Pergamon

0096-0551(94)00016-6

Comput. Lang. Vol. 21, No. 1, pp. 1–16, 1995

0096-0551/95 $9.50 + 0.00

A GRAMMATICAL SPECIFICATION OF HUMAN–COMPUTER DIALOGUE

ALBERT NYMEYER

University of Twente, Department of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands

(Received 19 May 1994; revision received 10 November 1994)

Abstract—The Seeheim Model of human–computer interaction partitions an interactive application into a user-interface, a dialogue controller and the application itself. One of the formal techniques of implementing the dialogue controller is based on context-free grammars and automata. In this work, we modify an off-the-shelf compiler generator (YACC) to generate the dialogue controller. The dialogue controller is then integrated into the popular X-window system, to create an interactive-application generator. The actions of the user drive the automaton, which in turn controls the application.

compiler generators dialogue models human–computer interaction formal techniques

1. INTRODUCTION

In 1985, a model of the interaction between the user and application was defined at a workshop in Seeheim [1]. This model, now popularly called the Seeheim Model, and shown in Fig. 1, partitions an interactive application into 3 components; a presentation component, which we will refer to as the user-interface, a dialogue controller, and an interface to the application itself. The presentation component is, in fact, what the user experiences as the user-interface—it is that part of the interactive application that is seen, heard, touched or spoken to. Window technology has improved dramatically since 1985, resulting in ever-fancier and more highly featured user-interfaces. Application-interface technology, on the other hand, has barely changed in this period [2]. It is still quite difficult to separate the application from the user-interface in an elegant way.

The dialogue component is responsible for the sequence of events that constitutes the interaction between the user and the application. Although the term "dialogue" is taken from real-world, human–human communication, human–human dialogue is essentially different from human–computer dialogue. Human–human dialogue consists of two (symmetric) parties that communicate—the output from the one party is the input for the other, and context plays a crucial role (the meaning of a word is often highly influenced by its context). While human–computer dialogue, in abstract terms, also refers to the conversation between two parties, these parties are not symmetric, and context is not important (but may play a role). In human–computer dialogue, we usually only specify the input 'language' from the user—the response from the computer (application) is a side-effect. In this sense it would be more accurate to say that we are dealing with a model of human–computer monologue. To avoid confusion, however, we use the same terminology as in the literature.

A system that provides tools to design and develop the presentation and dialogue components, and to integrate these components into an application, is called a User-Interface Management System (UIMS). A UIMS, then, is based on a conceptual model that separates the user-interface, the dialogue and the application. A UIMS acts like an interactive-application compiler–compiler. The UIMS complies a user-interface, dialogue specification and the application into an interactive application. In this analogy, the dialogue specification between the user and the application constitutes the source program.

An application that is programmed directly into a user-interface (using a toolkit, for example) can lead to complex, difficult to maintain and non-portable code. Control-flow, or dialogue, in such an application is achieved by the judicious use and selectivity of widgets.

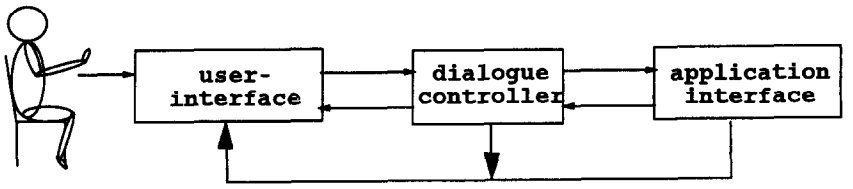

Fig. 1. The Seeheim model of human–computer interaction.

In this work we use a context-free grammar to specify the dialogue, and the parser generator YACC (Yet Another Compiler–Compiler [3])† to compile the dialogue. The X-window system is used to specify the user-interface. Together, these components comprise a UIMS. YACC was chosen to generate the dialogue controller for the following reasons.

- YACC is available in source-code form, and is easily modifiable.
- YACC is extremely popular.
- YACC parses LALR(1) grammars, which are more descriptive than LL(1) grammars.
- YACC is implemented in C, as is the X-window system.

This work shows that it is possible to build a direct no-fuss, and inexpensive implementation of a UIMS that clearly separates the user-interface, dialogue and application. It also provides insight into the relationship between each of these components.

In the following section we briefly review various models of UIMSs. In Section 3 we describe the compiler-generator system YACC, and we show how YACC can be used to specify the dialogue in an application. In Section 4 we describe salient features of the X-window system, and point out the shortcomings the system has in specifying dialogue. To build a UIMS, and interface the automaton generated by YACC to a user-interface, we need to make certain modifications to YACC. These modifications are explained in Section 5. In Section 6, we explain how an application can be built using the new tool. Finally, in Section 7 we present our conclusions.

2. FORMAL TECHNIQUES

Formal techniques have many advantages. A system that is formally specified can be analysed and its properties can be deduced. A system that is implemented from a specification can be readily and reliably updated and managed. Rapid prototyping is also enhanced. Problems that formal techniques do have include, firstly, real-world system specifications can be large, complex and possibly unreadable to all but experts, and secondly, the specification style and model may influence, or be influenced by, an implementation technique. Ideally, a specification should be independent of a particular implementation technique.

In a seminal work, Green [4] identified 3 formal categories of UIMSs: those that use a context-free grammar to specify the user-interface, those that use a transition network, and those that are event-driven. A fourth, currently very popular category, which is not formal, is called 'direct manipulation'. Green presents algorithms for converting formal specifications of the above forms into executable form. He finds that the event-driven approach has the greatest descriptive power, and presents (efficient) algorithms to convert the grammar and transition specification forms into event-driven specification forms. However, because each approach has its own particular advantage, Green advocates letting the interface builder use whichever approach he considers the best, and letting the UIMS translate context-free grammars and transition networks into an event-driven internal form.

Of the 3 formal methods, the *transition-network approach* has been around the longest. The first application of this approach to user-interface design was by Parnas [5]. The first real implemen-

†Actually, a public-domain version of YACC called 'Berkeley YACC' was used.

tations came in the mid-1980s, however, with work by Jacob [6] and Wasserman [7]. More recently, the system SCENARIOO has been developed [8]. The most usual criticism of this technique is that it is too verbose, it lacks abstraction, and networks become unwieldy and complex for realistic systems.

