

A Scalable Formal Method for Design and Automatic Checking of User Interfaces^{*}

JEAN BERSTEL

INSTITUT GASPARD-MONGE, UNIVERSITÉ DE MARNE-LA-VALLÉE

STEFANO CRESPI REGHIZZI

DIPARTIMENTO DI ELETTRONICA E INFORMAZIONE, POLITECNICO DI MILANO

GILLES ROUSSEL

INSTITUT GASPARD-MONGE, UNIVERSITÉ DE MARNE-LA-VALLÉE

PIERLUIGI SAN PIETRO

DIPARTIMENTO DI ELETTRONICA E INFORMAZIONE, POLITECNICO DI MILANO.

ABSTRACT: The paper addresses the formal specification, design and implementation of the behavioral component of graphical user interfaces. The complex sequences of visual events and actions that constitute dialogs are specified by means of modular, communicating grammars called VEG (Visual Event Grammars), which extend traditional BNF grammars to make them more convenient to model dialogs.

A VEG specification is independent of the actual layout of the GUI, but it can easily be integrated with various layout design toolkits. Moreover, a VEG specification may be verified with the model checker SPIN, in order to test consistency and correctness, to detect deadlocks and unreachable states, and also to generate test cases for validation purposes.

Efficient code is automatically generated by the VEG toolkit, based on compiler technology. Realistic applications have been specified, verified and implemented, like a Notepad-style editor, a graph construction library and a large real application to medical software. It is also argued that VEG can be used to specify and test voice interfaces and multi-modal dialogs. The major contribution of our work is blending together a set of features coming from GUI design, compilers, software engineering and formal verification. Even though we do not claim novelty in each of the techniques adopted for VEG, they have been united into a toolkit supporting all GUI design phases, i.e., specification, design, verification and validation, linking to applications and coding.

1 INTRODUCTION

Current industrial practice for designing graphical user interfaces (GUI) uses toolkits and interface builders, mostly based on visual programming languages, for producing the layout. These tools allow a simple and quick description of the geometric display, and may even give some support for designing interaction of components (e.g., QTDesigner by Trolltech, JavaBeanBox in Sun's BDK, Visual C++ or Glade). However, the dialog control must be hand-coded with conventional programming techniques and there is no support for checking the dynamic behavior of the interface other than testing.

This situation is unsatisfactory at best, since the resulting systems may be unreliable and difficult to revise and extend. In particular, the reactive nature of event-driven systems (such as a GUI) makes them much more difficult to test, since the output values strongly depend on the interaction that may occur during the computation. Traditional testing techniques may be costly and inadequate for complex GUI, yet the importance of verification and validation cannot be underestimated, since the majority of software applications in any domain have a complex GUI. The current inadequacy of existing techniques for GUI design and verification is particularly felt for safety-critical software, where, disregarding also the important ergonomic needs pointed out by Gray et al. [1999], rich

^{*} A preliminary version of this paper appeared in ICSE 2001, Int. Conference on Software Engineering, Toronto, Ontario, Canada, May 12-19, 2001. , IEEE Press, New York , 453-462. This research was partially supported by Esprit Open LTR

and potentially useful user interfaces may be discarded in favor of primitive interfaces that are easier to test and more dependable.

Formal techniques may allow one to perform systematically, or even automatically, validation and verification activities like testing, simulation and model checking [Clarke et al. 1986, Holzmann 1997], and to prove that the modeled systems possess desired properties. Hence, the validity of the design may be assessed ahead of development. Formality is widely acknowledged to support the construction of more reliable software, and hence is becoming more popular in critical areas. Many formal methods have been proposed for GUI design [Palanque et al. [1997], such as (augmented) transition diagrams in Hendricksen [1989], Petri nets in Bastide and Palanque [1995], formal grammars in Reisner [1981], process algebras in Paternò et al. [1992], and temporal logic in Brun [1997]. However, general formal methods are considered difficult to use by most engineers, and often get unwieldy as the system complexity grows (i.e., they are beneficial only for small systems or single components). Also, the methods proposed to give complete support to GUI design so far are not amenable to automatic verification techniques, supporting mainly simulation and testing. As pointed out by Shneiderman, [1997] (p. 159): “Scalable formal methods and automatic checking of user interface features would be a major contribution”.

We propose a new simple formal method, which combines various features such as *modularity*, *code generation* and *automatic verification*, to give a scalable notation to specify, design, validate and verify GUIs. Our approach, called Visual Event Grammars (VEG) is based on decomposing the specification of a large GUI into communicating automata. Breaking a complex scene down into communicating pieces may dramatically diminish the number of states, as shown by popular notations such as Statecharts [Harel 1987]. Each automaton is an object, described by means of a grammar, specifying a small part of the control of the scene, such as the behavior of a window or a widget. Automata may share common behavior and hence be seen as instances of general models. Automata interact by sending and receiving communication events in order to realize the expected global behavior. Data values, such as the actual contents of a text field or the color of a widget, may be associated with events and states, and Java code can manipulate them, following an attribute grammar style [Knuth 1968] of computation. The VEG approach allows the automatic generation of efficient code from the specification of the interface, and its integration with commercial design tools, by implementing the various automata as interacting parsers (where the input stream of each parser is the sequence of input events for the corresponding window or widget). A toolkit has been prototyped, to produce Java classes that implement the logical behavior of the GUI and to integrate them with the layout. Up to now, the toolkit handles interfaces of WIMP (Window, Icon, Menu, Pointer) style. Extensions to more sophisticated interaction paradigms like virtual reality or voice interactions are possible, and discussed in Section 7.

The VEG approach also allows clean separation both of data vs. control and layout vs. behavior. More precisely, separation of data and control means that every VEG specification is composed of a “purely syntactical” part, describing the event-driven behavior of a GUI (the control), and of a “semantical” part, implementing data manipulation. In fact, as already remarked, a VEG specification is akin to an attribute grammar, where the syntax aspects are complemented by semantic functions. The VEG approach thus follows the classical Model-View-Controller architecture pioneered by SmallTalk 80: the controller is specified with the syntax, and it interacts with the model and the view by means of input events and attributes. In particular, the specification and design of a GUI

is independent of its actual layout. This allows portability and ease of modification, but also the reuse of the same logic in implementing different interfaces for the same service (such as in the Sisl approach of Ball et al. [2000]), possibly with different data manipulation. Web services are a typical application that may benefit from this separation. For instance, the Web site of Politecnico di Milano has been redone a few times, mostly because the layout was unsatisfactory: each time the same services had to be partly reprogrammed, while with VEG this would have been unnecessary. Also banking systems may benefit from the layout/logic separation: they offer multichannel access through an automated teller machine, a web-based interface or bank-by-phone interface [Godefroid et al. 2000]. Usually, code duplication occurs, with the associated rise in development and maintenance costs, but this can be avoided with the present approach.

The separation of data and control, together with the independence of the layout, is convenient not only for maintenance and reuse, but also for fast prototyping. GUI prototyping is often necessary for the customer to understand whether the GUI is the right one. The layout may be very changeable in this phase. With VEG, we can quickly specify the logic, choose a layout, give it to the user of the application and verify what has to be changed. The layout may be thrown away afterwards, but VEG specifications remain. Data processing is not needed at this moment: only the control part of the GUI is programmed. Usually, the additional cost of rapid prototyping is that an expensive prototype must be thrown away. With VEG, the initial prototype may be a less costly VEG specification that can often be reused and extended into the final system.

Apart from the merits of individual notations, one of the major obstacles in the diffusion of formal methods outside academic research is the perceived difficulty in their use. Formal specification languages are considered hard to master, and formal verification techniques, such as theorem proving, to be for real experts only. On the other hand, automata, such as those used in VEG, are among the easiest formalisms, and every software engineer is familiar with them. Moreover, advances in automatic verification toolkits, such as model checking, are greatly simplifying formal verification, since in principle "pushbutton" verification is possible for many systems specified with automata.

In general, however, a specification or a program has to be "abstracted" to be amenable for automatic verification with model checkers. In fact, the number of states of even simple programs is typically too high for model checkers, especially because of the large range of data values. Constructing a powerful enough, sound abstraction may require considerable ingenuity, since the abstracted program must be a compromise between efficiency (i.e., the state space for concrete model checkers such as SPIN [Holzmann 1997], or the size of formulae encoding the system for symbolic model checkers such as SMV [Clarke et al. 1986]) and effectiveness (i.e., the meaningfulness of the verification results on the abstracted program). In particular, the abstraction must be *safe* with respect to the properties of interest (e.g., deadlock freedom): if the property holds in the abstracted system then it holds in the original program. A lot of research efforts are currently underway to allow the development of safe abstractions for programming languages such as C and Java, e.g. [Ball et al. 2000, Corbett et al. 2000], but the results still seem hard to generalize and use. In the VEG toolkit, however, an abstraction that is safe with respect to many important properties (such as deadlock-freedom and state invariants) can automatically be derived from the original specification, with a meaning that is very close to the original one. Typically, in VEG the abstracted version is the control (finite-state) part of the specification, while data and related functions are ignored. When

dealing with abstracted programs, however, even when an abstraction is shown or is known to be safe, *spurious counterexamples* may occur, i.e., errors in the abstracted program that do not correspond to feasible computations on the original program. In this case, one may either accept the result (and then try to fix a program that is already correct) or use some form of proof or symbolic execution of the counterexample on the original program, to understand whether the positive is a true one. In our experience with VEG specifications, the problem of spurious counterexamples does not seem to hamper verification as much as in software model checking. An explanation for this is that the notation and the methodology used tend to enforce a clear separation between control and data, while in traditional programs, where this distinction is usually not present, the control flow graph is a very poor abstraction. Moreover, since GUIs are usually event-driven, control-intensive systems, often the abstracted VEG specification is very close to the one used to produce the application: there is a high level of coherence between the application and its formal model.

We based verification and validation on SPIN, a widely disseminated model checking tool. Communicating automata fit particularly well into the domain of automatic verification: the VEG notation can be easily translated into the Promela language, which is the input language of SPIN. Currently, our toolkit supports, with simple "pushbutton" options, automatic detection of design errors such as deadlock and unreachable states, but it also allows state invariant verification, simulation and test case generation. The VEG support for verification may also help in checking features of an interface and in detecting requirement errors. For instance, a Save button in an Editor application should be reachable from every state where the document has been modified. This means that the GUI will never run into a configuration where a user will no longer be allowed to save her data. This is a liveness property, which can be easily verified by a model checker. Another example is the verification that all needed resources are available before a process can start: in a text editor, a document must be created or opened before writing into it. We found that even very large GUI can easily be checked, since usually the number of its (control) states is much smaller than the current limits of model checkers.

Our work draws on a long, well-established tradition of user interface design, called dialog independence (Hartson and Hix [1989], Hill [1986]) or syntax-semantics dichotomy (Foley [1987], Foley et al. [1989], Jacob [1982], Olsen [1983,1984], Reisner [1981], Shneiderman [1982] and others), and in particular on the Seeheim model of Greene [1983]. The goal of these models is the separation of the application from its user interface. Some of these works also applied context-free or regular grammars to the description of dialogs (already in 1981). The reason is that grammars have various advantages over other approaches. For instance, the terminal alphabet of a grammar is usually composed of high-level events at the application level, such as Start, Quit, Cut, etc., allowing platform-independence and often also widget-independence. Also, some authors introduced a special notation to supplement grammars whenever they seem unsuited to describe some features. For instance, Van den Boss [1988] has proposed and developed a special rich notation, exceeding the power of context-free grammars. For this and similar approaches, however, the possibility of automatic verification of the specification becomes quite small, since the more powerful the model is, the more reduced the automatic verification activities may be. VEG does not suffer of this problem, since most features extending traditional grammars are still finite-state, and the others (e.g., attributes) can usually be abstracted away during the verification phase. Also, in VEG special care was taken in order to introduce a small number of features that are really useful in designing GUI, while avoiding baroque notations often used in other approaches. Grammar approaches disappeared from the scenes of GUI design, since they do not have

constructs that are present instead in notations such as the above-cited Petri nets and Statecharts: they did not deal with parallelism, lacked structuring constructs and synchronization or communication mechanisms, and failed in cleanly integrating the data structure aspects of GUI into their control structure aspects. However, the VEG constructs support parallelism and structuring, and the use of attribute grammars neatly combines data and control structures.

