Formal specification of the user interface by using parametric context-free grammars*

Nikolaos S. Papaspyrou

National Technical University of Athens
Department of Electrical and Computer Engineering
Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Athens, Greece
nickie@softlab.ntua.gr

SUMMARY

Despite the significant progress and the emergence of proficient tools that have been witnessed recently, the development of user interfaces remains a hard task in the software development process. The use of formal methods for the specification of user interfaces has been proposed as an aid in this task and various formal models have been considered. Grammars were one of the earliest models that were proposed. Several drawbacks hinder the application of ordinary context-free grammars in practice and have led to the adoption of more complex variations. A relatively simple family of grammars, termed "parametric context-free grammars", is proposed in this paper for the formal specification of the user interface, in comparison to ordinary context-free grammars. The proposed grammars can describe interface languages with context-sensitive features, which are typical in user interfaces. Their main advantages lie in their simplicity and ability to produce shorter and more understandable specifications.

KEYWORDS: Formal methods, user interface specification, interface language, parametric context-free grammars.

INTRODUCTION

As software grows in size, the complexity of the software development process grows disproportionally. Contemporary software is orders of magnitude larger and more complex than what it was quite recently and the task of designing and implementing it seems now more and more unfeasible without the aid of carefully planned and rigid collaborative techniques. The use of formal methods improves software development, especially when used in conjunction with other informal and empirical techniques, by reducing errors and misinterpretations in requirements and design.

Although the complexity of the interaction between humans and computers has not increased significantly from the human's point of view, the development of the user interface grows in complexity, required time and effort at almost the same rate as the software system hiding behind it. Many formal specification methods have been proposed to formally express the structure, properties and behaviour of user interfaces and interactive systems. Introductions to the current state of the art can be found in a number of books [3, 5, 8, 10] and conference proceedings [1, 6, 7, 17]. The models that have been used fall largely into three categories: grammars, finite automata and nets. In general, formal methods present the advantage of specifying the user interface precisely and without ambiguity. In this way, they provide a means of ensuring that the implementation of a user interface meets its specification. Furthermore, formal methods offer possibilities for rapid prototyping and automating parts of the implementation of the user interface.

Grammars were one of the earliest proposed models for human-computer interaction. They have traditionally been used for modelling command-oriented languages, however grammars of various types have been suggested by many successful modelling formalisms, including:

- Action language [15, 16], which uses BNF to represent the task-action mapping. Similar approaches using more powerful grammars have been proposed recently [18, 12].
- Task-action grammar (TAG), which enhances rules written in a form similar to BNF with semantic features that can be used to group rules, with respect to their meaning for the user [11, 13].
- Extended tag-action grammar (ETAG), which is an extension of TAG using many levels of abstraction to specify the "user's virtual machine" [19, 20].
- Attribute grammars [14] and graph grammars [4].

The use of ordinary context-free grammars presents three main disadvantages when applied to the formal specification of user interfaces. First, a context-free grammar can only describe the static component of the interaction, i.e. the syntax of the interface language, but not the dynamic part, i.e. its semantics. Second, it is not capable of describ-

*This paper is based on work supported by the National Technical University of Athens, under the Programme for the Support of Basic Research "Archimedes". Project title "SynSemAL: Syntax and Semantics of Artificial Languages".

ing context-sensitive features which are typical of user interfaces. And third, the size and complexity of grammars grows very fast in realistic examples and results in specifications that are difficult to comprehend and manipulate. To overcome these disadvantages, most of the aforementioned formalisms have significantly extended the formal model of grammars by introducing new features, and thus significantly increased its complexity. It seems that research in this field has faced a tradeoff between the complexity of the formal model on the one hand and that of the developed specifications on the other, and has resolved it in favour of the former.

In this paper, we attempt to reverse this approach and propose a relatively simple formalism that we call *parametric context-free grammars*, for the specification of the static component of user interfaces. Such grammars are only a simple and intuitive extension of context-free grammars; however, they are more expressive and capable of describing context-sensitive features of interface languages. They are especially useful in the specification of languages with families of similar features [9]. In the rest of the paper, we first define parametric context-free grammars and then introduce their use for the specification of user interfaces. As an example, we discuss in some detail the user interface of the Notepad application.

