of a fork either (1) partition the token value alphabet between them, or (2) only know the names of different contexts.

An alternative parsing algorithm results from imposing the further restriction that sub-parsers inherit a fixed and statically determinable number of contexts. This means that synthesized contexts may not be used within the sub-dialogue in which the action appears, but may only be passed to an inferior sub-dialogue. With this rule it is possible to re-write each sub-dialogue as a conventional context-free grammar, without context attributes. Arbitrary (e.g. LR) parsing algorithms can then be employed.

Limiting sub-dialogues to a fixed number of contexts is not as serious a restriction as it might at first appear. Artificial sub-dialogues may be introduced in order to change to a new and different context:

```
S → ... X ...
X → Y && ϵ
```

For convenience we can introduce a unary operator `!`, that serves to begin a new sub-dialogue for the following symbol, without the necessity of creating a trivial branch:

```
S → ... ! Y ...
```

Both of the parsing techniques (predictive and general) are discussed in the section on **ALGORITHMS**.

Example

Consider a simple dialogue that creates two new sub-windows upon receipt of a special key. In the following, inheritance rules are shown in brackets. `W.ctx = S.ctx` means that the `ctx` attribute of `W` is copied from the `ctx` attribute of `S`.

```
S → W
                                        [W.ctx = S.ctx]
W → char+ && new N
                                  [char.ctx = W.ctx,
                                    new.ctx = W.ctx]
N → #create (X || Y)
                               [X.ctx = #create.ctx1,
                               Y.ctx = #create.ctx2]
```

`new` is a token that is not generated by `char` (for example, a special key), obeying rule (1). The action `#create` produces two new windows and passes their contexts to `X` and `Y`, obeying rule (2).

ALGORITHMS

Our predictive parsing algorithm differs from that of a standard LL(1) parser in two important ways. First, prediction of a forked production suspends the current parser and creates sub-parsers that continue in parallel, using their own parse tables. Completion of one (for `||`) or all (for `&&`) sub-parsers allows the suspended parent to continue. Second, each parse table entry contains

context information that is used to determine which of several running parsers should receive each input or output token.

The production $X \rightarrow \alpha$ may appear in entry $[X, a]$ of a parse table for either of two reasons. It may be that a is in $FIRST(\alpha)$, in which case the context expected for a can be found in one of the attributes of X. The appropriate attribute can be determined at dialogue generation time (via a straightforward extension of the algorithm for building FIRST sets), and the choice between parallel parsers can be made by inspecting the symbol at the top of each stack.

Alternatively, α may generate ϵ, and a may be in $FOLLOW(X)$, in which case the context expected for a can be found in an attribute of some symbol farther down the parse stack. Unfortunately, the identity of this deeper symbol may not be uniquely determined by X and a; it may depend on the production in which X was originally predicted. One obvious remedy is to arrange for unique context information by using exact lookahead information instead of $FOLLOW$ sets when building parse tables. In the worse case, this solution is equivalent to modifying the grammar so that every symbol that can generate ϵ appears on the RHS of only one production. The result is a potentially enormous increase in parse table sizes. If typical grammars approach this worst-case behavior, it may be preferable to maintain conventional tables and cope dynamically with multi-valued context information. The rules described in the section on **CONTEXT** ensure that the context of the a will be acceptable to at most one of the currently-active parsers. We can tentatively pursue ϵ productions in all those parsers at once; resulting attempts to match the a will fail in all but one.

In order to ensure that the expected context of a token is known when the token is first encountered, we must insist that no action routine produce the context value for the token used to predict that action routine. Since it is arguably counter-intuitive for *any* action routine in an interactive program to require examination of a following token, it might be appropriate to enforce the stricter rule that we never need to look through an action routine to see the token that predicts it.

Alternative approach

If input to a sub-dialogue is limited to a fixed number of contexts, it becomes possible to transform an extended grammar with context attributes into a conventional context-free grammar without context attributes. Any standard parsing algorithm can then be employed. The idea is to replace each symbol X in the extended grammar with a new symbol for every possible set of values of X's context attributes. The dialogue compiler can assign a name to each of the contexts of the sub-dialogue even though the actual context values will not be known until run time. A table created when the sub-parser begins execution can be used to translate from <token value, context value> pairs to token values in the alphabet of the new, conventional grammar.