The major contribution of our work is blending together a set of features coming from GUI design, compilers, software engineering and formal verification. Even though we do not claim novelty in each of the techniques adopted for VEG, they have been united into a toolkit supporting all GUI design phases, i.e., specification, design, verification and validation, linking to applications and coding. One of the major advantages of the VEG approach is the possibility of the interaction of development and step-by-step verification, thus allowing early and continuous testing of the application

The paper is organized as follows: Section 2 introduces the basic VEG control notation, Section 3 extends the notation with data manipulation, Section 4 illustrates automatic verification in VEG, Section 5 shows an example of modular GUI design and of its verification, Section 6 reports on the implementation, and Section 7 discusses extensions to more sophisticated interaction paradigms. Finally, Section 8 describes related work and Section 9 draws a few conclusions. An Appendix defines VEG syntax and semantics.

2 THE VEG FORMAL FRAMEWORK

The basic idea underlying the VEG framework is to consider sequences of user input events as sentences in a formal language. These sentences obey some syntactic rules described by grammars (or automata). For example, in some circumstances, opening a document that is already open should be forbidden. In this sense, the grammar describes all authorized sequences of input actions.

2.1 A simple example

The scene of this introductory example is composed of a window with three main graphical components (see Figure 2.1): the central component is a small widget representing a juggler while the other two are buttons, labeled Start and Stop respectively, to control the juggler's behavior. When the window is started, the juggler is idle. When the Start button is pushed, the juggler is expected to resume or to start juggling. On the contrary, when the Stop button is pushed, the juggler is required to stop juggling and to become idle. In addition, there is a Quit action to destroy the scene, selected by pushing the small cross at the topmost, right-hand corner of the window. We ignore the other buttons on the topmost part of the icon.

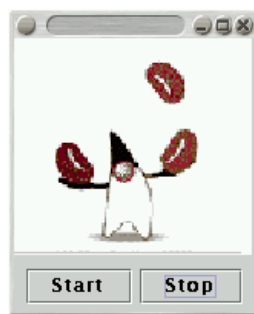


Fig. 2.1: A Juggler

We classify the user actions of pushing the Start, Stop or Quit buttons as *input events*, denoted with `<start>`, `<stop>` and `<quit>` respectively. A simple VEG specification of a Juggler window is the following one:

```
Model SimpleJuggler
Axiom inactive
inactive ::= \createScene idle
idle ::= <start> \startJuggling juggling
juggling ::= <stop> \stopJuggling idle
           ::= <quit> \destroyScene
End SimpleJuggler
```

A *model* such as SimpleJuggler is the smallest unit of modularity in VEG. A model is a local grammar describing the behavior of an automaton[†], and it is composed of a set of *grammar rules*. A rule has an optional left-hand side: a variable name (also called a *state* or a nonterminal symbol), such as *inactive*, *idle* or *juggling*. A rule always has a right hand-side, describing the behavior of the model when in that state. When a rule has an empty left-hand side, it is called a *default* rule and it may be applied in every state.

A rule is similar to a production in the traditional sense of BNF grammars, containing nonterminal elements (the states *idle* and *juggling*) and terminal elements, which are either *input events* (such as `<start>` and `<stop>`) or *visual actions*, (such as `\createScene` and `\startJuggling`). Also other types of elements may appear in a rule, to be detailed later. The textual notation reflects the category of the items: input events are in angle brackets, as `<start>`, and visual actions start with a backslash, as `\stopJuggling`.

An input event is actually an abstraction of a low-level event (such as a left-click of the mouse) or even of a low-level event sequence. For instance, in this example, the `<quit>` event models reception of a closing message received from the window manager, while `<start>` and `<stop>` model the activation of the corresponding buttons.

The VEG notation does not specify how the input events are linked to the low-level events. For instance, the input event `<start>` is not explicitly connected to a left-click of the mouse on a PushButton labeled Start. The link between events is not expressed in VEG, but by means of a dedicated, platform-dependent tool, which filters low-level events and translates them into input events. In our example, the `<start>` and `<stop>` input events could have been synthesized from button activation, menu selection or keystrokes shortcuts. In fact, the precise view of a “button” in the juggler (i.e., a visual object to be provided by some graphical component of the platform) is not specified: the only logical requirement is that this component may be activated, disabled and enabled. Since the same VEG specification is compatible with different layouts and portable on different platforms, it is possible to experiment with various layouts and platform elements and to reuse specifications defined in different projects, with a different layout.

Visible actions are the GUI responses to the interaction with the user. For example, the visible action `\createScene` is used to create a window or a widget, which are visual objects of the platform. Visible actions correspond to suitable callbacks, defined in a Semantic Library; they can also have parameters (i.e., variables), as described in Section 3. The content of the action may be explicitly programmed in a programming language or selected from a set of predefined actions.

[†] The automaton must be deterministic and often it is finite-state. Deterministic *pushdown* automata are also allowed, but they are rarely useful and limit the possibilities of automatic verification. Some of the constructs of VEG may actually lead to infinite-state behavior, as explained in Section 4.

2.1.1 Models, instances, visual objects, creation and destruction of scenes

Each model must be considered akin to a class in object-oriented programming languages. Hence, it must be instantiated before use. Each instance, also called a *VEG object*, has its own reference and its state, but it obeys the same rules of behavior prescribed in its model. The initial state of the object is an axiom specified at its instantiation.

Each VEG object usually has a visual counterpart, called the *visual object associated* with the instance (such as a `PushButton`, a `TextWindow`, etc.), which depends on the platform. The link is prepared at run-time by the visual action `\createScene`, which creates and initializes all the necessary visual components. The window or the widgets so defined are associated with a VEG object: only one window or one widget may be associated with one VEG object at any time. The visible action `\destroyScene` is used to destroy the window or the widget associated with a VEG object. When a window is destroyed, also all its visual components are destroyed.

Some VEG objects may not be associated with visual objects, for instance because they behave as a controller of other models or because the corresponding visual objects have not been created yet.

2.1.2 BNF format and syntax-diagrams

In general, the right-hand side of a rule may be in extended BNF format, i.e., with the alternative operator `|`, the optionality operator `[]` and the iteration operator `{ }`. In this paper, we use a simplified format for rules, called *generalized right-linear*, detailed in the Appendix, which is very convenient for automatic verification. Here we just notice that each rule may have various alternatives, separated by the operator `"|"`, each having at most one nonterminal variable in the rightmost position.

The textual notation of formal grammars that was shown here is introduced only for explanation purposes. It is not always very convenient, since often GUI designers have a preference for visual tools. Actually, the VEG toolkit includes a visual tool for entering and editing formal grammars represented as syntax diagrams, with a specific visual notation, but it is not detailed in this paper (see Figure 6.2). Internally, grammars are represented by an XML document, which of course is not expected to be directly edited or read, allowing the reuse of existing XML tools.

2.2 VEG constructs for modular decomposition

The aim of this section is to introduce and illustrate some of the other concepts used in the VEG framework, mainly modularity, enabling/disabling of widgets, parallel composition and communication events.

Any non-trivial GUI can hardly be described with one model. The overall grammar of a GUI system is usually partitioned into larger units, called *packages*. Each package is a collection of models, with standard visibility rules, similar to those of Java. Model declarations may also be imported from other packages.

Models may be instantiated and launched in a specified state, and then run in parallel by means of *parallel composition* in the following sense. They concurrently listen for input or communication events collected by a Dispatcher module: when receiving one event, a component is activated, consumes the event, executes a state transition and then goes back to the "listening" state. When a component is activated, the other components are not allowed to collect events until the first component has completed a state transition. Therefore, parallel composition has an interleaving semantics.

To illustrate these features, we take an overly simple case and show how the Juggler specification could be broken down into modules, by separately specifying the control of each component. The level of abstraction is lowered for the purpose of explanation of our model, by attaching a VEG object to each visual component of the

juggler instead of abstracting a visual component with an input event. This also shows that our tool allows GUI designers to consider different levels of abstraction, even in the same specification. The choice of the abstraction level may also have an influence on the validation phase described in Section 4: the more detailed the specification, the more precise (and the more difficult) the validation.

A model called Activator is in charge of controlling a button, regardless of the actual widget implementing it. An instance of Activator may be in one of two states: *enabled* or *disabled*. When enabled, it may be activated (e.g., pushed): in this case it sends a message (a communication event) *activated* to any model willing to listen and goes to the *disabled* state. When disabled, it waits for a message *enable*: if it receives it, it goes back to the *enabled* state. Also, an Activator object, when created, may be started in any of the two states. This may be easily accomplished because a model may have more than one axiom, to be selected at creation time.

This can be described in VEG with the following specification, which we assume is part of a package called BasicComponents.

Package BasicComponents

Model SimpleActivator

Axioms disabled, enabled

enabled ::= <activate> !activated disabled

disabled ::= ?enable enabled

End SimpleActivator

The input communication events (i.e., received messages) start with a question mark, followed by the name of the event, such as in *?enable*. Output communication events (emitted messages) start with an exclamation mark followed by the name of the event, such as in *!activated*. In both input and output communication events the name of the source or of the destination of the event is optional.

The juggler can be described by a model as follows:

Model Juggler

Axioms idle

idle ::= ?start.activated \startJuggling !stop.enable juggling

juggling ::= ?stop.activated \stopJuggling !start.enable idle

End Juggler

A juggler, when idle, expects to receive a communication event *?start.activated*, i.e., an event *activated* coming from a SimpleActivator called *start*. Next it starts juggling and sends an event *enable* to a SimpleActivator called *stop*, going to the state *juggling*. The definitions of *start* and *stop* must be given in the same package of the Juggler. When juggling, a juggler waits for a communication event *activated* coming from *stop*: if it receives it, it stops juggling, enables the Activator *start* and becomes idle again. Notice that the optional ability of specifying the source and the destination of communication events may avoid name clashes of events among different models, but it reduces the genericity of the model (e.g., the SimpleActivator).

Another model, called Box, is in charge of creating and initializing the visual objects and instantiating the models. To start a juggler, one has to instantiate the Box.

Model Box

Axioms start


```

start ::= \createScene begin
begin ::= launch(          --parallel composition of objects
    juggler = Juggler.idle,
    start = SimpleActivator.enabled,
    stop = SimpleActivator.disabled)
    running
running ::= <quit> \destroyScene
End Box

```

The *launch* operator (which may appear also in many rules) is used to denote a parallel composition: some children processes are created, named and started in the specified state. This is a form of *dynamic instantiation* of windows and widgets. In the example, an instance of the Juggler is created and initialized in the *idle* state, by means of the statement *Juggler.idle*, acting as a constructor of an object of type Juggler, and its reference is stored in an instance variable called *juggler*; two instances of SimpleActivator are created in the *enabled* state and in the *disabled* state, respectively and are stored in the instance variables *start* and *stop*. The three instances are then started in parallel.

In this example, the Box is only a *container* without human interactions (except for the event *<quit>*) in charge of launching the other components and of initializing the scene. In other cases, such as the speech dialogs of Section 7, a container may be a privileged center of communication between its members and other components.

The complete specification of the juggler is organized in a package as follows:

```

Package JugglerPackage
import BasicComponents
Model Juggler ...
Model Box...

```

Object names are global in the same package. All objects run in parallel (in the sense already explained) and may synchronize by exchanging communication events (which, basically, are higher-priority, internally-generated events).

2.2.1 Enabling and disabling widgets

In the Juggler example, the *start* Activator cannot be activated when the Juggler is already juggling. This behavior results from the VEG specification: when the Juggler is in the state *juggling*, the *start* Activator is in the state *disabled* and the input event *<activate>* is not accepted in this state. In other words, the event *<activate>* is not in the lookahead set of the state *disabled*. In general, the *lookahead* set is the set of events that may be accepted in the current state.

Various approaches may be followed for handling occurrence of events that are not in the current lookahead set. First, the GUI may simply ignore the event. This solution, however, has a problem because the user of the GUI is left wondering which actions will have an effect. A friendlier approach is to disable the components responsible for handling the event: Nymeyer [1995] introduced the use of lookahead sets to disable the components (such as buttons or menu items) that are not relevant at a given time, by shading them out: shading is precisely a visual counterpart of disabling. A more general solution is that more general actions can take place to treat undesired events, under the responsibility of the receiving instance. In this framework, the instance is just informed that it

cannot be reached from a given component and it executes the appropriate action that may disable and shade the component or raise a warning in response to undesired component events, etc. This solution also simplifies the reuse of the same logic with multiple views, and is followed for instance by Bastide and Palanque [1995]. VEG allows the designer to follow any of these approaches.

3 MANAGING DATA STRUCTURES

The expressive power of syntax is not always adequate to model specific behaviors. For example, a typical login session needs a function to check whether the password is correct. This kind of utility routine is of course independent of GUI design, but the result of a routine may influence GUI behavior to a significant amount. Semantic *functions* and *variables* are designed for modeling this role. They are collected in a so-called Semantic Library and are written in a classical programming language (Java in our prototype). The result is an attribute grammar-style declarative specification.