PARAMETRIC CONTEXT-FREE GRAMMARS AND PBNF

A natural extension of context-free grammars is obtained by introducing parameters denoting arbitrary strings in nonterminal symbols. A *parametric context-free grammar* (PCFG) is defined as a tuple of the form:

$$G = \langle T, NT, a, I, R, s \rangle$$

where T is a set of terminal symbols, NT is a set of nonterminal symbols, $a : NT \rightarrow \mathbb{N}$ is a function returning the number of parameters that each nonterminal symbol expects, I is a set of identifiers used as names for formal parameters, R is a set of production rules and $s \in NT$ is the initial nonterminal symbol, which must satisfy $a(s) = 0$. The sets T, NT and I must be distinct and all sets must be finite. Each rule $r \in R$ has the form:[1]

$$nt(i_1, \ldots, i_p) ::= \alpha$$

where $nt \in NT$ is a nonterminal symbol, $p = a(nt)$ is the number of parameters that nt expects, identifiers $i_k \in I$ are distinct and $\alpha \in S(\{i_1, \ldots, i_p\})$ where:

$$S(P) = \bigcup_{n=0}^{\infty} S^n(P), \text{ with } S^0(P) = \emptyset \text{ and}$$

$$\begin{aligned} S^{n+1}(P) = (T \cup P \cup \{nt(\alpha_1, \ldots, \alpha_p) \\ \mid nt \in NT \wedge p = a(nt) \wedge \\ \forall i : 1 \leq i \leq p.\ \alpha_i \in S^n(P)\})^* \end{aligned}$$

We should note here that $S(P)$ is the set of strings consisting of: (i) terminal symbols; (ii) formal parameters, i.e. identifiers drawn from the finite set $P \subseteq I$; and (iii) nonterminal symbols followed by the correct number of actual parameters, which are also elements of $S(P)$.

The one-step production relation $\Rightarrow$ for the parametric context-free grammar G is defined on elements of $S(\emptyset)$, that is, strings containing no free formal parameters. This restriction resolves a number of ambiguities that would result from the possible use of the same identifier as a formal parameter in more than one production rules. The production relation is defined as:

$$\begin{aligned} & \forall p \in \mathbb{N}.\ \forall nt \in NT : a(nt) = p. \\ & \forall \chi, \psi, \beta_1, \ldots, \beta_p, \gamma \in S(\emptyset). \\ & \quad \chi\, nt(\beta_1, \ldots, \beta_p)\, \psi \Rightarrow \chi\, \gamma\, \psi \quad \text{iff} \\ & \quad \exists i_1, \ldots, i_p \in I.\ \exists \alpha \in S(\{i_1, \ldots, i_p\}). \\ & \qquad \alpha[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p] = \gamma \quad \text{and} \\ & \qquad (nt(i_1, \ldots, i_p) ::= \alpha) \in R \end{aligned}$$

where $\alpha[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p]$ is the result of the textual substitution of all formal parameters $i_1, \ldots, i_p \in I$ by the actual values $\beta_1, \ldots, \beta_p \in S(\emptyset)$ in string α. Textual substitution is formally defined in Figure 1.

The reflexive and transitive closure of $\Rightarrow$, which is denoted by $\Rightarrow^*$, represents productions in zero or more steps. The language $L(G)$ generated by the context-free grammar G is defined by:

$$L(G) = \{\alpha \in T^* \mid s() \Rightarrow^* \alpha\} \subseteq T^*$$

It is easy to show that PCFGs are more expressible than CFGs. First, notice that every CFG can be trivially transformed to an equivalent PCFG. In addition, consider the PCFG with $T = \{x\}$, $NT = \{s, p\}$, $I = \{i\}$, $a(s) = 0$, $a(p) = 1$, and three production rules: $s() ::= p(x)$, $p(i) ::= i$ and $p(i) ::= p(i\, i)$. It is not hard to prove that this PCFG generates $L = \{x^{2^n} \mid n \in \mathbb{N}\}$, i.e. the language of strings over T with length equal to a power of 2. We know L is not context-free, therefore CFG $\subset$ PCFG.

In the spirit of the Backus-Naur Form (BNF) and its variations for CFGs, it is convenient to define a formalism for the representation of PCFGs, which we call Parametric Backus-Naur Form (PBNF). This formalism is based on EBNF, with the additional feature that parameters of nonterminal symbols are written inside curly braces ("{" and "}"), in order to distinguish them from grouping parentheses. The braces are omitted altogether if a nonterminal symbol expects no parameters.