The *event-driven approach* is often based on an object-oriented model, where abstract data structures, and incoming and outgoing messages, are used to handle input and output, and the structure of the dialogue is defined by the arrangement of the objects and their protocol. Examples of systems that use this technique are SASSAFRAS [9], GWUIMS [10] and the Smalltalk system.

The *context-free grammar approach*, as the name suggests, is based on the formalism of context-free grammars, and compiler building. In this approach we consider the dialogue component as a form of translator, and specify the dialogue using a context-free grammar. At the lexical level we specify the basic events; for example, pressing a key or button, the position of the mouse, colour, choice of window, type of window and so on. At the syntactic level, the structure of the dialogue is specified; terminals are used to represent the input from the user, and nonterminals to determine the structure of the dialogue. Finally, at the semantic level, application routines are called. These routines may produce output. The lexical, syntactical, and semantic levels can be mapped in a natural way onto the Seeheim Model.

One of the first systems developed using this approach was by Olsen and Dempsey [11–13]. This system, called SYNGRAPH, is a user-interface generator for interactive system graphics. In this system, the tokens that are used to represent graphical devices are parsed according to a context-free grammar. The semantics of the grammar is a set of data types (that define the pictures to be displayed) and a set of procedures. Olsen and Dempsey address certain problem areas, namely, continuous sampling of devices, prompting for user inputs, cancellation and rub-out processing.

In Ref. [14], Olsen developed a system called IPDA (Interactive Push-Down Automaton). As with its predecessor, IPDA implements the syntactic component of the user-interface manager, but more attention is paid to the interfaces between the lexical and syntactical component, and the syntactic and semantic component. IPDA contains extensions that enable an automata to be used interactively. Additional kinds of transitions are introduced that increase the semantic control, allow the automaton to change state without input, and cater for exceptional conditions. The context-free grammar is also transformed before it is interpreted (for example, triggers are associated with sampled devices). A further development of this work was carried out in Ref. [15]. In this work, an IPDA-based dialogue controller was integrated with a dynamic display model. The system (called GRINS) was built to study the interface between the input (sequence of commands) and the graphical feedback. The dialogue controller communicated with a logical input-device handler and a constraint interpreter. A layout editor that supports the IPDA was also built.

Scott and Yapp [16] also used a context-free grammar to specify the dialogue. They find that conventional context-free grammars are unable to specify two important aspects of dialogue, namely multi-threading and context. Multi-threading is the arbitrary interleaving of concurrent conversations. To specify multi-threading, they introduced parallel operators to the grammar. These parallel operators allowed productions to be 'forked'. A forked production suspends the current parser and creates sub-parsers (processes) that continue in parallel on sub-grammars, using their own tables. The problem of context was solved using a synthesized attribute. This attribute determined which of several running processes should receive a particular token.

A more elaborate grammatical model that also incorporates event handling was used by van den Bos [17]. This model is based on a hierarchy of interaction modules, each of which uses an input expression to specify the input sequence that will trigger a certain user-interface or application task. An interaction module is an abstract data structure with local variables and procedures, and (links to) tasks. Structure is obtained by locally including definitions of lower-level modules. Modules at the lowest level define the coupling to physical devices. The system provides multi-thread and multi-device user interaction, as well as context-dependent prompting, echoing, feedback, error correction, and expertise levels.

The grammatical formalism of van den Bos is more powerful and more complex than context-free. Olsen *et al.* and Scott and Yap, use extended LL(1) grammars to specify the dialogue.

The advantages of using a (top-down) LL(1) approach is that it is simple, and intuitive. The top-down approach lends itself well to the 'action' followed by 'arguments' style of most user-interface commands. During parsing, the action will determine the unique production that must be applied.

The class of (extended) LL grammars is contained in the class of LR grammars. LR-based parsers work very differently than LL-based parsers. An LR-based parser works with states, which represent the progress made in a *number* of productions (concurrently). (If we let this number be 1, then we have an LL grammar.) If we use an LR grammar, we may still choose the LL-style of specification (i.e. action before arguments), but we can also do a lot more. LL is an artificial restriction: a larger class of languages can be specified using an LR grammar. This is an important point because it means that the user-interface designer that uses LR has more 'power' to specify dialogue. For example, we can specify cancellation and application-directed dialogue quite easily in an LR grammar, but not in an LL grammar.

In general, the advantage of the grammatical approach is that the lexical, syntactic, and semantic levels naturally model the three tiers in the Seeheim Model, and that the formalism is abstract and concise. The major disadvantages of the approach are:

- The sequence of user actions in a dialogue is not explicitly reflected in a context-free grammar. The context-free grammar must often be artificially structured (made more complex) to reflect the structure of the dialogue. The approach is too awkward and rigid.
- The semantics of the application is only implicitly specified by application routine calls, and is static, requiring re-compilation each time a change is made. This problem receives particular attention in the CHIMERA system [18].
- Continuous devices cannot easily be handled. The parser cannot know when to check (sample) a device to see if the setting has changed.
- User-unfriendliness—Olsen [19] reports that (student) user-interface designers had difficulty applying a formal-language technique to specify a dialogue.

The *direct-manipulation approach* [20], unlike the above approaches, is not 'formal'. This approach, sometimes referred to as 'interaction by example', is the most user-friendly technique of building a user-interface—no specification language needs to be learnt; the user-interface designer simply builds a user-interface using the interaction devices and controls placed at his disposal.

In general, direct manipulation interfaces cannot be made to conform to the Seeheim Model [21]. The problem arises because direct manipulation requires that the semantics (i.e. the application) be brought closer to the user-interface, and this works against a clean separation [22]. The greater the separation, the more responsibility the user-interface must assume. The problems encountered in fitting a direct-manipulation UIMS to the Seeheim Model, and possible solutions, can be found in Ref. [21]. Jacob in Ref. [23] introduces an object-based specification language for direct-manipulation interfaces.

3. YACC, AND DIALOGUE SPECIFICATION

Given a context-free grammar (specification) of some language, YACC is a tool that generates a parser for that language. YACC is an LALR(1)-based parser generator. The generated parser can be used to translate input sentences of the language into some other representation. This translation is specified in the grammar. The parser (or translator) generated by YACC requires a scanner to read the (character) input and convert this input into tokens. A token is a representation of a symbol or construct in the input language. A token may have other information attributed (associated) to it. The parser calls the scanner for each new token. Within the parser, this is referred to as a 'shift' operation. Other operations are 'reduce', 'accept the input' and 'error detected'. In Fig. 2 we show the role of YACC schematically.