A *semantic function* takes some values as arguments and may compute other values as result. Also visible actions may receive/return values. *Variables* (or semantic attributes) can be associated with input events, states and communication events in order to store values to be passed to and from functions and actions. For instance, many input events, such as pressing a key or pointing and clicking, have some values associated with them, such as the character entered or the numeric value selected on a scale. Such values may be stored in one or more variables (attributes) of the event. Values associated with communication events may be used to interface the models in a parallel composition.

Each variable must be properly initialized before its use. For instance, whenever a VEG specification describes a state transition, each variable associated with the next state must be initialized calling a suitable semantic function. The set of values a variable can take constitutes its *domain*. A domain can be any data type, simple (boolean, integer, ...) or structured (array, record, ...). Attribute domains are declared with the syntax of the programming language to be used for coding the semantic library.

Some semantic functions may play a special role: they are semantic *predicates* and semantic *actions*. Semantic predicates are boolean functions that can be used as guards in a syntactic alternative of a production, as shown next, and hence may have a direct effect on the behavior of a GUI. Semantic actions may be inserted in any point in the VEG productions and are meant to model actions that the GUI must accomplish in response to events, but without a visual effect (such as searching an element in a list, storing a value in a file, etc.). Semantic actions and predicates are typically used to link a GUI to the actual application.

3.1 Example of semantic functions

A *Login Session* is a dialog where a user is required to enter a username and a password in order to start a session with a remote host. The dialog box is composed of two text-input fields, *user name* and *password* and the button *Ok*. After entering name and password, pressing the *Ok* button starts a verification procedure. If the login is correct, the dialog is closed and the connection is established. If the login is not correct, the user is asked to retry. After a fixed number of failed login attempts, the dialog is closed. It is possible to abort the dialog at any time by pressing the *Quit* button.

In a possible VEG specification the action of pushing the button labeled *Ok* after filling the two textfields is modeled with an input event `<enterData>`. The user name and the password are two string variables *userName* and

passwd of the event. The integer attribute *n* of the state *login* keeps track of the number of login attempts. To deal with variables, the specification is annotated with suitable calls to semantic functions that are italicized here. Each annotation is preceded by a slash, such as */login.n=0*.

Model LoginSession

Axioms start start ::= \createScene */login.n = 0* login

```
login ::= <enterData> /login.n = increment(login.n)
      if /loginCorrect(enterData.userName, enterData.passwd)
        launch(Session.start(enterData.userName)) \destroyScene
      elsif /notTooMany(login.n) \message("Retry") login
      else \message("Login Failed") \destroyScene
      fi
```

End LoginSession

The semantic predicates *loginCorrect* and *notTooMany* are used to discriminate among the various alternatives, by means of an *if... elsif... else fi* construct. We assume that there is one model called *Session*, which is instantiated and launched in the state *start*, receiving also the current value of *userName*; the latter value will be used by the *Session* object to initialize a variable associated with the state *start*. When the semantic functions *loginCorrect*, *notTooMany*, *increment*, to be included in a Semantic Library, are properly implemented in Java, the VEG toolkit is able to generate an integrated LL(1) parser/semantic analyzer, as illustrated in Section 6, and included in the run-time architecture of VEG.

The first version of a VEG specification need not detail semantic functions and variables, to be added in a successive refinement. In this case, a VEG specification is said to be *purely syntactical*. For instance, the rules for the login specification could be written as:

```
start ::= \createScene login
login ::= <enterData> /increment
      if /loginCorrect launch(Session.start) \destroyScene
      elsif /notTooMany \message("Retry") login
      else \message("Login Failed") \destroyScene
      fi
```

Executable code cannot usually be generated from a purely syntactical specification, which may lack variables and the implementation of actions. However, powerful forms of verification and validation, as shown in Section 4, are allowed for such specifications, making them very useful for prototyping and design. Since in this paper we are mainly interested in showing the support of VEG to design, verification and validation, rather than explaining how to program a real GUI, all VEG specifications in the remainder of the paper are purely syntactical.

3.2 Predefined semantic functions

In the example of Section 3.1, only *loginCorrect* really needs to be defined by a semantic function, since *notTooMany* and *increment* could be specified with finite-state counting. Using finite-state constructs instead of semantic functions is preferable for validation purposes, since clearly Java programs are much harder to deal with than finite-state machines. However, explicit finite-state control can be very inconvenient: counting up to a threshold *K* requires at least *K* control states. To make finite-state control easier, VEG provides a set of predefined

finite-state semantic functions and variables, such as those related to modulo counting: finite counters, boolean variables, increment or decrement, modulo comparison, initialization, comparison with a constant. Even though they are considered semantic functions, they are translated to finite state constructs for verification and validation. Potentially, these finite state constructs may also lead to very large state spaces, but this is rarely the case.

Currently, only counters and another set of finite-state semantic functions, namely the group selection features shown in Section 3.3 below, are part of the VEG toolkit. However, new semantic functions can be defined and modeled by finite-state constructs, possibly driven by verification needs. Nevertheless, in the generated code they are treated at the same level as programmer-defined semantic functions.

3.3 Containers, groups and counters

The *children* of a VEG object, at a given instant, are defined as the components *launched* (i.e., instantiated) by the object and which did not terminate their execution yet. A VEG object that may generate children is called a *container*. Some special VEG constructs are available to simplify communication among the children and between the container and its children. A container can broadcast a communication event to all its children, by using the target *all* (e.g., *!all.enable* sends an event *enable* to all the children of the container). To allow more flexibility, a container may also broadcast communication events to a selected set *x* of children, called a *group*, or to its complement $\sim x$. The creation and the modification of groups may be accomplished by means of a set of predefined semantic actions and predicates, called *group selection mechanisms*: the action */x.add* adds, to the group *x*, the child that sent the latest communication event to the container, while the action */x.remove*, removes it from *x*, if present; the semantic predicate */x.isEmpty* may be used to check whether a group *x* is empty, while */x.clear* makes *x* empty.

For instance, the fragment *?selected /G.add !G.enable !~G.disable* first waits for an event called *selected*. When a child *c* sends the event, *c* is added to the group *G*. Then an event *enable* is sent to all the children in the group *G*, and hence also to *c*, and an event *disable* is sent to the remaining children. An example of application of group selection mechanisms can be found in Section 5.

Group selection features, while being realized with semantic actions and predicates, are essentially finite state, as long as the number of children is bounded, and hence do not hamper the possibility of automatic verification of VEG specifications. In fact, they are useful shorthand notations to avoid an increase in the rules of a model.

4. AUTOMATIC VERIFICATION AND VALIDATION

Model checking [Clarke et al. 1986, Holzmann 1997] is a powerful and useful technique, allowing the automated verification that a finite state machine verifies a given property, often specified in a temporal logic language. Model checking is thus the ideal verification tool, because it is completely automated. However, the verification of non-trivial software systems requires the implementation to be *abstracted* to make the verification feasible. Abstracting a program into a meaningful and relevant finite-state version is not an easy task, since the number of possible states of the system must be greatly reduced. Hence, an abstraction must introduce significant approximations to the original system.

The VEG formalism has been designed in such a way that an abstraction of the system can easily be obtained, allowing an automatic translation into the Promela language of the model checker SPIN. In fact, when studying formal features of an event-driven user interface, the actual values of the parameters passed to and from the semantic and visible actions may be ignored: it suffices to record the event that a semantic or visible action has

been called, rather than calling it. When semantic predicates are present, however, this introduces a loss of precision, since the predicates used as guards of a branch cannot be evaluated anymore: the choice of the branch to be followed becomes nondeterministic.

The translation into Promela considers only some predefined semantic actions of VEG, such as the already cited counters and selection mechanisms, e.g. checking that no event is ever sent to an empty group. Also other semantic actions, such as the number of attempts in the login dialog example, may be easily modeled. On the other hand, programmer-defined semantic actions (ad hoc coded in Java) are ignored by the translation and treated as part of the terminal alphabet. However, the set of predefined semantic actions can be further extended to deal with other semantic actions that can be modeled in Promela.

4.1 Transducers, local and global states, reachability. Stable states.

In order to define the translation from VEG into Promela, a VEG specification P can be conceptually represented as a *transducer* (see, e.g., [Berstel 1979]) T_P computing a transduction from an input alphabet of input events to an output alphabet of (visible and semantic) actions. A transducer is a kind of state/transition automaton: the automaton is initially waiting for input events in an initial configuration; when it “reads” an input event then it makes a transition, consisting in the computation of semantic functions, variable values, internal exchange of communication events, and emission of suitable output symbols. The transition is completed when the transducer becomes ready to read another input event.

The transducer T_P is actually composed of various VEG objects, which may be created and destroyed. Each object is in a *local state* $?c_i, ?_i?$, where c_i is a control state (such as the state *login*), and $?_i$ is an assignment of a value to each semantic variable defined in the state (for instance, $?_i(login.name)$ is the string assigned to the variable *login.name*). A *(global) state* of the transducer T_P is composed of a local state for each object, i.e., it is a global configuration for T_P . In the following, global states are assumed to be *stable*, i.e., to correspond to configurations where the transducer is waiting for input events and no local state change is still pending.

A global state S' is a *successor* of a global state S if S' may be reached from S in one transition; S' is *reachable* from S if it is either a successor of S or a successor of a state reachable from S . Each transducer has an initial state: let the *reachability* $Reach(T_P)$ of a transducer T_P be the set of states of T_P that are reachable from the initial state.

4.2 Semantically-branching transducers and abstract states

Often, T_P contains semantic predicates on variables other than counters and groups, which may cause different branches to be taken depending on variable values. Denote with $Abst(T_P)$ the transducer obtained from T_P as follows:

1. replace the *if ... else* constructs, which contain semantic predicates involving semantic variables other than counters and groups, with the alternative operator “|”;
2. eliminate semantic variables and functions, other than those for counters and groups;
3. eliminate all arguments of semantic and visual actions.

In practice, the transducer $Abst(T_P)$ corresponds to a purely syntactical version of T_P . In general, the transduction computed by $Abst(T_P)$ is different from the transduction computed by T_P : in this case, T_P is called *semantically-branching*.

For each local state of the form $\langle c, \tau \rangle$ its *abstract* version is $Abst(\langle c, \tau \rangle) = \langle c, \tau' \rangle$, with τ' being the restriction of τ to counter and group variables. An *abstract state* of T_P is obtained from a global state of T_P by considering the abstract versions of its local states. We overload the name $Abst$ to denote also the operation of abstraction on a state or on a set of states. A state is also called *concrete* when it is not abstract.

4.3 Reachability, Deadlock, Invariants

We now show that the abstraction $Abst$ is meaningful for deadlock detection, analysis of unreachable states and state invariant verification. It is easy to see that:

$$Abst(Reach(T_P)) \subseteq Reach(Abst(T_P)) \quad (*)$$

i.e., the abstract states in the reachability of T_P are also in the reachability of $Abst(T_P)$. Hence, if a concrete state is reachable for T_P , its abstract version must be reachable for the transducer $Abst(T_P)$. This suggests that information about T_P may be deduced from verifying $Abst(T_P)$.

Given a transducer T_P , assume that all its semantics functions, semantic actions and visible actions terminate for all their inputs. A *deadlock state* for T_P is a state of T_P that has no successor; T_P is *deadlock-free* if no state in $Reach(T_P)$ is a deadlock state.

A *state property* is a set of states of T_P . A *state invariant* J for T_P is a superset of $Reach(T_P)$ (i.e., $Reach(T_P) \subseteq J$). For instance, a state invariant for the Juggler example may assert that the Activator *start* is in the local state *enabled* if, and only if, the Activator *stop* is in the local state *disabled*. Notice that if an abstract state verifies a state property, also its concrete version does.

The following properties are a consequence of the above definitions and of property (*):

Statement 1.

- 1) For each state x of T_P , if $Abst(x)$ is not reachable in $Abst(T_P)$, then x is not reachable in T_P .
- 2) If $Abst(T_P)$ is deadlock-free, then T_P is deadlock-free.
- 3) Let J be a state property defined on the states of $Abst(T_P)$. Then for all abstract states $\langle c, \tau' \rangle$, if $\langle c, \tau' \rangle \in Reach(Abst(T_P)) \subseteq J$ then $\langle c, \tau' \rangle \in Reach(T_P) \subseteq \hat{J}$, where $\hat{J} = \{\langle c, \tau' \rangle \mid Abst(\langle c, \tau' \rangle) \in J\}$.