USER INTERFACE SPECIFICATION

Parametric context-free grammars can be used for the formal specification of user interfaces. Let us first define a

[1] Throughout this paper, we use the notation $\alpha\,\beta$ for the concatenation of strings α and β, ϵ for the empty string, $L_1\, L_2$ for the concatenation of languages L_1 and L_2 defined as $L_1\, L_2 = \{\alpha\,\beta \mid \alpha \in L_1 \wedge \beta \in L_2\}$, L^n for the concatenation of L with itself n times, defined as $L^{n+1} = L\, L_n$ with $L^0 = \{\epsilon\}$, and L^* for the Kleene star of L defined as $L^* = \bigcup_{n=0}^{\infty} L^n$. We use small Greek letters ($\alpha, \beta, \chi, \psi$) to denote strings.

$$
\begin{array}{l}
\forall p \in \mathbb{N}.\, \forall i_1, \ldots, i_p \in I.\, \forall \beta_1, \ldots, \beta_p \in S(\emptyset).\, \forall t \in T. \\
\quad t[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p] = t \\
\forall p, k \in \mathbb{N} : 1 \leq k \leq p.\, \forall i_1, \ldots, i_p \in I.\, \forall \beta_1, \ldots, \beta_p \in S(\emptyset). \\
\quad i_k[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p] = \beta_k \\
\forall p, q \in \mathbb{N}.\, \forall i_1, \ldots, i_p \in I.\, \forall \beta_1, \ldots, \beta_p \in S(\emptyset).\, \forall nt \in NT : a(nt) = q.\, \forall \alpha_1, \ldots, \alpha_q \in S(\{i_1, \ldots, i_p\}). \\
\quad nt(\alpha_1, \ldots, \alpha_q)[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p] = nt(\alpha_1[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p], \ldots, \alpha_q[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p]) \\
\forall p \in \mathbb{N}.\, \forall i_1, \ldots, i_p \in I.\, \forall \beta_1, \ldots, \beta_p \in S(\emptyset).\, \forall \alpha_1, \alpha_2 \in S(\{i_1, \ldots, i_p\}). \\
\quad (\alpha_1\, \alpha_2)[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p] = \alpha_1[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p]\, \alpha_2[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p]
\end{array}
$$

Figure 1: Definition of textual substitution.

simple model of human-computer interaction, which nevertheless is expressive enough to cover the typical personal computer software application. There are two agents that participate in the model: the *user* and the *computer*, or more precisely the software that is running on the computer. The two agents communicate by means of generating *events*, i.e. atomic units of communication that are perceived by both.[2] Two types of events are distinguished: *input events*, generated by the user and addressed to the computer, and *output events* are generated by the computer and addressed to the user. Both types of events can be viewed at various levels of abstraction. At a relatively low level, examples of input events in a typical graphical user interface are the pressing of a key or the movement of the mouse, whereas an example of output event is the drawing of a rectangle on the screen. At a much higher level, a more abstract input event is the selection of command "Statistics" from a menu and a more abstract output event is the displaying of a window containing the requested statistical results.

The discrete nature of events should not be considered as a severe limitation of the model. Continuous operations, such as mouse movement or drag-and-drop, can be modelled by sequences of successive events. This technique is widely used in common practice and has been also followed in the implementation of popular windowing systems, like X-Windows or Microsoft Windows.

Syntactically, the *interface language* is defined as the set of all legal sequences of events communicated between the user and the computer during a complete operation of the software. A similar concept for the interface language is defined in the Fusion Method [2] under the term "life cycle model". The adjective "legal" in the previous definition explicitly excludes sequences of events that cannot be obtained; e.g. in most software applications it is not possible that the output event of printing the current document precedes the input event of requesting such a printout.