The parser contains a push-down automaton, a set of tables, and the semantic actions of the grammar. The tables and actions are created from the specification. The push-down automaton uses the tables and an input token to change to a new state. In the process of changing state, a semantic action can be carried out. An action usually involves emitting a piece of translation code.

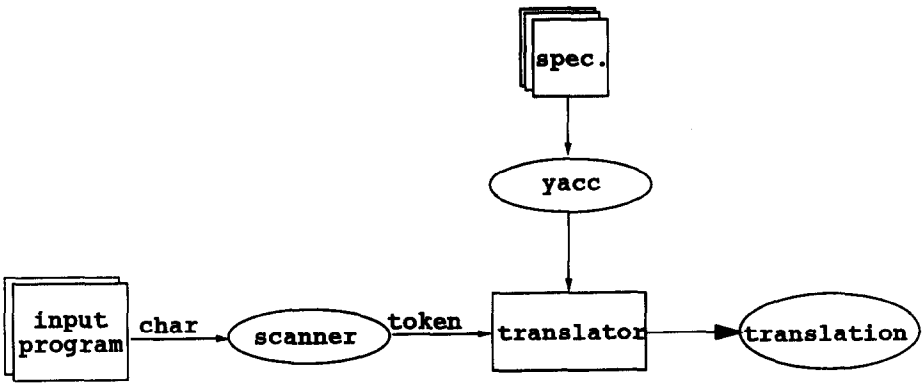

Fig. 2. The YACC compiler model.

The specification that YACC reads is a context-free grammar, with ancillary C-code (the semantic actions). It is quite straightforward to specify the dialogue between the user and an application using a context-free grammar. Note that, in so doing, we are actually specifying the *syntax of an application*. The *semantics of the application*, also referred to as the functionality, is then contained in the ancillary C-code. In Fig. 3, we show the structure of a system that uses an automaton generated by YACC to control dialogue.

Consider, for example, the game of *hangman*. In this game, the computer chooses a random word, and tells the user how many letters are in the word. The user must determine what the word is by guessing letters in the word, or by making a stab at the word itself. After each letter guess, the computer tells the user whether the letter is in the word or not, and if so, where it appears in the word. The aim of the user is to determine the word using as few letter or word guesses as possible.

In Fig. 4 we show the syntax of this game in 'YACC-grammar' form. In this specification, nonterminals begin with an upper-case letter, and tokens are underlined. The rules in the grammar consist of a nonterminal symbol on the left-hand side, and terminal and nonterminal symbols on the right-hand side. The left and right-hand sides are separated by a colon, and a rule is terminated by a semi-colon. Different rules that share the same left-hand side nonterminal are indicated by a vertical bar. We differ slightly from standard YACC notation and denote an 'empty' rule by ϵ.

In essence, the dialogue in *hangman* consists of zero or more letter and word guesses (corresponding to tokens letter and word), optionally terminated by the token giveup. To complete the specification of the game, we must add to the dialogue specification application code in the form of semantic actions. For example, when the user guesses a letter, we must first check whether

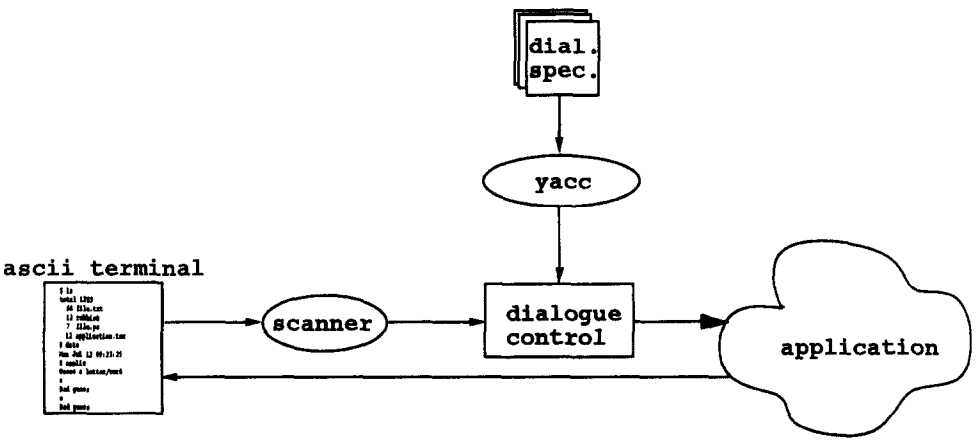

Fig. 3. Using YACC to control dialogue.

```
Hang  :  List
         Fini;
Fini  :  ϵ
      |  giveup;
List  :  ϵ
      |  List LorW;
LorW  :  letter
      |  word;
```

Fig. 4. A part of a dialogue specification of the game *hangman*.

that letter has been tried before, and if not, whether the word contains it. If the word contains the given letter, then we must check whether the word will be complete when we fill the letter in. Code to carry out these and other semantic actions can be added to the grammar.

4. THE X-WINDOW SYSTEM

The X-window system has in recent years become an important standard-platform window system for graphical applications. The system is not only hardware and operating system independent, it is also network transparent. In the X-window system, an application cannot interact directly with windows on the screen. The user communicates with the application by generating events. Pushing a button on the screen, pressing a key, or moving the mouse, for example, all trigger an event. These events can be linked to parts of the application code. When an event is triggered, the corresponding application code is executed, after which control returns to the interface. This is called a *callback*.

The user-interface 'screen' is constructed out of widgets. A widget has characteristics that determine the *look and feel* of the interface. Not all types of widgets are visible on the screen. Those that are visible are generally referred to as windows, and the others act as container widgets for other types of widgets. The user interacts with an application via the widgets in the user-interface, and the keyboard and mouse of course. The semantics (functionality) of the application is largely determined by the arrangement of callbacks and widgets.

Note that since an X-window application is event-driven, and in principle, events can occur at any time, it is up to the application programmer to enable and disable widgets that correspond to parts of the application that may or may not be executed at any given time. Note also that the user is in control—if some event results in the application being called, then after the application has finished, the control is given back to the user.

Widgets can be configured by the application programmer by changing global variables associated with the widgets. These variables are called resources. Resources can be set in application start-up files called *resource specifications*, at widget creation time, or at run-time. Resource specifications can also be used to set the hierarchical structure of the widgets.