Proof: Part (1) is an immediate consequence of Property (*). Part (2) follows by contradiction: if T_P has a deadlock state x then, by (*), $Abst(x)$ is reachable in $Abst(T_P)$. Notice that the only case of a deadlock state is a communication event pending for an object that is able to consume it (typically, the event was sent to an object that was not waiting for it). In fact, in VEG, as formalized in the Appendix, each *if ... fi* construct has always an *else* branch, in each state all guards depend only on (communication or input) events, and we assumed that all visual actions and semantic functions terminate on all inputs: a deadlock never arises because of variable values. Hence, since $Abst(x)$ has the same communication event still pending (because it is not abstracted), it must also be a deadlock state. Part (3) is just a special case of the main result of Clarke et al. [1994]. A direct proof is as follows: let $\langle c, \tau' \rangle \in Reach(T_P)$. Apply the $Abst$ operator on each side: $\langle c, \tau' \rangle \in Abst(Reach(T_P))$. By (*), $\langle c, \tau' \rangle \in Reach(Abst(T_P))$. By hypothesis and modus ponens, $\langle c, \tau' \rangle \in J$. Thus, $\langle c, \tau' \rangle \in \hat{J}$.

Statement 1 shows that the chosen abstraction is safe with respect to reachability analysis, deadlock-freedom and state invariants. However, the verification on the abstract version may find *spurious counterexamples*, i.e., errors on the abstraction only. In fact, the converse properties of Statement 1 may not hold when T_P is semantically-branching. For instance, a concrete state guarded by a semantic predicate may be unreachable because the predicate

is always false, while its abstract version is reachable (because it has no guard), contradicting the converse property of Part 1; also, the abstract version may be a deadlock state, even though is not reachable in T_P , thus leading to a spurious counterexample.

4.4 Finite vs. infinite state

Often, $Abst(T_P)$ is finite-state (e.g., the Juggler example) and hence it can be verified by using conventional model checking techniques. The number of states of $Abst(T_P)$ for a GUI is usually not very large: for instance, it can be just a few hundreds for a simple text editor like the Windows Notepad and even much larger numbers can be handled by model checkers.

The transduction $Abst(T_P)$, however, is not finite-state when the number of VEG objects in $Abst(T_P)$ cannot be bounded by any integer (e.g., the user is allowed to instantiate any number of copies of a window). In this case, $Abst(T_P)$ is called an *unbounded* transduction and cannot be verified using the standard model checking techniques of SPIN. In this case, we introduce another approximation, by fixing an upper bound N on the number of VEG objects that can be created. Under these assumptions, we obtain a *finite-state approximation* $Abst_N(T_P)$ of $Abst(T_P)$. The transduction $Abst_N(T_P)$ can be verified by model checking when N is not too large. The verification results for $Abst_N(T_P)$ are in most cases extendable to $Abst(T_P)$ and then to T_P , as shown by the next statement.

Statement 2. *Let $Abst(T_P)$ be an unbounded transduction.*

- 1) *If a state x is reachable in $Abst(T_P)$ (and thus in T_P) then it is reachable in some $Abst_N(T_P)$.*
- 2) *If there is a deadlock in $Abst(T_P)$ then there is a deadlock also in some $Abst_N(T_P)$.*
- 3) *If an invariant J is not verified for $Abst(T_P)$ then it is not verified also for some $Abst_N(T_P)$.*

Statement 2 cannot give any guarantee that the results of a verification on a finite-state version $Abst_N(T_P)$ may be extended to an unbounded $Abst(T_P)$: for a given N , $Abst_N(T_P)$ may not be a safe abstraction of T_P . For example, a deadlock in $Abst(T_P)$ may occur only when a certain number $k > 0$ of instances of a model have been created: checking only $Abst_N(T_P)$ with $N < k$ cannot detect the deadlock. However, it is often the case that either a system does not work correctly for a small value of N (such as three) or it works correctly for every N . Hence, even with small values of N we can increase the confidence in the correctness of the specification: the results of the verification obtained on the approximation give some useful and meaningful hints on the correctness of the original specification and may potentially detects errors, although they cannot be automatically generalized. There are also cases where the generalization is actually guaranteed, as shown by Norris Ip and Dill [1996]. A symmetry-based reduction on finite, unordered sets (not to be used as an array index or a loop index) allows the automatic reduction of an infinite state case to a finite state one. This technique could be exploited also for VEG, since its assumptions are often verified by VEG containers.

4.5 Verification and validation in practice

During the experiments that various developers and we performed, many errors have been found and corrected on complex specifications by using SPIN. Once an error is found, SPIN builds a counterexample that shows the internal sequence of events and state transitions leading to the error. Knowing the internal behavior of the system makes debugging much easier: in traditional testing of the actual GUI only the "external" behavior of the program is visible. At present, there is no tool allowing to trace back errors from SPIN to the original VEG specification, but this is not a major obstacle for comprehension, since the generated Promela code is conceptually, even though not syntactically, very close to the original VEG specification.

We do not expect average GUI designers to find Promela easy to use as a design language, since its constructs may have a very subtle semantics. Also, it is not clear how to integrate semantic functions into Promela code. However, we included in VEG only that small part of Promela's expressive power that is useful for GUI construction. Hence, VEG is much simpler and explicitly devoted to the production of GUIs, and it can be much more convenient to use than Promela for such applications.

Some of the errors detected with SPIN could have been found also with simulation and testing, but some would have probably escaped even an accurate testing phase. For instance, in the specification of a Notepad-style editor (a small piece of which is reported in Berstel et al. [2001]), when the mouse button was held to make a selection, it was possible to activate other dialogs with keystrokes and to send commands such as paste or cut (for instance, allowing to cut a text which was not yet completely selected). With SPIN this error was immediately reported by deadlock detection, *before* designing the layout and linking it to the VEG specification. This kind of error can also be found with traditional testing, but with higher costs of detection and correction.

As already defined, validation is also supported by means of state invariants (i.e., boolean assertions that predicate about the states of the various model instances) to be checked against a VEG specification. State invariants are meaningful only in stable states (i.e., configurations where the transducer is waiting for input events and no local state change is still pending).

State invariants in VEG are typically associated with a model or a package but they may also be associated with a rule, to denote invariant properties that hold whenever an object is in the stable state associated with a rule. The notation $x(S)$ may be used as a boolean predicate to denote that the object named x is in the stable state S . Also, to allow bounded quantification, the notation $existsX(S)$ may be used to denote that at least one launched instance of a model X is in the state S . Analogously for $forallX(S)$.

Of course, since SPIN allows the checking of linear temporal logic formulas and the verification of logic assertions, an experienced model checking specialist could directly use SPIN for stating and proving the desired properties. However, we prefer to provide some explicit support in VEG for defining at least invariant properties, so GUI designers avoid of dealing with Promela. For instance, an invariant for the above-cited Notepad example of Berstel et al. [2001] is that the cut and the copy items of an Edit menu are enabled if, and only if, the text is selected. If the text is a VEG object that may be in one of two states: *unselected* and *selected*, then the invariant may be written as the conjunction of the following two formulas, where \rightarrow and $\&\&$ stand for logical implication and logical conjunction respectively:

text(unselected) \rightarrow cut(disabled) $\&\&$ copy(disabled)

text(selected) \rightarrow cut(enabled) $\&\&$ copy(enabled)

Each invariant is translated into an *assertion* of SPIN. SPIN can verify assertions (i.e., Boolean formulas) in any point of the specification, with the same effort of deadlock and reachability analysis.

This kind of validation activities is well supported by SPIN. In the above example of the Notepad editor, there was actually a problem with the copy button, which, as usual in text editors, should always be enabled whenever the text is selected: actually, the copy button was an Activator that became disabled after its activation, even though the text remained selected. The problem was that the controller did not send an event to enable it again after an activation. This is a specification error, which could not be detected by deadlock or unreachability analysis, but which was easily found using the assertion checking capabilities of SPIN to verify that the above invariant

text(selected) -> cut(enabled) && copy(enabled) was not verified. SPIN showed the path leading to the error, allowing us to identify the problem in the usage of activators that become disabled after the activation. Again, the error could have been found also with traditional testing, but with much higher costs of detection and correction.

SPIN also allows writing temporal logic specifications. Currently, our tool provides no direct notation for doing this: a formula must be inserted via the SPIN interface. This requires a certain level of competence with SPIN and with temporal logic.

5. HIERARCHICAL DESIGN OF SCALABLE COMPONENTS AND ITS VERIFICATION

This section shows how to combine some VEG constructs for modular design of graphical interfaces, in order to obtain a hierarchical description of visual communicating modules. To show that VEG is scalable, we illustrate the modular design of a typical GUI. The following example shows also how complex communication is modeled by a VEG grammar and illustrates the group selection mechanisms.

5.1 The Switcher

The Switcher is a user interface with two or more lists, and one button allowing transfer from one list to the other. This interface is used for instance in some ftp tools, and was popularized by the Macintosh Font/DA Mover. A typical layout is shown in Figure 5.1 with only two lists.

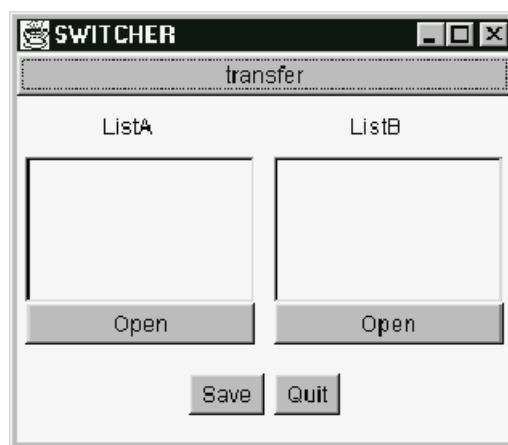


Fig. 5.1: The Switcher

Indeed, the main purpose of the application consists in allowing the move of a selected item from its list to other lists, when the Transfer button is activated. The interface has an apparent simplicity, but the interaction between the components is rather involved. For instance, the transfer button is enabled only if at least two lists have been opened and some item has been selected in one list. However, the button should not be enabled when only one list is open, even if an item has been selected, since no transfer may occur in this case.

Each of the lists is composed of an Open button and of a widget that can show a set of items. This button allows a sequence of items in the list to be loaded. As shown in the picture, there is also a Save button. The role of this button is to save the lists that have been modified. Of course, this button may be activated only after a transfer has taken place since the last save action. We assume for simplicity that pushing the Quit button causes the application to quit without a save dialog, but this could be easily extended to a more realistic, and user friendly, case.

The state of a list is one of *closed*, *ready* or *selected*. Here, *closed* means that no file has been loaded yet, *ready* means that a file has been loaded, but no item has been selected; and *selected* is the remaining case, i.e., an item has been selected in the list. We assume for simplicity (but this is not critical) a single-selection model: only one item

can be selected at any time. When several lists are ready, only one can be selected: the selection of one item in a list immediately deselects any item that could have been selected in another list.

In view of the hierarchical design we adopt here, the VEG specification of the Switcher is composed of a *Box* with three components: *transfer*, the transfer button, *save*, the save button, *quit*, the quit button and the *controller*, that is a container of the lists.

The buttons *transfer* and *quit* have all the same behavior: they may be enabled or disabled based on the global state of the system (hence, on receiving suitable communication events). This behavior of the menu entries is so general that it should be inserted in a library of reusable components, such as the package BasicComponents. Hence, we first describe a more complex Activator than the one used in Section 2, namely an activator that can also be disabled or enabled by receiving a suitable communication event regardless of its current state. To make the Activator as general as possible, the enabling/disabling events are considered only if a source called *client*, specified as a parameter of the model, has sent them. The actual value of the *client* parameter is a VEG object to be specified at creation time. The client is also the target of the communication event *activated*. This model may also be included in the Package BasicComponents.

Package BasicComponents

Model Activator(client)

Axioms disabled, enabled

enabled ::= <activate> \changeAspect !client.activated disabled

DEFAULT ::= ?client.disable disabled -- default rule

| ?client.enable enabled

End Activator

A default rule is in charge of handling the *disable* and *enable* events (hence, no rule for the *disabled* state is necessary). The visible action *\changeAspect* must be specified for changing the aspect of the button (for instance, changing the displayed text, etc.).

The *Controller* contains instances of a *List* model, each one associated with an *open* button. To make the design modular, all communications to and from the lists is through the controller. For instance, no knowledge even of the existence of the lists is necessary for the transfer button: it just notifies the controller when it is activated. This design has also the advantage of allowing a variable number of lists, without any modification to the *Controller* model, except for the initialization (start) state.

The Box model, part of a package *Switcher*, is used to launch a switcher window, instantiating the various buttons and a Controller described below. Both the Transfer and the Save buttons are Activators. The lists are not initialized here, but by the Controller itself.

Package Switcher

Model Box

Import BasicComponents

Axioms start

start ::= \createScene -- scene initialization

launch(

cont = Controller.enabled,

```

        save = Activator(cont).disabled,
        transfer = Activator(cont).disabled
    )
    running
running ::= <quit> \destroyScene
End Box

```

After scene initialization, the Box launches a Controller called *cont* and two Activators, called *save* and *transfer*, by instantiating the client parameter of the Activator model with the controller itself. Then the Box goes to the *running* state where it waits for the *<quit>* input event (associated with the Quit button), before destroying the scene.