If a sub-dialogue inherits c contexts, a token in the original extended grammar may be replaced by up to c tokens in the new grammar. A non-terminal A with t inherited context attributes may be replaced by up to c^t non-terminals in the new grammar. Each of the productions for which A forms the LHS will be replicated up to c^t times. As with the use of exact lookahead sets in the predictive parsing algorithm, this technique has the potential to increase parse table sizes dramatically. It is not yet clear how much storage would be required for typical dialogue grammars.

The predictive parsing algorithm (with regular $FOLLOW$ sets) has the advantage of small, conventional tables. Either the exact lookahead sets or the alternative algorithm would save time on ϵ productions, but with potentially unacceptable space requirements. The alternative algorithm can be used with a shift-reduce parser, but but this may not be much of an advantage; predictive parsing seems ideally suited to interactive programs.

USAGE

We illustrate the use of our grammar notation with two applications: a source level debugger and a rudimentary TTY driver.

Dbxtool

We describe a simplified version of the *dbxtool* source debugger for Sun Workstations [1]. *Dbxtool* has a command window, a button window, a status window and a source window. Assume that the command window parses its input stream and sends only complete tokens such as `print` or `continue` to the dialogue. The button window provides many of the commands of the command window in the form of radio buttons for user convenience.

The initial production starts up all windows:

```
S → #create (C+ || status+ || source+)
                          [C.cmd = #create.cmd,
                           C.but = #create.but,
                      status.ctx = #create.st,
                    source.ctx = #create.src]
```

The action `#create` starts up all four windows. Inputs from both the command and button windows are accepted by the sub-productions of `C`. For each radio

button, C derives two productions, one starting with the command typed in and the other with the command clicked on. In the productions below, the capitalized tokens are associated with and derive context from the button window. The actions triggered have been elided. The triggered actions are different in **print** and **PRINT** as the latter uses the currently highlighted entity.

```
C → print entity_name ...
C → PRINT #check_selection ...
C → stop location ...
C → STOP #check_selection ...
C → next | NEXT | step | STEP
```

One point not apparent from the grammar fragments above is that the dialogues for the sub-windows need not be grouped together. Productions can be placed close to related data structures and program actions to form interaction modules. With appropriate tools for building libraries and encapsulating modules, sub-grammars can provide an extremely effective form of dialogue abstraction.

TTY driver

Our second example is a TTY driver that obeys the XOFF/XON convention for suspending output. This example shows how a grammar can handle input from more than one source, in this case the program and the user. It also demonstrates that although we have used examples drawn from windowing systems, the notation is general enough to describe any kind of dialogue that handles information as tokens.

```
TTY → (char #copy | XOFF+ XON)+ || ABORT
      [char.ctx = TTY.out, XOFF.ctx = TTY.in,
       XON.ctx = TTY.in, ABORT.ctx = TTY.in]
```

#copy is an action that sends the character to be output to the presentation.

PREVIOUS WORK

In addition to context-free grammars, two other major classes of notation have been proposed for specifying dialogues [6]: transition diagrams and event handlers.

Transition diagrams have been used by Newman, Edmonds, Guest, Jacob and Wasserman [4,8,14,17,22]. Nodes in the network correspond to states in the program and arcs to actions that cause a change in state. Actions are triggered by user input. Recursive transition diagrams are needed to represent nested interactions. Transition diagrams provide an excellent means of presenting information visually; Harel [10] uses them as the basis of a visual programming notation called *statecharts*. The major drawback with transition diagrams is the verbosity of the representation. A typical textual representation of a transition diagram enumerates for each state: the input tokens, the successor states resulting from acceptance of tokens, and actions, if any.

Event handlers were proposed by Green [7] and have been used in the U. of Alberta UIMS [7] and ALGAE [5]. The dialogue layer is divided into event handlers. Each handler contains internal state which may be altered by the execution of actions upon the receipt of events from outside. The source of events may be the application, the presentation, or another handler. Event handlers may choose the types of event they are willing to accept. The entire collection of handlers resembles an object-oriented system such as Smalltalk-80, except that inheritance of properties is not generally required. Event handlers have the drawback that the input/output language cannot readily be determined without inspecting handler code. Clarity also suffers from the fact that an event may activate more than one handler. It may not be easy to determine which handlers will be active at any given time. Finally, because they so closely resemble program code themselves, event handlers provide relatively little opportunity for labor-saving compilation. The abstraction level of grammars is significantly higher.