An X-window application (see Fig. 5) loops continually waiting for the next event. If an event occurs, then the callback (if any) associated with that event is executed. There is no explicit flow-of-control in the resource specification, only implicit. The hierarchy of widgets, and their creation and deletion, determine the nature of the interaction between the user and application.

Consider, for example, the following user-interface. We define a widget called **test* that consists of a sub-widget containing the string "old message", and another sub-widget containing the string "Quit". The widget **test* is shown in Fig. 6(a). To build this user-interface we simply need to construct a resource specification.

Now let us add the following 'application' to this user-interface. If we click the left-mouse button

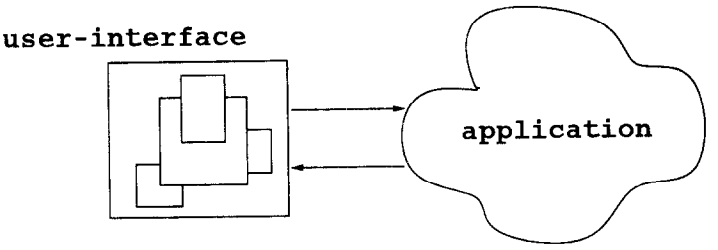

Fig. 5. A schematic of an X-window application.

on the first sub-widget, we want the string to change from "old message" to "new message", and if we click on the second sub-widget, we want the application to terminate (i.e. the widgets should disappear). Actually, the sub-widgets are called (screen) buttons, and the strings are labels. To build such an application we simply attach callbacks (which are also resources) to the respective buttons. To the first button, for example, we attach a callback routine that writes the label "new message" to the calling widget, and to the second, a routine that terminates the application. The result after clicking the first button is shown in Fig. 6(b).

It is not control flow, therefore, but events generated by the user that drive an X-window application. Applications that require frequent interaction with the user are therefore most suited to the X-window system. However, the dialogue between the user and the application is largely unspecified. What is specified is the relationship between the widgets and the callbacks, but by default, the user can perform any action at any time. Some dialogue control can be imposed on an X-window application at the user-interface level, or they can be imposed in the application itself. This is done in the following way.

- In the user-interface, the application programmer can control the execution of callbacks by ensuring that not all buttons (on the screen) are available to the user. Those widgets that correspond to callbacks that may not be executed at a given time must be 'switched off'. The X-window system provides 2 mechanisms to exercise dialogue control. The first is the selective disabling of callbacks, and the second is the general shielding of events outside a particular widget. Selective disabling involves setting a boolean resource. Only if this resource is true will the widget concerned respond to an action from the user. The problem with this is that it is too *fine-grained*—the application programmer must explicitly desensitize those widgets that correspond to callbacks that may not be executed at a given time. The second mechanism, general shielding, effectively desensitizes all but one widget. This mechanism is too *coarse-grained*. Dialogue control in the X-window system, then, is generally *ad hoc*, and inadequate.
- Dialogue control can also be programmed into the application. This is achieved by adding "guards" to callbacks, where a guard is a condition that must be true before the corresponding callback can be executed. The value of a guard is determined by the state of the application. This mechanism is related to the traditional technique of using flags to provide control-flow. It overloads the application, and mixes application code and dialogue control.

Both methods of dialogue control are therefore inadequate.

Fig. 6. The user-interface before and after the message is changed.

5. XYACC

As we saw in Section 3, an application 'generated' by YACC consists of an automaton, which controls the dialogue, and application routines. The automaton waits for input from the user, and calls application routines as directed by the rules of the grammar.

If we now want to use YACC to generate an automaton that interfaces to an X-window-based user-interface, then we must modify YACC. The modifications necessary are to the module in YACC responsible for generating the automaton, and to the module responsible for parsing the grammar. The modifications are the following.

- The automaton must be a co-routine of the user-interface.
- An error in the automaton must not be fatal.
- The user should be able to cancel a part of the dialogue.
- The application must also be able to direct the automaton.
- The user-interface should indicate to the user those widgets that are 'active'.

These modifications of YACC are discussed below. The modified version of YACC is called XYACC.

5.1. Co-routine

The first modification is to the generated automaton. In the traditional model (see Fig. 2), the automaton requests a token from the scanner every time it is ready to 'shift'. The scanner is a 'slave' of the automaton.

In contrast, an automaton that converses with a user-interface needs no scanner. The role of the scanner (namely, converting user actions into tokens) is assumed by the user-interface itself. Further, the user, via the user-interface, is in control—the automaton must wait for each token from the user-interface.

We implement this scheme by making the automaton a co-routine of the user-interface, with the token (representing the user action) as formal parameter. The automaton 'shifts' this token and carries out any reductions that it can. If it can do no more, then the automaton goes into a 'waiting' state. It remains in this state until the automaton is again called. Only when the user terminates the application does the automaton 'accept' the input (and the automaton dies).

5.2. Error handling

YACC is designed to act as a syntax-analysis tool, not as a dialogue generator. A consequence of this is that the error-handling in (an automaton generated by) YACC is inappropriate. In a dialogue, making an incorrect action should not, of course, result in the whole dialogue being rejected. There can be no fatal errors in the automaton.

To keep the dialogue specification manageable, the automaton should either be 'shielded' from invalid actions (tokens) by the user-interface, or, if the user does make an invalid action, then it should be caught and handled by the automaton. In either case, the existing error-handling scheme in YACC can be removed. The concept of valid tokens is discussed in more detail in Section 5.5.

5.3. Cancellation

Cancellation is stopping a process that had been started at a certain point, returning to an earlier stage in the process, and continuing from there. In the case of dialogue, the process is the parsing of a rule. During dialogue design, we try to predict where a user may wish to cancel a part of the dialogue. In the automaton, a part of the dialogue is represented by a nonterminal, so if the user wishes to cancel this part of the dialogue, we must be able to reinstate the state of the automaton to just before the nonterminal was parsed.

Consider, for example, the following grammar S_1.