The list components of the switcher have a less general behavior and the corresponding model is included in the Switcher package rather than in a library package.

A list is initially *closed*. When its open button is pushed (i.e., an *open* input event is generated), it executes a visible action *\load* (such as, load a list of items from a file and show them), it notifies the controller and goes to the state *ready*. In the *ready* state, if an item is selected the list goes to the *selected* state, notifying the controller; if it is opened again (e.g., loading a new set of items) it stays in the current state; otherwise, it waits for the controller to send suitable events in order to save its current content or to insert a new item in the list (by means of the visible actions *\load* and *\insert*). In the *selected* state, another item may be selected, or the list may be opened again (losing the previous selection and notifying the controller) or it may save the current content; finally, the list may receive a *setTransfer* event, deleting the current item. Notice that changing the selected item is treated internally to the list (i.e., no communication event is sent). Moreover, there is no input event to deselect a list: deselection may only be triggered by a selection in another list or by a new *open* command on the same list (possibly, with the effect of reopening the file). The specification of the actual item to be inserted or deleted is left to the semantic part of the specification (e.g., a reference to it may be exchanged via the communication events *getTransfer* and *setTransfer*). Notice that a list does not notify the controller when it is reopened.

Package Switcher

Model List

Axiom closed

```

closed ::=
    <open> \load !firstOpened ready
    | ?save /save closed
    | ?getTransfer closed
ready ::=
    <selectItem> !selected selected
    | <open> \load ready
    | ?save /save ready
    | ?getTransfer \insert ready
selected ::=
    <selectItem> selected

```

```

| <open> \load !unselected ready
| ?deselect ready
| ?save /save selected
| ?setTransfer \delete ready
End List

```

The Controller *cont* is used to synchronize the behavior of the lists. After launching the lists, the controller may be in one of the states: *allClosed*, *oneReady*, *severalReady*, *transferable*. In the *allClosed* state, the controller waits for one of the lists to become open. In the *oneReady* state it waits for another list to become open. If a list is selected before any other list is opened, then the controller stays in the state *oneReady*, but it adds the list to a group, called *sel*, in charge of storing the last selected list. If the second list is opened, then it checks whether the first list was already selected (*if sel.clear*): if this is the case, a transfer is already possible, otherwise it goes to the *severalReady* state. When one list is ready and another is selected, then the controller enables the transfer button and enters the state *transferable*. Finally, if the save button is activated, the controller notifies all lists in the group and goes back to *severalReady*. In the state *transferable*, if the transfer button is activated (*transfer.enable*), the controller makes the transfer happen, sending the *getTransfer* and *setTransfer* events, enabling the save button and going back to *severalReady*. Otherwise, if one of the lists is unselected, it goes back to *severalReady*, or if a list is selected, it deselects the list which was selected, clears the *sel* group and add the selected list to the group, staying in the *transferable* state. Finally, if the save button is activated, the controller notifies all lists to save their content.

The List model and the Controller model were designed to work without any knowledge about names and number of the lists. Since each list has three states, it may seem that 3^n states are required to represent the behavior of the Controller for $n > 1$ lists. In fact, only four states are needed because of the selection mechanism, which allows the identification of symmetric configurations. For instance, *oneReady* holds in four configurations, namely when one of the lists is either ready or selected and the others are closed, but it can be represented with one state since the group selection keeps track of which list is selected. This state saving may appear small if n is small (four states against nine if $n=2$), but the number of states of the controller is actually a constant, not changing with the number of lists.

The save button is enabled only after a transfer has been done.

Package Switcher

Model Controller

Axiom start

Group sel

start ::= launch(List.closed, List.closed) allClosed

allClosed ::= ?firstOpened oneReady

oneReady ::=

 ?firstOpened if /sel.isEmpty severalReady else !transfer.enable transferable fi

 | ?selected /sel.add oneReady

 | ?unselected /sel.remove oneReady

severalReady ::=

 ?firstOpened severalReady

 | ?selected /sel.add !transfer.enable transferable

 | ?save.activated !all.save severalReady

```

transferable ::=
  ?firstOpened severalReady
  | ?selected !sel.deselect /sel.clear /sel.add transferable
  | ?unselected /sel.remove !transfer.disable severalReady
  | ?transfer.activated !sel.setTransfer !~sel.getTransfer /sel.clear !save.enable severalReady
  | ?save.activated !all.save transferable
End Controller

```

5.1.1 Variations of the example.

We could avoid the use of the semantic predicate *isEmpty*, by expressing this choice syntactically, introducing a new additional state to record the configuration when the only ready list is also selected. On the other hand, we could add further semantic predicates and merge also the states *transferable* and *severalReady*. The choice between syntactic and semantic solutions should be driven by readability of the code and by the fact that verification is simpler with the syntactic solution.

This example, while small, is very complex. However, the complexity resides in its logic and not in the VEG notation, as it should be clear by looking at the very long informal description of its detailed behavior. The VEG notation is actually much clearer than any informal one. Moreover, some of the complexity of the notation depends also on the fact that the example has been designed to work with many lists: a transfer is done from the selected list to all other ready lists. For instance, for three lists, just write in the controller:

```
start ::= launch(List.closed, List.closed, List.closed) allClosed
```

This illustrates the generality and extendibility of this approach and the reusability of the components. However, one should not assume that the validation of the two-list switcher implies the three-list version is correct: the latter specification should also be validated with SPIN. It is also easy to generalize the Controller specification in order to handle dynamic instantiation of lists. However, in this case verification should be bounded to a maximum number of lists (three being the most reasonable bound).

5.2 Verification and validation of the switcher

The Switcher example was the perfect test bed for applying verification and validation techniques before implementation, since its subtle logic may cause many design errors. Actually, various errors were found in earlier versions of the Switcher by using deadlock detection in SPIN.

For instance, the original specification used a SimpleActivator instead of an Activator. We recall that if a SimpleActivator is enabled then it waits either for a *disable* event or an *activate* event, but it does not expect an *enable* event. However, the Controller, after every transfer, sends an *enable* event to the Save button, which is already enabled and cannot receive it: the system enters a deadlock.

Another example is the original specification of the switcher, intended to allow any number of lists, where some parts relied on the assumption that only two lists were used. For instance, the specification of the List wrongly assumed that no *save* event could be received when the list is still closed: with more than two lists, a *save* event may actually be received by a closed list after a transfer between two ready lists, since the statement *!all.save* sends a *save* event to all lists, whether ready or closed. Similarly, the *getTransfer* event is sent to all lists, including the closed ones: a first version of the specs did not deal with a *getTransfer* in the closed state of the lists, causing a deadlock in the specification.

Reachability analysis allowed us also to find that we introduced some useless alternatives in the Controller rules (e.g., waiting for an *unselected* event in the *severalReady* state of the Controller, which will never happen, since in this state no List is selected): the alternatives were eliminated, leading to a simpler specification.

Another issue is validation: how can we be confident that the behavior of the Switcher is exactly the intended one? As already remarked, simulation and testing may also be used, but property verification may be a very useful validation technique. For instance, it is easy to understand that the correct behavior of the Controller is based on the fact that in the state *closed* no list is ready, in the state *oneReady* only one list is ready, in the state *severalReady* at least two lists are ready but none is selected, and in the state *transferable* one list is selected and at least another list is ready. These properties can be considered as invariants of the specified system.

In particular, the invariant asserting that if the Controller *cont* is in the state *transferable* then there is one List that is in the state *ready* and another one that is in the state *selected*, is written as:

```
cont(transferable)-> existsList(ready) && existsList(selected)
```

As already pointed out, also the temporal logic LTL may be used. For instance, the above invariant may be written in temporal logic as:

```
[](stableState -> (cont(transferable) ->
    (existsList(ready)&&existsList(selected))))
```

where *stableState* (with a suitable definition) holds whenever the system is in a stable state. Another example is a liveness property that does not hold on the Switcher specification (where \diamond is the Eventually operator):

```
[](stableState -> (existsList(selected) ->  $\diamond$  cont(transferable) ))
```

with the meaning that if a list is selected, then sooner or later the Controller will reach the state *transferable*. This was verified by SPIN to be false on a 4-list switcher in only 20 atomic steps, showing that there is infinite cycle open/select/open... of events for the same list, preventing the controller from ever reaching the state *transferable*.

6. EXPERIMENTS, IMPLEMENTATION AND OTHER FEATURES OF VEG

This section reports on the VEG implementation and its application, describing also some minor features of VEG to make GUI design more flexible and more efficient.

6.1 The VEG toolkit

The notation and results presented in the previous sections are part of a Long Term Research project of the European community, called Gedisac (Graphical Event-Driven Interface Specification and Compilation). The project comprised several partners from university and industry. The aim of the project was to develop a set of tools, based on compiler technology, to support GUI design and testing. These tools are designed to complement traditional layout tools such as those provided by Java Workshop or J++. The intended users of the tools are both GUI designers and programmers. The toolkit, developed in Java, includes a visual editor of VEG specifications, a parser generator, and tools and libraries for linking specifications to the platform. The development process is depicted in Figure 6.1.

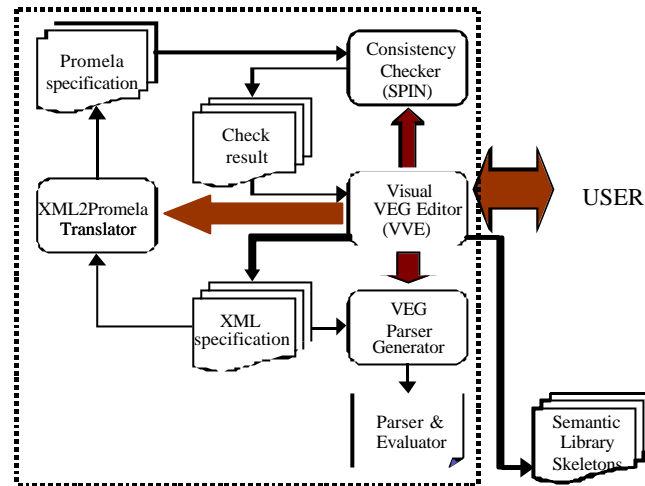


Fig. 6.1 The VEG development process.

The GUI designer interacts with the Visual VEG Editor (VVE) to write a VEG specification (a snapshot of the toolkit is shown in Figure 6.2). Following a request, the VVE produces three different specifications: the first one is an internal XML encoding of the VEG specification without attributes. The second one contains the set of semantic routines used by the VEG specification implemented in the target language Java (Semantic Library Skeletons). The third one is the Promela translation. The designer may check the consistency of the specification, by using the Promela file as input to the SPIN model checker. This model checking is done in a separate process and could be omitted, but this step is essential for verification and validation activities.

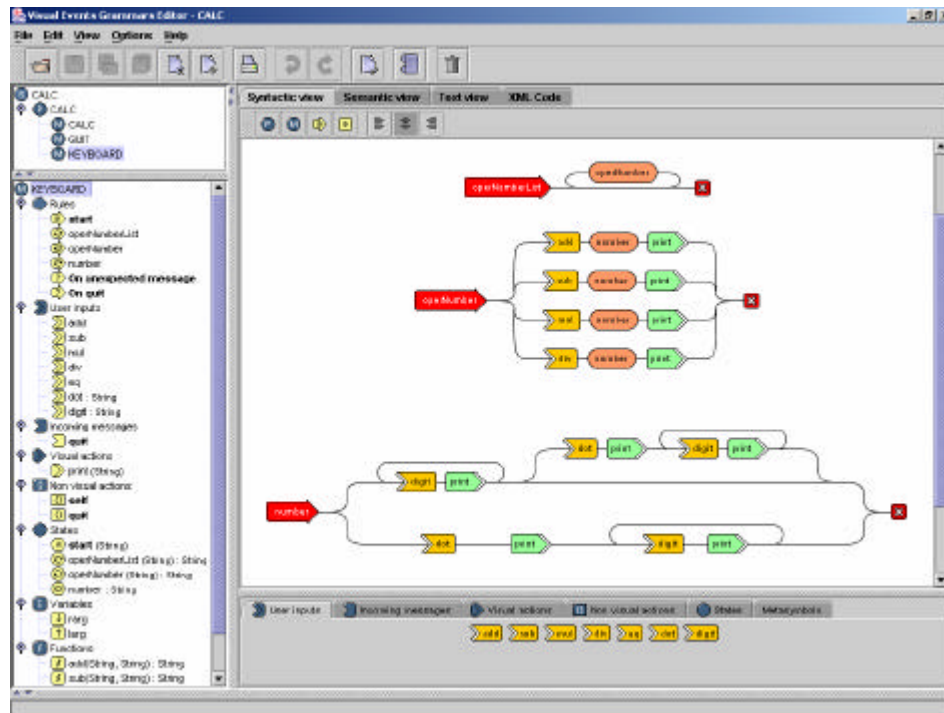


Fig. 6.2 A snapshot of the Visual VEG Editor (on the example of a pocket calculator).