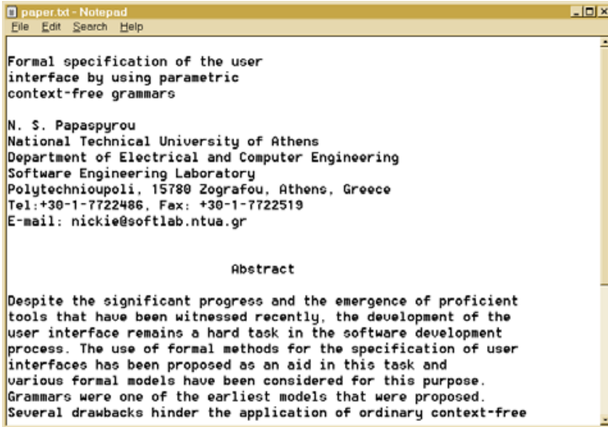
In the study of artificial languages, researchers typically distinguish the notions of syntax and semantics. *Syntax* is the static part of the language and specifies the set of legal phrases. On the other hand, *semantics* is the dynamic part of the language and specifies the meaning of legal phrases. In the case of user interfaces the separation of these two notions is not so easy. In order to accurately specify the interface language, detailed knowledge of the software application's execution behaviour is often required. In this paper, we are only interested in the formal specification of the static part, that is, the syntax of the interface language. The complete specification of interactive systems is generally a very hard task and beyond the aims of our approach. In the attempt to disconnect the syntax of the interface language from its semantics, one is often forced to make simplifications that introduce deviations from the actual syntax. Such deviations are inevitable and should not be considered harmful.[3] In fact, the resulting formal specification corresponds to a superset of the interface language and is a valuable tool in the study and development of the user interface.

Since the interface language consists of strings of events, the parametric context-free grammar used for its specification should have events as terminal symbols. The nonterminal symbols and all other characteristics of the grammar should be appropriately chosen by software analysts. Production rules define how the strings of the language are formed and therefore determine the static part of the user interface. The PBNF notation can be used for representing the grammar and a reasonable option is to prefix all output events by "#", in order to easily distinguish them from input events.

Depending on the needs of the software application, the PBNF notation may be extended by additional operators if necessary, in a way similar to proposed extensions for ordinary context-free grammars. For example, operator "‖" allowing the arbitrary interleaving of strings of terminal symbols would be useful in an application featuring multiple independent human-computer interactions occurring in parallel.

[2] A simple multi-agent model could be defined by replacing events with *messages*, exchanged between agents. The use of parametric context-free grammars and the results of this paper apply to the multi-agent model as well.

[3] As a direct analogue from the study of programming languages, the use of a grammar alone seldom leads to an accurate specification of a language's syntax. Additional semantic restrictions need to be imposed for this purpose, e.g. type checking, and these restrictions cannot generally be included in the formal specification of the syntax.


paper.txt - Notepad
File Edit Search Help

Formal specification of the user
interface by using parametric
context-free grammars

N. S. Papaspyrou
National Technical University of Athens
Department of Electrical and Computer Engineering
Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Athens, Greece
Tel:+30-1-7722486, Fax: +30-1-7722519
E-mail: nickie@softlab.ntua.gr

Abstract

Despite the significant progress and the emergence of proficient
tools that have been witnessed recently, the development of the
user interface remains a hard task in the software development
process. The use of formal methods for the specification of user
interfaces has been proposed as an aid in this task and
various formal models have been considered for this purpose.
Grammars were one of the earliest models that were proposed.
Several drawbacks hinder the application of ordinary context-free

Figure 2: Snapshot from the Notepad application.

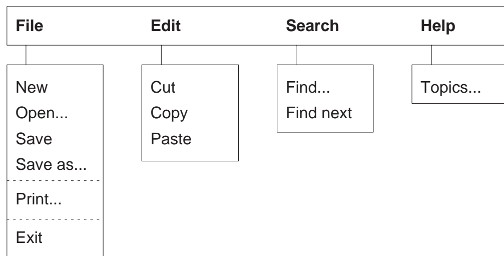


Figure 3: Menu structure for the Notepad application.

EXAMPLE: THE NOTEPAD APPLICATION

In order to illustrate the use of PCFG for the formal specification of user interfaces, we take as an example a simple text editor inspired from the Notepad application that comes with Microsoft Windows. Similar applications exist in all contemporary windowing operating systems. A snapshot from the Notepad application is shown in Figure 2. Its user interface is relatively simple. User input is given through direct typing or through the menu in the upper part of the window, and the computer's output prompts are mostly shown in modal dialog boxes. The structure of the command menu is shown in Figure 3.