S_1 : A
  | B;
A : α;
B : β;

The symbols α and β denote some right-hand side consisting of terminals and nonterminals. A valid sentence in the language specified by this grammar must match either nonterminal A or B. If we assume that the sentences that match A and B do not share a common prefix, then once the first character of a sentence has been entered, there is no turning back. In a dialogue this amounts to the user being forced to carry out a certain sequence of actions. This is often unacceptable. To allow the user to abort or cancel a part of a dialogue, we provide a *cancellation* facility in the grammar. To cancel the input means to 'throw away' the input that has been received to date from some starting point. This facility allows, for example, multiple attempts at matching a particular nonterminal to be made. If the strings w_i, for $i = 1, m$, match the nonterminal A or B above, then we require that the string $w_1 \otimes w_2 \otimes \ldots w_{m-1} \otimes w_m$ be accepted by a grammar modified to provide cancellation. Here $w_1, w_2, w_3, \ldots, w_{m-1}$ each represent a series of actions (tokens) that the user has subsequently cancelled. The act of cancellation is represented by the token $\otimes$. The string w_m was not cancelled, and is used to continue the dialogue.

If we now wish to modify S_1 to explicitly allow for cancellation of A or B, then we could use the following grammar.

S_2 : N
  | $N \otimes S_2$;
N : A
  | B;
A : α;
B : β;

Here the first 2 productions allow nonterminal N to be 'cancelled' any number of times. This technique of specifying cancellation can become non-transparent and clumsy in complex dialogue. Instead, we modify the YACC grammar to explicitly indicate where cancellation may occur, and where the subsequent dialogue must restart. We can rewrite the original grammar S_1 using this explicit technique in the following way.

S_3 ! AX
  | BX;
A : α;
B : β;
X : ϵ
  | $\otimes \langle S_3 \rangle$;

The first rule in this grammar is called a *store* rule, this being indicated by the exclamation mark. The last rule is the *reinstate* rule, indicated by the nonterminal enclosed between angular brackets. In the generated automaton, the store rule causes the state of the automaton to be stored (before parsing of the corresponding rule commences). The reinstate rule does the reverse—it forces the automaton to continue parsing with the named (and previously stored) state as its starting state.

As a realistic example of the use of cancellation, consider the situation where a user can carry out 2 tasks, one of which is inputing a name. A grammar that represents this dialogue is the following.

Session ! Task1
  | Task2;
Task1 : ...
Task2 : Getname
  | Task2 Getname;
Getname : readtok
  | canceltok⟨Session⟩;

Let's assume that the user wishes to enter a name, which is *Task*2 in the dialogue. Because the *Session* rule is a store rule, before we begin parsing, the state of the automaton is stored. When the user activates the appropriate widget, and types in a name, the user-interface generates a token readtok. This corresponds to a single read of a name. In *Task*2, the user can enter a name any number of times, until the application is satisfied. If, however, the user is unable to remember the

name, then the user must be able to cancel (abort) this part of the dialogue, and possibly go on with *Task* 1. This is achieved by the reinstate rule that is parsed as a result of receiving the token canceltok. Note that if the application wishes that the user cancels the dialogue (e.g. restrict the number of guesses a user may have at the name) it is also possible for the application to generate this token. We discuss this in the next section.

The store and reinstate rules that are added to the grammar must satisfy the following conditions.

- A reinstate should be the last item of a rule. This should be intuitively obvious—everything after the reinstate will never be executed.
- We can only reinstate a rule that derives the current rule.
- The nonterminal specified in a reinstate is unique because each nonterminal can only appear once on the left-hand side of a rule.
- A reinstate can only be to a stored rule.

In practice, XYACC inserts an extra rule into the grammar when it encounters a store rule. This extra rule has as semantic action a call to a routine that stores the current state. For example, the rule "A ! α;" becomes

```
A : {store(A);}B;
B : α;
```

The function *store(A)* stores the state of the automaton in a structure indexed by its argument (in this case *A*). When a reinstate rule is encountered, a call to a routine that reinstates the stack is inserted in the semantic action of that rule. Of course, while dialogue can be cancelled, semantic actions that are executed during the dialogue cannot be undone. This is the responsibility of the application programmer.

5.4. Application-directed dialogue

We have assumed until now that the tokens that the dialogue receives come only from the user-interface. This need not, however, be the case. By changing the interface to the automaton slightly, we can allow the application to also generate tokens. We do this by defining a global variable *nexttoken* than can be set by the application, and by adding a routine that after having called the automaton, checks to see if a token has been placed in the variable (by the application, during the call). If so, then it re-calls the automaton with this token as parameter, otherwise it simply returns. The new interface loops on the variable *nexttoken* until the application places no more new tokens in this variable.

To illustrate the role the application can play in a dialogue consider the following grammar.

```
S : ϵ {if (init( )) nexttoken = abort;}
  | AB;
A : ϵ
  | α;
B : ϵ
  | abort {printf("unable to init\n");};
```

Here we show for the first time semantic actions in the form of simple C-code. In this dialogue, the function *init*( ) is first called to initialize the application. If it is not able to do so, then the token abort is placed into *nexttoken*. The effect of this is to make the automaton take another step (and process the token abort): the nonterminal *A* will match the empty string, *B* will match the terminal abort, an error message is generated, and the dialogue is finished. If, however, the application was initialized successfully, then the nonterminal *A* will match α, and *B* will match the empty string.

5.5. Highlighting active widgets

In standard X-windows, a widget that has a callback will be highlighted whenever the mouse is positioned on it. But not all widgets, of course, correspond to tokens that are valid (to the automaton). Because we have separated the user-interface and the dialogue, the user-interface, and hence the user, cannot know which widgets are active (correspond to valid tokens), and which are

inactive. This is information that only the dialogue has at its disposal. If all widgets that have a callback are highlighted, then this will mislead the user.

It is true, of course, that if the user activates an invalid callback, then the corresponding token will be ignored by the automaton, but this behaviour does not support the user in any way—his time has been wasted, and he cannot actually be sure that the application did nothing, it just appeared to do nothing. We could remedy this problem by including 'error' rules in the corresponding context-free grammar. These rules 'catch' invalid tokens, and generate error messages. This, however, is a messy and error-prone process, and makes the specification less readable and manageable.

During parsing, the tokens that are valid to the automaton are referred to as the *look-ahead set*. Normally speaking, the look-ahead set is not available to the user. The look-ahead set can be extracted from the automaton and used to indicate which widgets are active. This can be presented to the user in 2 ways.

- If the mouse is positioned on a valid widget, then it is bolded.
- After every user action, invalid widgets (buttons) are automatically shaded out, so the user knows (independent of the position of the mouse) that they are inactive.