In order to produce the real application, the XML file is used to build a set of communicating parsers and semantic evaluators. These objects are linked with the semantic libraries and with the visual components corresponding to the models, which produce the actual input events.

6.2 Applications and experiments

Realistic applications have been specified and implemented, like a Notepad-style editor, a graph construction library and a large real application to medical software. Various experiments (reported in Campi [2000]) have found no difference in performance between VEG-generated code and hand-written Java code implementing the same application.

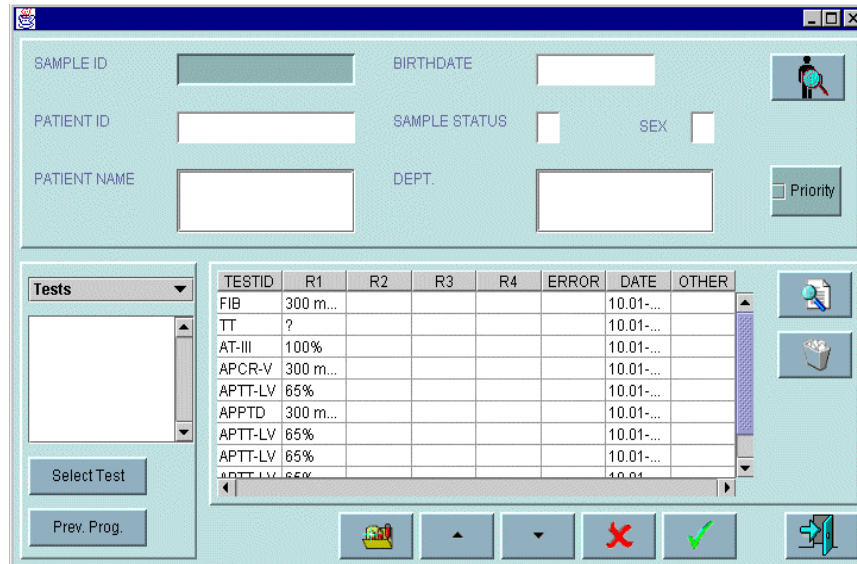


Fig. 6.3 Window of GUI of a medical application whose behavior was developed using VEG.

The development of the complete medical application took originally 24 person/months, of which GUI development accounted for 3.6 p/m. The GUI was redesigned in 1999 with VEG by Txt e-Solutions, an Italian-based software company. The GUI consisted of 10 windows, one of which is shown in Figure 6.3, and 90 widgets of various kinds (such as buttons, lists, text fields). The developers of the medical software application claimed overall significant time saving over the original development, as shown in Figure 6.4. The largest benefits were, predictably, concentrated in the testing phase (the potential benefit of verification and validation via SPIN was not addressed in this project since that feature was not yet supported).

We consider this a good indication of VEG potential, since the redesigned GUI was fairly efficient and was developed quickly. However, the comparison with the original development must be used with some caution: the original data were collected from previous documentation of unknown quality, and it is usually easier to redesign than to design, even when different people are involved.

Total size of application mm	24,00			
	Original mm	VEG mm	SAVING %	SAVING mm
Interface (15% of total)	3,60	2,48	31%	1,03
requirements(30%)	1,08	0,86	20%	0,22
development(50%)	1,80	1,35	25%	0,45
testing(20%)	0,72	0,36	50%	0,36

Fig. 6.4 Experimental data for the medical software application.

6.3 Run time architecture

Each model of a VEG specification corresponds to a Java class defining a LL(1) integrated parser/analyzer. The ANTLR compiler-compiler [ANTLR], based on a LL(1) parser generator, is used to generate Java code. The Java library defining the semantic functions is statically linked to the final application. When a model is instantiated, a new parser/analyzer instance is created. During the execution of the instance, attributes and semantic functions are

evaluated and executed, following traditional techniques for L-attributed grammars [Katayama 1984]. Look-ahead sets are computed in advance and may be used for enabling/disabling of widgets (see section 2.2). The choice of ANTLR, apart from the excellent quality of the tool, is based on its explicit support to attribute definition and evaluation and to the BNF format. Moreover, every VEG module corresponds quite naturally to a parser class of ANTLR, and new VEG objects may be instantiated simply with a call to the Java `new` operator, making the implementation of the `launch` operator straightforward.

All internal communication events, generated by the parsers in response to input events, are collected and processed by a Dispatcher module. Communication events are queued and then sent to the relevant parser(s). According to the operational semantics of VEG, no new input event is accepted until the parsers reach a stable configuration (i.e., one when no communication event is still pending).

The link between input events and low-level events (such as pressing a mouse button) is currently implemented by a Visual Platform Editor, which defines a suitable listener for each widget in the actual layout. Each listener basically defines a mapping from low-level events (or even sequences thereof) into VEG input events. The code for the layout is then automatically linked with the parser/analyzer code.

6.4 Plug and Play

Plug and Play is the possibility of run-time extension of a GUI with a new component. Given a VEG application, the GUI development tool is able to provide two objects: an executable file that includes the run-time support, or an Application Package that can be included in an existing VEG run-time architecture. The Application Package consists of Java classes in the format of `.class` files, implementing the parsers and the Semantic Library specific to the package, and of a translation table, mapping platform events to input events. Also, the Application Package includes the Application Interface, i.e., a list of the communication events accepted and generated by the application. To be able to include a new package, the components of VEG run-time support should be defined as containers of objects, and should include only code that is independent of the particular application. The current VEG toolkit, however, gives no support to plug and play, because of its prototypal nature.

6.5 Other features of the VEG notation

VEG has a few more features, which we briefly summarize here, that are not currently supported by the VEG toolkit and are not included in the description of syntax and semantics of Appendix A.

6.5.1 *Reset and quit*

There are two special predefined states, called *reset* and *quit*. When the next state is *reset*, the control must return to the initial state, and also all data values are reinitialized. If the object entering the *reset* state is a container, then by default a special *!reset* event is sent to each member of the group. When the next state is *quit*, then the object is terminated. If the object is a container, by default the special event *!quit* is sent to each member of the group. The *?reset* and *?quit* events are by default handled by default productions, which force the receiving object to enter the states *reset* and *quit*, respectively. The default behaviors of the *reset* and *quit* states and of the default productions for *?reset* and *?quit* may be overridden by defining ad hoc rules.

6.5.2 *Timeouts and Delays*

VEG has also mechanisms to specify delays and timeout, for instance to describe the behavior of the “timed help bullet”: a bullet appearing next to a widget only after the mouse pointer has been continuously over the button for

some time. Delays and timeouts may be modeled in VEG with a special input event called `<timer>`, which takes a delay as an argument. Thus, `<timer 20>` means a delay of 20 units. For instance, the VEG rule:

```
state0 ::=  <activate> \doit1 state1
           | ?got_it \doit2 state2
           | <timer 20> \doit3 state3
```

specifies the following behavior. When entering state `state0`, the object starts a clock. If any event (either an input or an communication event) is received before the end of the delay (here, 20 units), then the action corresponding to the event is taken: either `doit1` or `doit2` are executed and the object changes to `state1` or `state2`. If, instead, no event is received before the timeout expires, then the behavior continues as if a `<timer>` event is received: action `doit3` is executed and control goes to `state3`. A timeout event is considered to be served only when the corresponding alternative starts its execution.

6.5.3 Modal and Hidden Dialogs

A dialog is modal when it has priority over the other active dialogs. For instance, a window notifying an error must block the execution of the other components of the GUI until the user clicks the “Ok” button. Modal dialogs may also be specified in VEG, by using the keyword *Modal model*. To improve performance, also *hidden* models may be specified, i.e., models whose creation and destruction is only apparent. An instance of a model declared hidden is created when its container GUI starts its execution. However, it stays hidden (i.e., invisible) until the instance is launched. It becomes again invisible when the instance is terminated, staying ready to be launched again. It is actually destroyed only when the application ends. This technique is often used in GUI design since it may have a considerable effect on performance.

7. SUITABILITY OF VEG FOR OTHER INTERACTION STYLES

Our hypothesis that the specification of interaction is independent of the layout of the visual interface offers potential application to spoken and multi-modal dialog management, a promising channel for communicating via mobile and portable terminals such as cell phones. Here we only report initial investigation on the potential of VEG for modeling spoken and multi-modal man-machine dialogs. Some more reporting on other interaction styles may be found in the Conclusions.

Dialog management systems (DMS) have a long history behind them, and encompass conceptual and technical knowledge from several sources: speech recognition/synthesis, grammars, theory of speech acts, AI plan based approach, etc. Surveys of the state of the art are in Cole et al. [1996] and in Churcher et al. [1997] who provide the frame for our discussion. In spoken DMS, two aspects have to be managed: how user sentences (or lesser units) are interpreted, and the strategy to perform the intended task, e.g., to make a travel reservation. Multi-modal DMS that combine widget-based and spoken interaction will be considered at the end.

The typical building blocks of a spoken DMS are: *speech recognition*, *meaning extraction*, *response generation*, and *speech output*. In addition, the *application* block exports the methods which are specific of, say, a travel reservation service. The *dialog manager* block (DM) coordinates the other blocks. In some systems the dialog strategy is localized into another block, the *task manager*.

Recognized speech is transcribed, parsed and semantically interpreted. Next the DM matches the interpretation with the expectations of the application, and produces a written reply (a prompt), in accordance with a predefined strategy, via the response generation block. The speech output block synthesizes the speech. Since speech

recognition errors as well as other forms of misunderstanding often occur, a recovery strategy is required, allowing the conversation to continue.

Low-end voice dialog systems are very simple and constrain the user to choose the reply from a small set of single words. They can be readily specified within the framework of VoiceXML 2.0 [Larson 2003], a proposal for implementing conversational voice Web applications that can be accessed by tele- or cell-phones. Their logical structure is essentially a tree of multiple-choice questions that is easy to model in VEG.

It is more interesting to consider systems that allow more sophisticated dialogs. However, if dialogs are more complex than a single question followed by a system response, discourse phenomena (e.g., ambiguity, anaphora, ellipsis) complicate the interaction. These more natural dialogs require advanced language analysis methods, which are beyond the scope of VEG approach. In the middle between these two extremes, we focus on dialogs where user requirements are spread over more than one turn and a user utterance may consist of more than one prompted word.

To map such organization onto the VEG model we depart from generality and consider a specific, typical case, the DMS design approach of McGlashan [1996], which has been applied to several reservation and inquiry services. Its principles are:

1. Only the system's goals are represented in the dialog model: user utterances are not assigned dialog acts.
2. Only local transitions are modeled: the dialog as a whole is not modeled, but global structure can still emerge.
3. Task level information in user utterances is assigned a semantic function indicating its effect on the accessible part of the discourse model.

These functions are applied to goals in the current dialog model: they may satisfy a goal, modify it, or introduce another one. The results are then evaluated to find which goal makes an optimal continuation of the dialog, and the system reports these goals to the user.

We are going to examine the analogies, but also the important differences, between spoken dialogs and GUIs. For instance, in the former case the initiative is taken by the system which has a goal to accomplish, and recursively pursues some sub-goals: leaving the initiative to the user, as in GUIs, would present the risk of losing control, because of the difficulty of interpretation of free speech. Consider an example of a user request, the completion of a query to a flight data-base: this goal is recursively decomposed into goals of various types, such as open the dialog, seek information, seek information via spelling, check information, give information, explain behavior, force termination, close the dialog. When a goal is active the system initiates by prompting the user, then processes the user utterance. Prompting is determined by the strategy and by the state of the dialog: for instance the confirmation strategy may consist of a Y/N answer or of a spelling answer. The utterance is transcribed into a text by the recognizer; to increase recognition rate, the recognizer may be guided by supplying the list of words which are acceptable in this dialog state. The transcription is passed to the meaning extractor. The difficulty and uncertainty of this task for free speech exceeds the possibility of the typical user interfaces we are considering, and would require sophisticated AI techniques. More pragmatically, a careful choice of system prompts can restrict the user language and overcome the shortcomings in speech recognition. The semantic interpretation is matched against the current open goals, and determines their evolution in accordance with the strategy. For instance, after a prompt for seeking information, depending on the recognized transcript, a goal of confirmation (by Y/N or by spelling) may be activated, or a goal of behavior explanation.