In this section, we are primarily interested in demonstrating the use of the proposed specification method and, for this reason, we have chosen a high-level re-engineering approach. We attempt a formal specification of Notepad's user interface at a rather high level of abstraction and focus our attention to the interaction that is related to document management, e.g. creating a new file, saving it, etc. The complete grammar in PBNF is shown in Figure 4. We take as input events all commands that can be issued from the menu, e.g. **new** and **open**, user responses to dialog boxes, e.g. **select-printer** and **cancel**, and the special event **type** which represents direct typing from the keyboard into the current document. As output events we take the dialog boxes that are presented to the user, e.g. **#open-dialog**. Output events denote the opening of dialog boxes; their closing is not shown here.

The interface language is complicated for two reasons. First, it is important whether the current document has been changed since the last time it was saved. If there are no recent changes, commands such as **new** or **exit** can be executed immediately and the **save** command must be unavailable. Otherwise users must be given the option to save their changes before the current document is closed. Second, it is important whether the current document has been associated with a file: the **save** command requires that a file name has already been given. The current document may be "saved" or "not-saved" with respect to the first parameter, and "titled" or "untitled" with respect to the second. Thus, there are four possible states, resulting from the combination of these two parameters and modelled by four nonterminal symbols with no parameters.

With all this in mind, it is useful to distinguish the input commands in two categories: those that produce changes in the current document and those that do not. It is therefore reasonable to introduce two nonterminal symbols in the grammar for this grouping. Finally, one nonterminal symbol is introduced for each dialog box and models the exchange of events from the moment it opens until the moment it closes. Dialog boxes that may affect the state of the current document in more than one way, depending on user input, are associated with parametric nonterminal symbols. For example, the dialog box used for opening an existing document allows the user to respond with **select-file-name** or **cancel**, and this choice affects the current document's state. This dialog is associated with <open-dialog>$\{\alpha, \beta\}$, with parameter α representing the resulting state if the user selects a file name and parameter β the resulting state if the user chooses to cancel the dialog.[4]

In the example that was illustrated in this section, it seems that the PBNF grammar serves as a representation for a state transition system, which could also be represented by a regular (type 3) grammar. Indeed, in the case of the Notepad application it is possible to produce a state transition diagram equivalent to the suggested PBNF grammar. Such a diagram is shown in Figure 5, using a lot of notational conventions in order to keep its size manageable. It should be noted however that the regular grammar corresponding to this diagram contains one production rule for each arrow, after the notational conventions are taken out. An equivalent BNF grammar would certainly consist of less production rules, but still would not be as concise as the PBNF grammar given in Figure 4. As a last remark, the parametric nonterminal symbols that were used in the PBNF specification correspond directly to recurring patterns in the state transition diagram, as shown in Figure 5.

[4]It is interesting to notice the resemblance between parameters used in this way and the continuation passing style that is often encountered in the theory and practice of programming languages.

<untitled-saved> ::= <change> <untitled-not-saved> | <do-not-change> <untitled-saved> | **new** <untitled-saved>
| **open** <open-dialog>{<titled-saved>, <untitled-saved>} | **save-as** <save-as-dialog>{<titled-saved>, <untitled-saved>} | **exit**

<untitled-not-saved> ::= (<change> | <do-not-change>) <untitled-not-saved>
| **new** <not-saved-dialog>{<untitled-saved>, <untitled-not-saved>}
| **open** <open-dialog>{<not-saved-dialog>{<titled-saved>, <untitled-not-saved>}, <untitled-not-saved>}
| **save-as** <save-as-dialog>{<titled-saved>, <untitled-not-saved>} | **exit** <not-saved-dialog>{ϵ, <untitled-not-saved>}

<titled-saved> ::= <change> <titled-not-saved> | <do-not-change> <titled-saved> | **new** <untitled-saved>
| **open** <open-dialog>{<titled-saved>, <titled-saved>} | **save-as** <save-as-dialog>{<titled-saved>, <titled-saved>} | **exit**

<titled-not-saved> ::= (<change> | <do-not-change>) <titled-not-saved>
| **new** <not-saved-dialog>{<untitled-saved>, <titled-not-saved>}
| **open** <open-dialog>{<not-saved-dialog>{<titled-saved>, <titled-not-saved>}, <titled-not-saved>} | **save** <titled-saved>
| **save-as** <save-as-dialog>{<titled-saved>, <titled-not-saved>} | **exit** <not-saved-dialog>{ϵ, <titled-not-saved>}

<change> ::= **cut** | **paste** | **type**
<do-not-change> ::= **search-first** <search-dialog> | **search-next** | **copy** | **help** <help-dialog> | **print** <print-dialog>