The former solution is clearly unacceptable. It requires the user to 'wander' around the screen to determine which (command) widgets are active. The latter solution results in the user-interface refreshing itself after each user action. The user-interface automatically sensitises those widgets that become members of the look-ahead set, and desensitizing those that leave.

The implementation of a scheme to highlight active widgets is a twofold problem.

- First the look-ahead set needs to be extracted. This can be done statically (while the automaton is being generated), or dynamically (during interaction between the user and application). The former solution is the most efficient, but it does have the disadvantage that the look-ahead information must be stored for later use. For this reason, and reasons of transparency, it was decided to extract the look-ahead set dynamically. This involved adding a routine to the generated automaton that simulates all possible parses. That is, the automaton, before returning control to the user after a 'shift' (and possibly some number of reductions), calls a 'dummy' parser for each token. Those tokens that could be successfully parsed by the dummy parser are valid, and are placed in the look-ahead set. Doing this computation for every token generated by the user-interface could be expected to degrade the performance of the system. This was not the case, however. There was no noticeable performance degradation.
- The second problem is making use of the look-ahead set. While the automaton knows what the lookahead-set is, it does not know the names of the widgets corresponding to each of the tokens in the set. To overcome this problem, the user-interface initializes, at start-up time, an array that couples token names and widget names. The call to the initialization routine is simply a callback associated with the root widget.

  Given the look-ahead set, and the array coupling tokens to widgets, the automaton can quite simply sensitize/desensitize each widget by setting/unsetting the appropriate resource.

As an example, consider the following grammar.

```
S    : AorBT;
T    : CorDU;
U    : EorFV;
V    : GorHS;
AorB : atok|btok;
CorD : ctok|dtok;
EorF : etok|ftok;
GorH : gtok|htok;
```

In the dialogue specified here, the user must click on, in succession, an **a** or **b** button, then a **c** or **d**, an **e** or **f**, and finally a **g** or **h**. Clicking on an **a** button generates the token *atok*, clicking on a **b** button generates the token *btok*, and so on. The dialogue cycles endlessly. The appearance

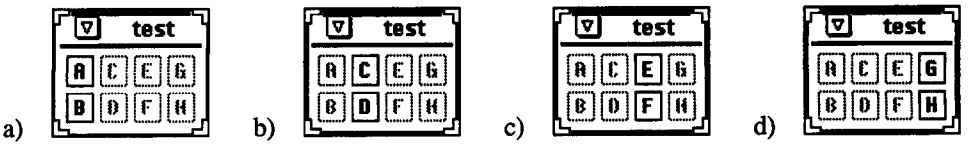

Fig. 7. The user-interfaces resulting from a sequence of user actions.

of the user-interface at each step in the process is shown in Fig. 7. Notice in this figure that only those buttons that are active are highlighted (bolded) at each step, and the others are shaded out.

6. XYACC, AND DIALOGUE SPECIFICATION

XYACC is now able to generate an automaton, which we call a dialogue controller, for an X-window application. In such an application, the user-interface plays the role of a scanner by generating tokens that represent the actions of the user. As these tokens are processed, the dialogue controller calls application routines. A schematic showing the relationship between the user-interface, dialogue controller, and application is shown in Fig. 8.

To demonstrate how this scheme works in practice let us add a dialogue controller to the simple user-interface described in Section 4. To recap: in this user-interface the user can replace the label "old message" by the label "new message" in a button by clicking on that button, and he can terminate the application by clicking on the "Quit" button.

We must first modify the user-interface. Instead of calling the application, the user-interface must now call the dialogue controller for each user action. Each call has a token to represent the action as parameter. This is a minor change to the resource specification of the user-interface.

We now need to define the dialogue. Because the user can do very little, the dialogue specification is also near-trivial. There are, however, some design decisions that need to be made. Should the user, for example, be allowed to click on the first message only once, in which case the dialogue could be specified using the rule

```
S  :  reptok;
```

or should we explicitly cater for clicking on this widget *ad nauseum*, by using a recursive rule like

```
S  :  reptok
   |  S reptok;
```

The latter rule contains much redundancy because second and subsequent clicks on this widget not only do nothing in the application (although one could argue that the user should nevertheless be

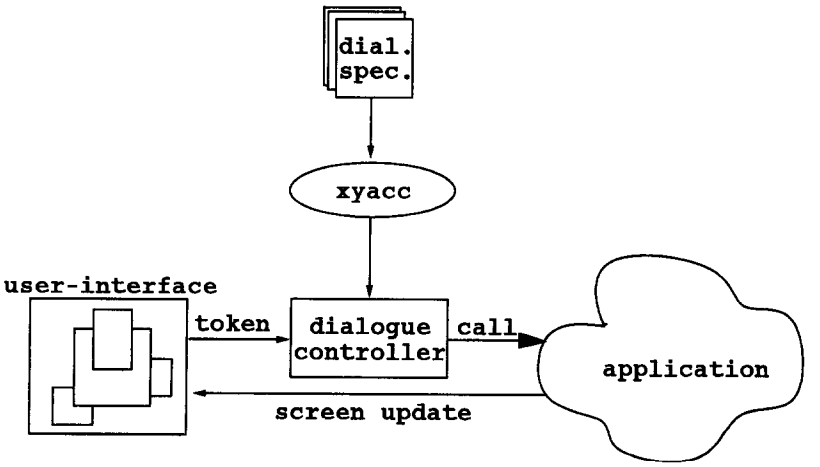

Fig. 8. A schematic of an X-window application with dialogue.

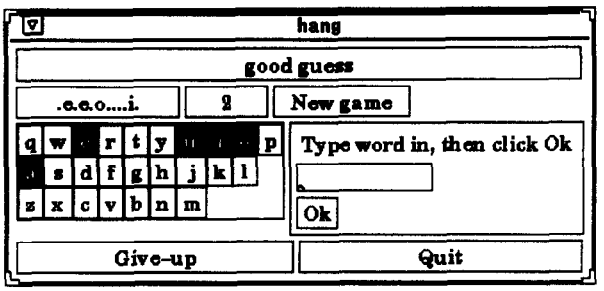

Fig. 9. A snapshot of the user-interface of *xhangman*.

given the freedom to do this), the extraneous tokens that are generated are also ignored by the automaton. We choose the former rule.