In essence, in this design approach, the user turn is a data-entering action, where the data-type depends on the current goal. Examples are Y/N answers, spelling of a city name, or more complex queries consisting of such as aggregate: <query=arrivaltime, from=Stockholm, company= BA, flightid=BA777>. In fact, a user turn is the analogue of a conceptual widget. For instance, a user turn after a prompt for entering a city name can be simulated by a menu-like widget that offers the list of all cities, plus an unknown item for unrecognized names. In the same way, a user turn returning aggregate information may be simulated by multiple list menus.

Thus in substance, user turns can be modeled by a set of VEG models akin to our early login example. Each model matches a certain state of the dialog that is a certain set of active goals.

After a user turn, the recognized item is assigned to a VEG semantic variable (or to an aggregate variable). Then, the dialog manager is the analogue of a VEG controller model; it activates the relevant user turn models, by sending them communication events. In order to choose the models to activate, the manager interrogates the application (typically a data-base server) and makes use of the dialog strategy. After utterance recognition, the user turn models are disabled.

Refinements are needed to cope with uncertainty in voice recognition and meaning extraction, as well as with user misunderstanding the prompts, or supplying inconsistent, incomplete or redundant information. To simulate this behavior in VEG, we propose to return, in addition to the selected menu-like items, a confidence number which can condition the choice of the next goal. For instance, a low confidence would trigger a confirmation goal, by Y/N answer or by spelling. Moreover, to avoid dialog drifting into unproductive loops, the dialog manager may count how many times a goal has been activated. Over a threshold, the manager may move to an explanatory goal (to tell the user what information is missing or inconsistent), or force termination.

In the above description we did not discuss how dialog strategies should be incorporated into the manager. In some dialog design approaches the strategies are encapsulated into a separate block, while in others the strategy is wired into the manager. Using VEG, both ways are possible: the manager model can interact via communication events with a separate strategy model; or the manager rules may directly implement the decision logic of the intended strategy.

Comparing spoken and visual dialogs, we find the former to be less predictable and more subject to user errors. Extensive use of default rules should allow sufficient control over dialog. Semantic variables may be associated with communication events, allowing the manager to send the list of relevant words together with the enabling signal, to the user turn models.

Finally, we briefly consider *multi-modal dialog systems*, taking again an example from McGlashan.

In addition to speech and language component the multi-modal dialog system is composed of a direct manipulation interface which provides graphical information and widgets for navigation, an animated face whose speech is synchronized with the lips movement. A dialog manager is required to coordinate interpretation and generation in both modes. According to McGlashan, the manager differs from a pure voice DM in the following respects: non-speech I/O, referencing visual objects, modality selection, navigation and filtering.

Information carried by the two modalities should be interchangeable: a user may provide input via buttons and the system speak in reply; or the user may refer in speech to a visual object, e.g., by saying “OK, print *them* all”. The last case must be dealt with by finding compatible accessible concepts (say a timetable) the pronoun “them” refers to. This requires rather well established linguistic techniques.

Often, interaction may require fusion of different modalities (e.g., by saying “OK, print *these*” and then selecting the desired objects with a few mouse clicks), as in systems first introduced in [Bolt 1980] and studied more recently by Nigay and Coutaz [1995].

Limiting our consideration to the use of VEG, we do not see any special difficulty arising from multi-modality, but rather the contrary. The VEG approach is suitable to multi-modal dialogs by the central assumption of independence of the layout of the visual interface. For instance, modality selection in system turns is dictated by the characteristics of the output information, and the expressiveness and efficiency of the alternative modalities for realizing it. Sometimes both modalities are enabled, for example to present a travel schedule by voice and by a displayed table. Modality selection criteria and fusion can be wired in the dialog manager (or encapsulated into a separate building block).

In conclusion, the VEG approach, although originally not intended for voice or multi-modal interaction, appears to be suitable to specify these modalities. Model checking or other formal methods have apparently never been considered for early validation of dialog managers, while they may provide a great potential contribution. A difficulty we anticipate is that our model checking approach does not consider the semantic variables in the state space to be explored, while most spoken dialogue interfaces make essential use of them (city names, flight identifier, etc.) A solution is partitioning the semantic space into a finite set of cases, corresponding to the dialogue acts and goals: opening, seek information, confirmation, explanation, etc. This would allow the model checker to explore a significant but not overwhelming state space.

8. RELATED WORKS

There are of course many formal notations, other than grammars, which have been widely applied to GUI specification design, at least in academic research. Among those formal methods that have been explicitly tailored towards GUI applications we recall the cited [Bastide and Palanque 1995, Brun 1997, Paterno and Faconti 1992], which have allowed the implementation of successful design toolkits. However, they have the disadvantage that the modeling power of their notation is typically not finite-state, making them harder to verify than VEG. Up to now, GUI validation and verification (V&V) has been addressed mainly by applying testing techniques. A recent example is in Memon et al. [2000], where test cases are generated and then checked with a test oracle. Formal verification methods such as theorem proving and model checking have already been applied to interactive system verification (see, e.g., the review of Campos and Harrison [1997]). For instance HOL, a higher order logic theorem prover, has been used by Bumbulis et al. [1996] to verify user interface specifications. Because of the equivalent expressiveness of the formalism used in the analysis and of the specification language, properties may be studied on the original specification rather than on its approximation. However, theorem proving usually requires complex user interaction to derive the proof, which may be overkill for many properties that could be checked automatically by model checkers. Theorem provers are more adequate when the verification process cannot be automated, as for instance the verification of a user interface specification against its perception by the users [Doherty et al. 2000].

Model checking has also been applied for V&V of GUI, e.g., by Dwyer et al. [1997], where an existing GUI is abstracted into a finite state version that can be checked with Symbolic Model Verifier (SMV) of Clarke et al. [1986]. In Abowd et al. [1995], user interfaces are specified using Action Simulator, a language to specify simple finite state machines, and then translated into the SMV input language, in order to be checked against some Computational Tree Logic (CTL) formulae. The main drawback of this work is the poor expressive power and

extensibility of Action Simulator, since only very small systems can be modeled with this tool. On the other hand, simplicity may also be an advantage in certain cases and some kind of interface simulation is possible. Several other works using model checking to verify user interfaces use the powerful notion Interactor [Faconti and Paternò 1990, Duke and Harison 1993] as the structuring concept of the specification. An Interactor is a component that encapsulates a state, interacts with its environment through events and is capable of rendering its state into some presentation medium. In Paternò [1995], interactors are specified using Lotos. For verification, the specification is translated into a finite state machine and then analyzed using Action-based Temporal Logic (ACTL). In d'Ausbourg et al. [1996], interactors are directly extracted from the GUI and modeled in the synchronous language Lustre, which is also used to verify properties. The first difference of these two approaches with VEG is that they base verification on events, rather than on states, making expressing of invariants harder. The work of Paternò [1995] does not propose an executable version of the specification. This is not the case of d'Ausbourg et al. [1996], but the expressive power of the resulting specification appears to be poor (only the boolean data type is available for communication) and no code generation support is provided. Campos et Harrison [2001] propose to introduce verification with SMV at the early stages of the specification. As in Paternò [1995], they use interactors to structure their specification but they limit the specification to the parts of the system required by the property to be verified. This minimizes the number of states to explore, and permits verification of internal parts of the system together with its interface, but it forbids executability of the specification since only a small part of the application is specified.

In general, these approaches (test oracles, theorem provers or model checkers) have the advantage of being applicable to any GUI (programmed in any language), but they need to build an explicit high-level model of the system: the abstraction to be applied is application-dependent and there is no guarantee of the significance of the verification results. In VEG, we already have a high-level model of the system: the model checker verifies exactly the same specification that will be implemented, giving greater confidence in the analysis. Moreover, the model checker may be applied as a debugging tool to verify and validate a GUI before its implementation takes place, rather than only checking it afterwards. Finally, it is clear that a certain amount of testing of the GUI is always necessary, even after formal verification: in principle, the model checker could also be used to generate test cases (and test oracles) as well Gargantini and Heitmeyer [1999]. VEG shares these advantages of allowing formal verification and automatic code generation with various formal methods, such as the already cited Statecharts and many others. The language Esterel [Berry and Cosserat 1984], in particular, appears suitable for the specification, verification and implementation of human-computer interfaces (e.g., in Feket et al. [1998]). Esterel, however, is a general-purpose language, certainly much richer than VEG, but it is not specialized for GUI development.

Another aspect of VEG is that the specification is layout-independent, i.e., the first version of a GUI can be written without worrying about the layout, and then different layouts can be adopted. This approach is for instance similar to Ball et al. [2000], where interactive services using more media than one channel are considered (e.g., automated teller machine, bank by phone or web-based interfaces). The key principle is that all user interfaces share the same service logic (Sisl: Several Interface Single Logic). The logic is specified in an event based model with a constraint-based language using reactive constraint graph described in XML and translated into Java. The constraint graph may be translated into state machines for automatic test case generation.

VEG shares many features with the Fudget toolkit of Carlsson and Hallgren [1998]. Fudget is a declarative-style, high-level notation, based on a functional programming language, supporting hierarchical design, layout-independence, concurrent programming, extensibility and state encapsulation. These advantages are also present in the VEG toolkit, which in a sense is also based on a functional notation (attribute grammars), but Fudget gives a more limited support to verification and validation.

Software Model Checking (e.g., the Bandera tool of Corbett et al. 2000) is now an important research subject. The main issues in the field are identifying safe abstractions (i.e., an abstraction where all the violation of the desired properties of a given program can always be found on the abstracted version) and eliminating spurious counterexamples (counterexamples for the abstraction only, corresponding to infeasible paths). In VEG, we already have a safe abstraction that is less subject to the problem of spurious counterexamples, at least for event-driven specifications. The semantic part (the Java code) of a specification is however ignored by our analysis, assuming that other techniques, such as traditional testing, are adopted. An interesting approach would be the integration of software model checking, for the semantic parts, into the VEG techniques.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown how to apply grammars to the specification, design, verification and implementation of GUI. Dialogs are specified by means of modular, communicating grammars called VEG (Visual Event Grammars).

A VEG specification is independent of the actual layout of the GUI, but it can be easily integrated with various layout design toolkits. Moreover, a VEG specification may be verified with the model checker SPIN, in order to test its consistency and correctness, to detect deadlocks and unreachable states, and also to generate test cases for validation purposes. Since GUIs are usually event-driven, control-intensive, rather than data-intensive, applications, model checking techniques can be very effective.

Efficient code is automatically generated by the VEG toolkit, based on compiler technology. Realistic applications have been specified, verified and implemented, like a Notepad-style editor, a graph construction library and a large real application to medical software.

One of the major advantages of the VEG approach is the possibility of the interaction of development and step-by-step verification: the “push-button” model-checking facility of our toolkit allows continuous testing of the system under development, also very early in the design phase. Moreover, since many aspects of behavior may be easily described using VEG and combined with the layout long before coding semantic actions, VEG may give strong support to rapid prototyping: a user may be shown not only a traditional GUI mock-up, but rather a working GUI that may include most behavioral aspects.

Our examples emphasized WIMP interaction style, with only some discussion on other input styles, such as voice interaction [Bolt 1980] or multi-modal dialogs, in Section 7. This should not be considered as a limitation but only as a convenient simplification for exposition. More input styles such as virtual reality [Folet 1987] can be modeled in our framework as long as the “numerical” part of the events is considered in the semantics of the model as this already holds in the WIMP model. As an example, the insertion of values with a slider may be decomposed into a syntactic sequence of input events of the type *press-drag-release*, and a sequence of the actual slider values associated with the events. The syntactic sequence may be modeled by state transitions, but the slider values must be passed as arguments to semantic functions. A 3D input widget like an arcball [Shoemake, 1992], is quite similar

to a slider. The dragging process results in a sequence of input events containing the geometric position (a pair of numbers) of the pointer in its semantic attributes. Our model does not manage this numerical information in the syntactic part, and thus could not be used in a verification process. Anyway, this information is unbounded by nature and so it would make verification much harder. It is a challenging problem to design an extension of our model that takes into account, at the syntactic level, changes in numerical values in a satisfactory way. Usual data structures like stacks or queues seem to be too restrictive. On the other hand, if the aim is still to use model checking tools, severe restrictions have to be imposed on the nature of numerical data as we did it already to deal with dynamic instantiation of widgets.

In gesture-based interaction, the design problem is quite similar: again, numerical values associated with the events need to be handled at the semantic level. However, interface changes based on the acquired experience of the behavior of particular users, are based on thresholds, and hence they can easily be modeled by a change of state in VEG.

Future work will consider the application of the method to the design of safety-critical software, in order to exploit the verification capabilities of VEG, but also to handheld devices, to exploit the one interface-many layouts capability of VEG.