<open-dialog>{α, β} ::= **#open-dialog** (**select-file-name** α | **cancel** β)
<save-as-dialog>{α, β} ::= **#save-as-dialog** (**select-file-name** α | **cancel** β)
<not-saved-dialog>{α, β} ::= **#not-saved-dialog** (**yes** α | **no** α | **cancel** β)
<search-dialog> ::= **#search-dialog** (**search-text** | **cancel**)
<help-dialog> ::= **#help-dialog close-help**
<print-dialog> ::= **#print-dialog** (**select-printer** | **cancel**)

Figure 4: Specifications for the user interface of the Notepad application.

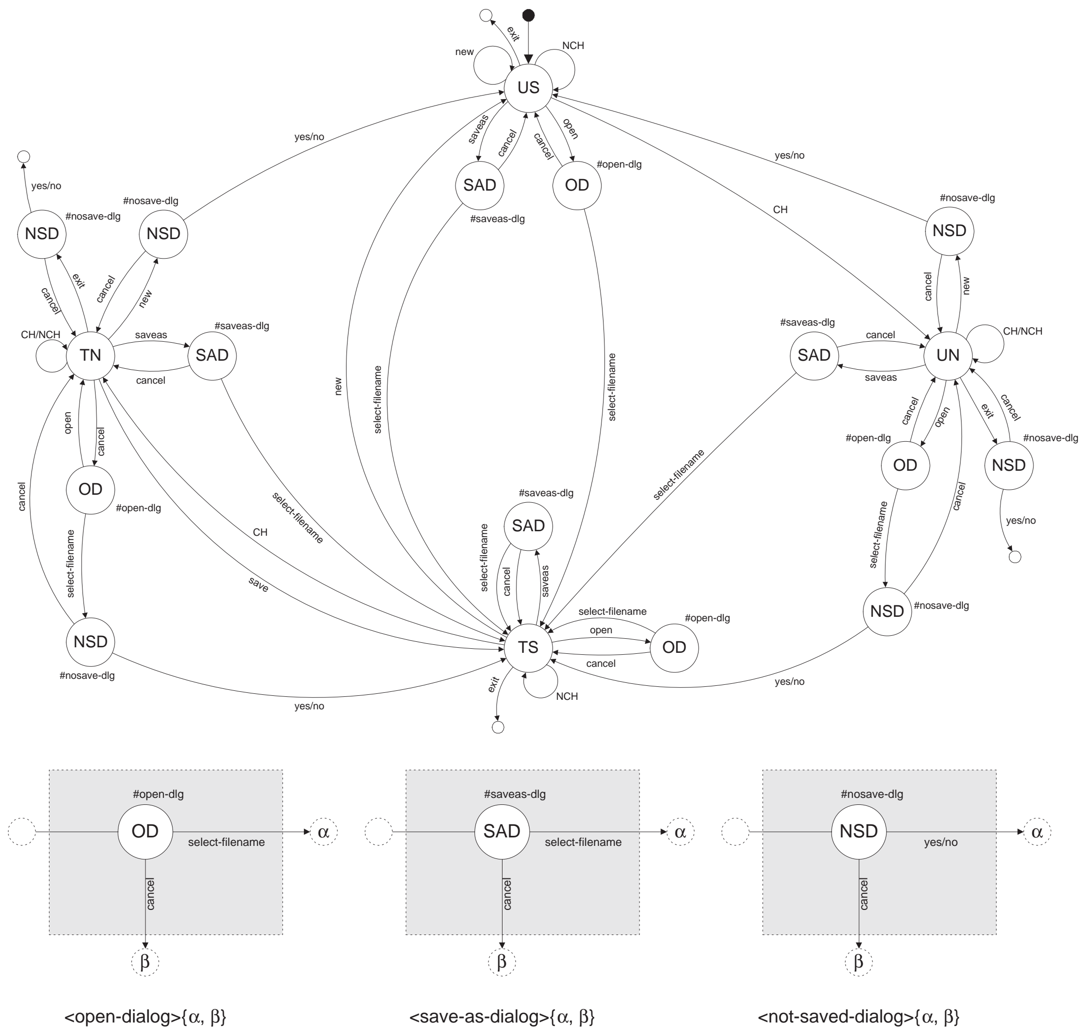


Figure 5: State transition diagram and recurring patterns for the Notepad application.