To complete the specification we must add calls to the 'application'. This consists of a simple routine *write2widget*( ), which takes 2 parameters: the name of an activated widget, and a (new) string. The effect of this routine (the details are not shown) is to replace the label in the given widget by the string. The full dialogue specification is as follows.

S:reptok{write2widget($1, "new message");};

In this grammar, the attribute associated with reptok, referred to by $1, is the name of the activated widget.

The above example is too trivial to show off the virtue of a dialogue specification. Let's consider a more realistic example, namely the game *hangman*. We considered this game in Section 3, where we specified this game using a context-free grammar. This grammar, in fact, can be used to form the basis of a dialogue specification. Before presenting this, however, let's consider a user-interface to the game, as it appears on the screen. An example of a game in progress is shown in Fig. 9. We will refer to *hangman* with this interface as *xhangman*. The user-interface consists of 8 widgets. From top to bottom, left to right, these are: a message widget (in this case containing the message "good guess"), an incomplete-word widget, a count widget, a button labelled "New game", a widget that consists of buttons for each letter in the alphabet, a dialogue widget (containing an empty widget and a button labelled "Ok"), and 2 buttons labelled "Give-up" and "Quit". At this stage in the game the user has made 5 guesses (the vowels), of which 2 were incorrect (namely 'a' and 'u'). At any time the user can guess a letter, guess the word, give-up (the word will then be revealed), quit, or start a new game. If the user makes a good guess, then a corresponding message appears in the message widget, otherwise the message "bad guess" appears, and the count (widget) is incremented.†

The resource specification of the user-interface to this game defines the layout and presentation of widgets, and associates callbacks to certain widgets (buttons). These callbacks send tokens to the dialogue controller. The labels on the buttons, and the associated tokens, are shown in Table.

Table 1

Button	Token
New Game	newgame
Ok	word
Give-up	giveup
q	letq
w	letw
⋮	⋮
m	letm

Actually, various types of widgets are used in this user-interface. For example, the buttons representing the alphabet are called *toggle widgets*. They are called toggle widgets because they will

†The word, by the way, is *heterocyclic*.

```
Hang    !               {   initialise( cannot );
                            write2widget( *message, "Your word has %d letters", strlen(temp));
                            write2widget( *count, "%d", count);
                            write2widget( *word, "%s", temp);
                        }
            List
            Cancel
            Fini
            Cancel;

Fini    :   cannot      {   write2widget( *message, "Unable to init"); }
        |   giveup      {   write2widget( $1, "The word is %s", word); };

List    :   ϵ
        |   List
            LorW;

LorW    :   Letter      {   if (letcmp($1, word)) {
                                setcmp($1, word, temp);
                                if (strcmp(word, temp))
                                  write2widget( *message, "Good guess" );
                                else
                                  write2widget( *message, "CORRECT!");
                                write2widget( *word, "%s", temp);
                              }
                              else {
                                write2widget( *message, "Bad guess");
                                write2widget( *count, "%d", ++count);
                              }
                            }
                        }
        |   word        {   if (!strcmp( getword( $1 ), word)) {
                              write2widget( *message, "CORRECT!");
                              write2widget( *word, "%s", word);
                            }
                            else {
                              write2widget( *message, "Bad guess");
                              write2widget( *count, "%d", ++count);
                            }
                        };
Cancel  :   ϵ
        |   newgame
            <Hang>;
Letter  :   leta        {   $$ = 'a'; toggle( $1 ); }
        |   ...
```

Fig. 10. A dialogue specification of *xhangman*.

permanently change from white to black when they are activated. (In the game, this tells the user that he has already guessed that letter.) Building a certain amount of 'intelligence' in the user-interface reduces the amount of work that the application and dialogue must do. The cost of this convenience, however, is that the system becomes less portable. (For example, if we only want to replace the user-interface, then we must choose an 'equally intelligent' one.)

The tokens in the above table are used to control the dialogue. The corresponding dialogue specification of this game is shown in Fig. 10. Notice in the specification that explicit cancellation is used, and that the application can direct the dialogue during initialization. If the routine

initialise( ) cannot fetch a random word, then the next token *nexttoken* is set to cannot. A description of the rest of the semantics is shown in the Appendix.

7. CONCLUSIONS

In this work, we have used a language model and a parser generator to build a UIMS. The user-interface dialogue, and application are separated in a clean, uniform way. The user-interface is specified using the X-window system, the dialogue controller is specified using a context-free grammar, and the application consists of a library of routines. The protocol between the user-interface and the dialogue controller consists of tokens, and between the dialogue controller and application, calls to application routines. The resulting UIMS is integrated, and requires a minimum of programming.

The LALR parser generator used in this research, YACC, was able to handle a wide variety of dialogue-specification problems. It is a more powerful generator than the more popular ELL parser generators that have been used by other researchers (for references, see Section 2). In our case, this extra power was necessary because we concentrated on solving dialogue-specification problems syntactically.

The modifications that have been made to YACC have been described. A step-by-step explanation of each modification is given, showing not only how each modification was made, but also why. Because of the simplicity of the system, it was possible to describe the operation of the system in some detail.

The separation between the components in the system does come at some cost. There are times, for example, that the application needs to know to which window to send data. This information must come from the user-interface via the dialogue. Unfortunately, this communication channel is sometimes clumsy and inconvenient. Attributes must be used, but these are limited. As a consequence, much data must be made global, but this blurs the boundaries, and results in dependencies between the components in the system. Unattractive 'fixes' like the initialization routine (described in Section 5.5) that couples widget names and token names are necessary. Although the problem of separate or shared data models for the user-interface and application faces all implementations of the Seeheim Model, this problem is not often addressed in the literature. Exceptions are the recent work of Neelamkavil and Mullarney [24], who built a UIMS called PAPILLON, and Wood and Gray [18], who built CHIMERA. Neelamkavil and Mullarney do not resolve the problem, but Wood and Gray do by adding a special interface, called a linkage component, between the dialogue and application layers of the Seeheim Model.