ACKNOWLEDGEMENTS

The VEG notation and toolkit is the result of an Esprit Long-term research project, called Gedisac, started in 1998 and completed at the beginning of 2000. Special thanks to Marco Pelucchi, who developed the VEG toolkit and various prototypes, first as a graduate student and later while working at Txt e-solutions. We gratefully acknowledge the contributions of many people to the development of the VEG ideas and toolkit. Among them, Alberta Bertin, Txt e-solutions, Fabien Lelaquais, Marie Georges and Christian de Sainte-Marie, Ilog, and Alessandro Campi, Politecnico di Milano. We also thank the project reviewers Gorel Hedin, Lund Institute of Technology, and Ian Sommerville, Lancaster University, and the project officers Pierrick Fillon and Michel Lacroix, for their many useful suggestions. Many thanks to Eliseo Martinez who has developed the new version of the VEG Visual Editor and provided Figure 6.2, and to Emanuele Mattaboni for developing the Visual Platform Editor. We also thank the anonymous reviewers of an earlier version of this paper for their constructive feedback and many suggestions to improve the manuscript.

REFERENCES

- ABOWD, G. D., WANG, H.-M. AND MONK, A. F. 1995. A formal technique for automated dialogue development. In *Proceedings of the First Symposium of Designing Interactive Systems - DIS'95*, August 1995. ACM Press, New York, 219–226.
- ANTLR Web Site: Complete Language Translation Solutions, <http://www.antlr.org>.
- BALL, T. AND RAJAMANI, S.K. 2001. The SLAM Toolkit, *13th Conference on Computer Aided Verification, CAV2001*, July 18-23, 2001, Paris, France, LNCS 2102, Springer Verlag, 260-264.
- BALL, T., COLBY, C., DANIELSEN, P., JAGADEESAN, L., JAGADEESAN, R., LAUFER, K., MATAGA, P. and REHOR, K. 2000. Sisl: Several Interfaces, Single Logic. *International Journal of Speech Technology*, 3(2): 93-108.
- BASTIDE, R. AND PALANQUE, P. 1995 A Petri Net Based Environment for the Design of Event-Driven Interfaces, In *16th Int. Conference on Application and Theory of Petri Nets (ATPN'95)* Torino, Italy, 20-22 June 1995. De Michelis, G. Diaz, M. Eds. LNCS 935, Springer Verlag, 66-83.
- BERRY, G. AND COSSERAT, L. 1984. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, ROSCOE, A.W., WINSKEL, G. AND BROOKES, S.D. Eds. LNCS 197, Springer-Verlag, 389-448.
- BERSTEL, J. 1979. *Rational Transductions and Context-Free Languages*. B. G. Teubner, Stuttgart.
- BERSTEL, J., CRESPI REGHIZZI, S., ROUSSEL AND SAN PIETRO, P. 2001. A Scalable Formal Method for Design and Automatic Checking of User Interfaces. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Ontario, Canada, May 12-19, IEEE Press, New York, 453-462.
- BOLT, R. A. 1980. Put-that-there: Voice and gesture at the graphics interface. *ACM Computer Graphics*, 14(3):262–270.
- BRUN, P. 1997. XTL: A Temporal Logic for the Formal Development of Interactive Systems. In *Formal Methods In Human-Computer Interaction*, PALANQUE, P. AND PATERNO, F. Eds. Springer Verlag, 121-139.

- BUMBULIS, P., ALENCAR, P. S. C., COWAN, D. D. AND LUCENAN, C. J. P. 1996. Validating properties of component-based graphical user interfaces. In *Proceedings of 3rd Eurographics Workshop on Design, Specification and Verification of Interactive Systems* June 1996, BODART, F. AND VANDERDONCKT, J. Eds. Springer-Verlag, New York, 347–365.
- CAMPI, A. 2000. *Design and Verification of GUIs by syntactical methods* (in italian). Master Thesis, Politecnico di Milano, Italy, Dec 2000.
- CAMPOS, J. C. AND HARRISON, M. D. 1997. Formally Verifying Interactive Systems: A Review. In *Proceedings of 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems* June 1997, HARRISON, M. D. AND TORRES, J. C., Eds. Springer-Verlag, New York, 109–124.
- CAMPOS, J. C. AND HARRISON, M. D. 2001. Model Checking Interactor Specifications. *Automated Software Engineering*, 8(3-4):275–310, (August 2001).
- CARLSSON M. AND HALLGREN T., 1998. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*, Ph.D. Thesis, Computing Science Department, Chalmers University of Technology and University of Göteborg. March 1998.
- CHURCHER, G. E., ATWELL, E. S. AND SOUTER, C., 1997. *Dialogue Management Systems: A Survey and Overview*, Report 97.06, University of Leeds, School of Computer Studies, Leeds, UK.
- CLARKE, E.M., EMERSON, A. AND SISTLA, A. P. 1986. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications, *ACM TOPLAS* 8(2): 244–263 (1986).
- CLARKE, E.M., GRUMBERG, O. AND LONG, D.E. 1994. Model Checking and Abstraction. *ACM TOPLAS* 16(5): 1512 – 1542 (Sep. 1994).
- COLE, R., MARIANI, J., USZKOREIT, H., VARILE, G.B., ZAENEN A., ZAMPOLLI, A. AND ZUE, V. 1996. *Survey of the state of the art in human language technology*. Printed version of Cambridge University Press.
- CORBETT, J., DWYER, M., HATCLIFF, J., PASAREANU, C., LAUBACH, S. AND ZHENG, H. 2000. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, June 4 – 11, 2000, Limerick, Ireland, 439 – 448.
- d'AUSBOURG, B., DURRIEU, G. AND ROCHE, P. 1996. Deriving a formal model of an interactive system from its UIL description in order to verify and to test its behaviour. In *3rd Eurographics Workshop On Design, Specification And Verification Of Interactive Systems* Namur, Belgium, June 5-7 1996. BODART, F. AND VANDERDONCKT, J. Eds., Springer-Verlag, New York, 105–122.
- DOHERTY, G., CAMPOS, J. C. AND HARRISON, M. 2000. Representational reasoning and verifications. *Formal Aspects of Computing*, 12(4), 260–277.
- DUKE, D.J. AND HARRISON, M.D. 1993. Abstract Interaction Objects In *Proc. of EUROGRAPHICS 93*, HUBBOLD, R.J. AND JUAN, R. Eds. In Computer Graphics Forum, Vol. 12, No.3, 26–36.
- DWYER, M. B., CARR, V. AND HINES, L. 1997. Model Checking Graphical User Interfaces Using Abstractions. In *Proc. 6th European Softw. Eng. Conf.*, 244–261.
- FACONTI, G. AND PATERNO, F., 1990. *An Approach to the Formal Specification of the Components of an Interaction*. In *Proc. of EUROGRAPHICS 90*, VANDONI, C. AND DUCE, D. Eds. 481–494.
- FEKETE, J.D., RICHARD, M., AND DRAGICEVIC, P. 1998. Specification and Verification of Interactors: A Tour of Esterel. Presented in *Formal Aspects of Human Computer Interaction Workshop (FAHCI'98)*, Sept 1998, Sheffield Hallam University, Sheffield, UK.
- FOLEY, J. D. 1987. Interfaces for Advanced Computing, *Scientific American*, Vol. 257(4), 127–135.
- FOLEY, J.D., KIM, W.C., KOVACEVIC, S. AND MURRAY, K. 1989. Defining Interfaces at a High Level of Abstraction. *IEEE Software* 6(1): 25–32.
- GARGANTINI, A., AND HEITMEYER, C. 1999. Using Model Checking to Generate Tests from Requirements Specifications. In *Proc., Joint 7th Eur. Software Engineering Conf. and 7th ACM SIGSOFT Intern. Symp. on Foundations of Software Eng. (ESEC/FSE '99)*, Toulouse, France, September 1999, LNCS 1687, Springer Verlag, New York, 146–162.
- GODEFROID, L., JAGADEESAN, J., JAGADEESAN, R., AND LAUFER, K. 2000. Automated systematic testing for constraint-based interactive services. In *Proc. 8th Intl. Symp. on the Foundations of Software Engineering (FSE'2000)*, San Diego, Nov. 2000, 40–49.
- GRAY, W. D., PALANQUE, P., AND PATERNO, F. 1999. Introduction to the special issue on interface issues and designs for safety-critical interactive systems: when there is no room for user error. *ACM Trans. on Computer-Human Interaction*, 6(4): 309–310.
- GREEN, M., 1983. Report on Dialogue Specification Tools. In *Proceedings of the Workshop on User Interface Management Systems*, Seeheim, Germany, Nov. 1–3, 1983. PFAFF, G. E., Ed. Springer-Verlag.
- HAREL, D. 1987. Statecharts: a visual formalism for complex systems. *Science of Comp. Progr.* 8, 231–274.
- HARTSON, H. R. AND HIX, D., 1989. Human-computer Interface Development: Concepts and Systems for its Management. *ACM Computing Surveys* 21.1 (1989): 5–92.
- HENDRICKSEN, C. S. 1989. Augmented state-transition diagrams for reactive software. *ACM SIGSOFT Software Engineering Notes*, Vol. 14(6), 61 – 67.
- HILL, R. D. 1986. Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction—the Sassafra UIMS. *ACM Transaction on Graphics*, 5(3):179–210.
- HOLZMANN, G. J. 1997. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), 279–295.
- JACOB, R. J. K. 1982. Using formal specifications in the design of a human-computer interface. In *Proc. of the 1982 Conference on Human Factors in Computer Systems*, Gaithersburg, Maryland, United States. ACM press, New York, 315–321.
- KATAYAMA, T. 1984. Translation of attribute grammars into procedures, *ACM Trans. on Programming Languages and Systems*, 6(3), 345–369.
- KNUTH, D. E. 1968. Semantics of context-free languages, *Mathematical Systems Theory*, 2(2), 127–145. Correction in 1971: *Mathematical Systems Theory* 5(1), 95–96.
- LARSON, J.A. 2003. *VoiceXML: Introduction to Developing Speech Applications*, Prentice Hall, 2003.
- MCGLASHAN, S. 1996. Towards Multimodal Dialogue Management. In *Proceedings of 11th Twente Workshop on Language Technology 11*, Enschede, The Netherlands.
- MEMON, A., POLLACK, M., AND SOFFA, M. L. 2000. Automated Test Oracles for GUIs, In *Proc. Eighth International Symposium on the Foundations of Software Engineering (FSE 2000)*, San Diego, CA, Nov. 6–10 2000, 30–39.
- NIGAY, L. AND COUTAZ, J. 1995. A generic platform for addressing the multimodal challenge. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, Denver, Colorado, United States. ACM Press, New York, 98 – 105.
- NORRIS IP, C. AND DILL, D. 1996. Better Verification Through Symmetry. *Formal Methods in System Design* 9, 1/2, 41–75.
- NYMEYER, A. 1995. A grammatical specification of human-computer dialog, *Computer Languages*, 21(1):1–16.
- OLSEN, D. R. Jr. 1983. *Presentational, Syntactic and Semantic Components of Interactive Dialogue Specifications*. In *Proceedings of the Workshop on User Interface Management Systems*, Seeheim, Germany, Nov. 1–3, 1983, PFAFF, G.E., Ed. Springer-Verlag, Vienna, 1985.
- OLSEN, D. R. Jr. 1984. Pushdown automata for user interface management. *ACM Transactions on Graphics*, 3(3):177–203.
- PALANQUE, P. AND PATERNO, F. 1997, Eds. *Formal Methods In Human-Computer Interaction*, Springer Verlag, December 1997.

- PATERNÒ, F. AND FACONTI, G. 1992. On the Use of LOTOS to Describe Graphical Interaction. In *People and Computers VII: Proceedings of the HCI'92 Conference*, Cambridge University Press, 155-173.
- PATERNÒ, F., *A Method for Formal Specification and Verification of Interactive Systems*, PhD thesis, Department of Computer Science, University of York, 1995.
- PETERSON, J.L. 1981. *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliff, NJ.
- REISNER, P. 1981. Formal Grammar and human factor design of an interactive graphics system. *IEEE Trans. on Software Engineering*, 7(2), 229-240.
- SHNEIDERMAN, B. 1982. Multiparty grammars and related features for defining interactive systems. *IEEE Trans. Syst. Man Cyber.*, 12(2), 148-154.
- SHNEIDERMAN, B. 1997. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 3rd edition (July 1997), Addison-Wesley.
- SHOEMAKE, K. 1992. ARCBALL: A user interface for specifying three-dimensional orientation using a mouse. In *Proceedings of Graphics Interface '92 Canadian Annual Conference*, 151-156.
- VAN DEN BOSS, J. 1988. Abstract interaction tool: a language for user-interface management systems. *ACM Trans Prog. Lang Syst.*, Vol. 10, 215-247.