Context-free grammars have been used by Hanau and Lenorovitz [9], and Olsen [18,19]. The Input Tools notation of Van den Bos [21] can in some sense be considered a cross between event handlers and CFG notations. Our work borrows the fork operators from Input Tools. To our knowledge, no previous work has approached the problem of providing multi-threading and token dispatch in CFGs by extending the notation and hence the class of languages accepted. Unlike Input Tools, our method does not require prohibitive run-time overhead [16].

When arbitrary actions with side effects are allowed in the dialogue, all three major models have the same descriptive power as a Turing machine. In fact, a dialogue that does not require Turing-equivalent power is likely to be trivial. We prefer grammars because the notation is both concise and abstract, the input language is specified explicitly and the separation between input specification and action is clear.

STATUS OF WORK

We have completed the design of the dialogue language of our UIMS. We will begin soon to build a dialogue compiler that generates parse tables from grammars written in the notation described.

To gain acceptance by designers, a notation must allow hierarchical composition of dialogues, starting with

system-provided primitives. Libraries of often used interaction techniques will minimize redundant effort. Recent work has focused on object-oriented approaches to dialogue objects. Jacob [13] has described a system in which each object is specified with a single-threaded state diagram, and objects are combined to form the overall dialogue.

Our next goal is to design a language in which dialogue grammars, private data, and executable code are grouped into modular dialogue objects. We will provide the interface designer with a design tool to compose these interaction objects to form the user interface.

ACKNOWLEDGMENT

David Sher provided the proof that fork operators can generate non-context-free languages.

REFERENCES

1. Evan Adams and Steven S. Muchnick. Dbxtool: a window-based symbolic debugger for Sun workstations. *Software Practice and Experience*, July 1986.

2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.

3. William Buxton and Brad A. Myers. A study in two-handed input. In *CHI'86 Conference Proceedings*, pages 321–326, April 1986.

4. E. A. Edmonds. *Adaptive Man-Computer Interfaces*, pages 389–426. Academic Press, London, 1981.

5. Mark A. Flecchia and Daniel R. Bergeron. Specifying complex dialogs in ALGAE. In *Conference Proceedings of Human Factors in Computing Systems and Graphics Interface, Toronto, Canada*, April 1987.

6. Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

7. Mark Green. The University of Alberta user interface management system. *Computer Graphics*, July 1985.

8. Stephen P. Guest. The use of software tools for dialogue design. *International Journal of Man-Machine Studies*, 16:263–285, 1982.

9. Paul. R. Hanau and David. R. Lenorovitz. Prototyping and simulation tools for user/computer dialogue design. In *SIGGRAPH '80 Conference Proceedings*, pages 271–278, 1980.

10. David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

11. Ralph D. Hill. Event-response systems — a technique for specifying multi-threaded dialogues. In *Conference Proceedings of Human Factors in Computing Systems and Graphics Interface, Toronto, Canada*, April 1987.

12. Ralph D. Hill. Supporting concurrency, communication and synchronization in human-computer interaction — the Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

13. Robert J. K. Jacob. A specification language for direct-manipulation user interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.

14. Robert J. K. Jacob. Using formal specifications in the design of a human–computer interface. *Communications of the ACM*, 26(4):259—264, April 1983.

15. Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

16. J. Matthys. Recent experiences with input handling at PMA. In *User Interface Management Systems*, Springer-Verlag, 1985.

17. W. M. Newman. A system for interactive graphical programming. In *SJCC 1968*, 1968.

18. Dan R. Olsen Jr. Automatic generation of interactive systems. *Computer Graphics*, January 1983.

19. Dan R. Olsen Jr. and Elizabeth P. Dempsey. Syngraph: a graphical user interface generator. *Computer Graphics*, July 1983.

20. Günther E. Pfaff, editor. *User Interface Management Systems*. Springer-Verlag, 1985.

21. Jan van den Bos. Input tools – a new language construct for input-driven programs. In *Proceedings of the European Conference on Applied Information Technology of IFIP*, September 1979.

22. Anthony I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, SE-11(8):699–713, August 1985.