CONCLUDING REMARKS

In this paper we have proposed the use of a new kind of grammars, which we call parametric context-free grammars (PCFG), for the formal specification of user interfaces. This family of grammars is adequate for describing the static component of the user interface for a large category of software applications. It combines the simplicity of ordinary context-free grammars with expressive power and context sensitivity that is encountered in significantly more complex formal models.

The drawbacks and limitations of the suggested approach, both from a theoretical and a practical point of view, have yet to be determined through future experimentation with large-scale software systems. However, it seems that this approach deserves the effort to be applied to the software development process and this should be the main direction for future research. Although only the syntax of a software system's interface language can be specified by parametric context-free grammars, an interesting direction for future research would be towards the formal specification of the language's semantics, based on its syntactic structure. Techniques and formalisms widely used in the study of programming languages, such as operational or denotational semantics, can be used for this purpose. The ambitious aim of this attempt would be a complete formal specification of a software system, which would be based on its interface language.

BIBLIOGRAPHY

1. F. Bodart and J. Vanderdonckt, editors. *Proceedings of the Eurographics workshop on design, specification and verification of interactive systems*, Namur, 1996. Springer Verlag.
2. D. Coleman et al. *Object-oriented development: the Fusion method*. Prentice Hall, Englewood Cliffs, NJ, 1994.
3. A. Dix. *Formal methods in interactive systems*. Academic Press, 1991.
4. M. Goedicke and B.E. Sucrow. Towards a formal specification method for graphical user interfaces with modularized graph grammars. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 56–65, 1996.
5. M. Harrison and H. Thimbleby. *Formal methods in human-computer interaction*. Cambridge University Press, 1990.
6. M. Harrison and J. Torres, editors. *Proceedings of the Eurographics workshop on design, specification and verification of interactive systems*. Springer Verlag, 1997.
7. P. Palanque and R. Bastide, editors. *Proceedings of the Eurographics workshop on design, specification and verification of interactive systems*, Toulouse, 1995. Springer Verlag.
8. P. Palanque and F. Paternò. *Formal methods in human-computer interaction*. Formal approaches to Computing and Information Technology Series. Springer Verlag, London, UK, 1998.
9. N.S. Papaspyrou and V.C. Vescoukis. Facilitating the definition of programming languages by using parametric context-free grammars. In *Proceedings of the 7th Hellenic Conference on Informatics*, volume II, pages 91–98, Ioannina, Greece, August 1999.
10. F. Paternò. *Interactive systems: design, specification and verification*. Focus on Computer Graphics Series. Springer Verlag, Heidelberg, Germany, 1995.
11. S.J. Payne. Task-action grammar. In B. Shackel, editor, *Proceedings of Interact'84*, pages 139–144, 1984.
12. S.J. Payne and T.R.G. Green. The user's perception of the interaction language: a two-level model. In *Proceedings of CHI'83*, pages 202–206, 1983.
13. S.J. Payne and T.R.G. Green. Task-action grammars: a model of the mental representation of task languages. *Human-Computer Interaction*, 2:93–133, 1986.
14. Hua Qingui. An approach to user interface specification with attribute grammars. *Journal of Computer Science and Technology*, 12(1):65–75, 1997.
15. P. Reisner. Formal grammar and human factors design of an interactive graphics system. *IEEE Transactions on Software Engineering*, 7:229–240, 1981.
16. P. Reisner. Formal grammar as a tool for analyzing ease of use: some fundamental concepts. In J.C. Thomas and M.L. Schneider, editors, *Human Factors in Computer Systems*, pages 53–78. Ablex Publishing Co., Norwood, NJ, 1984.
17. C. Roast and J. Siddiqui, editors. *Proceedings of the workshop on formal aspects of the human-computer interaction*. Springer Verlag, 1997.
18. B. Shneiderman. Multiparty grammars and related features for defining interactive systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-12(2):93–113, 1982.
19. M.J. Tauber. On mental models and the user interface. In G.C. van der Veer, T.R.G. Green, J.M. Hoc, and D.M. Murray, editors, *Working with computers, theory versus outcome*, pages 89–119. Academic Press, London, UK, 1988.
20. M.J. Tauber. ETAG: extended tag-action grammar: a language for the description of the user's task language. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *Proceedings of Interact'90*, pages 163–168, 1990.