While the division of labour between the 3 components at an abstract level is clear, an interface builder is often confronted in practice with the problem of whether to place (input/output) facilities in the user-interface, or in the dialogue. This is often a clean design versus convenience issue, which can be greatly influenced by the available resources (the widget set, for example), but it is also a matter of style. A 'clever' presentation component, for example, can significantly reduce the amount of work that the dialogue and application components must do. However, a 'clever' presentation component requires application-specific information, which is inconsistent with the Seeheim Model, and blurs the boundary between the user-interface and the application [25]. There is unfortunately little reference to this issue in the literature. The exception here is Wiecha *et al.* [26], who, as well as defining a dialogue component, split the user-interface into a layer that contains a set of screen primitives and a layer that consists of style rules. This style layer defines the presentation and behaviour of a family of interaction techniques.

The major improvement to this work would be the addition of a dynamic application interface. It is simply not good enough to define the semantics as calls to routines. As well, a clearer separation of concerns and a better data model are necessary. Multi-threading, error handling, a help facility, and a mechanism to handle the continuous sampling of events are also needed.

Acknowledgement—Many thanks to Rick Mud who built an early prototype of the system and first specified the dialogue of *hangman*.

REFERENCES

1. Green, M. Report on Dialogue Specification Tools. In *Proceedings of the Workshop on User Interface Management Systems* (Edited by Pfaff, G. E.), pp. 9–20. Seeheim: Springer-Verlag; 1983.
2. Morse, A. and Reynolds, G. Overcoming current growth limits in UI development. *Commun. ACM* **36**: (April) 73–81; 1993.
3. Johnson, S. C. *YACC—Yet Another Compiler-Compiler*. Murray Hill, NJ: Bell Laboratories, CSTR 32; 1975.
4. Green, M. A survey of three dialogue models. *ACM Trans. Graphics* **5**: (July) 244–275; 1986.
5. Parnas, D. L. On the use of transition diagrams in the design of a user interface for an interactive computer system. *Proc. 24th Nat. ACM Conf.* (1969).
6. Jacob, R. J. K. Using formal specifications in the design of a human–computer interface. *ACM Trans. Graphics* **26**: (April) 259–264; 1983.
7. Wasserman, A. I. Extending state transition diagrams for the specification of human–computer interaction. *IEEE Trans. Softw. Engng* **11**: 699–713; 1985.
8. Roudaud, B., Lavigne, V., Lagneau, O. and Minor, E. SCENARIOO: a new generation UIMS. In *Human–Computer Interaction–INTERACT '90* (Edited by Diaper, D. *et al.*), pp. 607–612. North Holland: Elsevier Science; 1990.
9. Hill, R. D. Supporting concurrency, communication and synchronization in human–computer interaction—the Sassafras UIMS. *ACM Trans. Graphics* **5**: (July) 179–210; 1986.
10. Sibert, J. L., Hurley, W. D. & Bleser, T. W. *Advances in Human–Computer Interaction 2*. Norwood, NJ: Ablex; 1987.
11. Olsen, D. R. Automatic generation of interactive systems. *Computer Graphics* **17**: 53–57; 1983.
12. Olsen, D. R. & Dempsey, E. P. Syntax directed graphical interaction. *ACM SIGPLAN Notices Symp. Progr. Lang. Issues Softw. Syst.* **18**: 112–117; 1983.
13. Olsen, D. R. and Dempsey, E. P. SYNGRAPH: a graphical user interface generator. *Computer Graphics* **17**: 43–50; 1983.
14. Olsen, D. R. Pushdown automaton for user interface management. *ACM Trans. Graphics* **3**: 177–203; 1984.
15. Olsen, D. R., Dempsey, E. P. and Rogge, R. Input/output linkage in a user-interface management system. *Computer Graphics* **19**: 191–197; 1985.
16. Scott, M. I. and Yap, S.-K. A grammar-based approach to the automatic generation of user-interface dialogues. *Proc. CHI'88 Conf.* 1988.
17. van den Bos, J. Abstract interaction tool: a language for user-interface management systems. *ACM Trans. Progr. Lang. Syst.* **10**: 215–247; 1988.
18. Wood, C. A. and Gray, P. D. User interface–application communication in the Chimera UIMS. *Software–Practice Exp.* **22**: 63–84; 1992.
19. Olsen, D. R. MIKE: the menu interaction kontrol environment. *ACM Trans. Graphics* **5**: 318–344; 1986.
20. Shneiderman, B. Direct manipulation: a step beyond programming languages. *IEEE Computer* **16**: 57–69; 1983.
21. Edmonds, E. and Hagiwara, N. An experiment in interactive architectures. In *Human–Computer Interaction—INTERACT '90* (Edited by Diaper, D. *et al.*), pp. 601–606. North Holland: Elsevier Science Publishers; 1990.
22. Hartson, H. R. User-interface management control and communication. *IEEE Software* **6**: 62–70; 1989.
23. Jacob, R. J. K. A specification language for direct-manipulation user-interfaces. *Commun. ACM* **5**: 283–317; 1986.
24. Neelamkavil, F. and Mullarney, O. Separating graphics from application in the design of user interfaces. *Computer J.* **33**: 437–443; 1990.
25. Hurley, W. D. and Sibert, J. L. Modeling user interface-application interactions. *IEEE Software* **6**: 71–77; 1989.
26. Wiecha, C., Bennett, W., Boies, S., Gould, J. and Greene, S. ITS: a tool for rapidly developing interactive applications. *ACM Trans. Inform. Syst.* **8**: 204–236; 1990.

APPENDIX

initialise( ) Places a random word in a string *word*, a series of dots in the string *temp*, and initialises an array *used* to store the letters that have already been guessed. Return success or failure of these activities.

letcmp( ) Check to see if the first parameter, a letter, is in the second parameter, *word*. Return success or failure.

setcmp( ) The first parameter is a letter. For each instance of this letter in the second parameter, *word*, copy the letter to the corresponding position in the third parameter, *temp*.

strcmp( ) If the two string parameters are equal, return success, otherwise return failure.

About the Author—ALBERT NYMEYER received his Bachelor of Mathematics degree in 1974 and Diploma in Computer Science in 1975, both at the University of Newcastle, Australia. After working for a period as a research assistant, he began his Ph.D. studies at the same university in the field of theoretical physics. This research involved computer simulations and series analysis of the critical behaviour of various lattice models. He received his Ph.D. in 1985. In 1987 he took up a lecturing position in computer science at the University of Twente, The Netherlands. He is currently a member of the Tele-Informatics and Open Systems group, and is involved in research into compiler design, software tools supporting the development of formal-specification languages, and user-interface management systems. In 1995 a book of compiler generation will be published in which he is co-author.