A Scalable Formal Method for Design and Automatic Checking of User Interfaces

Jean Berstel[+] Stefano Crespi Reghizzi[*] Gilles Roussel[+] Pierluigi San Pietro[*]

[*]*Politecnico di Milano*
Dipartimento di Elettronica E Informazione
P.za Leonardo da Vinci, 32
20133 Milano, Italia
+39 02 23993405
{crespi,sanpietr}@elet.polimi.it

[+]*Institut Gaspard Monge*
Université de Marne-la-Vallée
5, Bd Descartes
77454 Marne-la-Vallée Cedex 2, France
+33 1 609 575 57
First.Last@univ-mlv.fr

ABSTRACT

The paper addresses the formal specification, design and implementation of the behavioral component of graphical user interfaces. Dialogs are specified by means of modular, communicating grammars called VEG (Visual Event Grammars), which extend traditional BNF grammars to make the modeling of dialogs more convenient.

A VEG specification is independent of the actual layout of the GUI, but it can be easily integrated with various layout design toolkits. The 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. The complete VEG toolkit is going to be available soon as free software.

Keywords: Formal methods, Human-computer interaction (HCI), Applications of model checking, GUI design.

1 INTRODUCTION

Current industrial practice for designing graphical user interfaces (GUI) uses toolkits and interface builders, usually based on visual programming languages, for producing the layout. These tools allow a simple and quick description of the geometric display, and frequently give some support for designing interaction of components. However, the dialog control must be hand-coded with conventional programming techniques and there is no support, other than testing, for checking its logic.

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. Hence, traditional techniques may be costly and inadequate to build complex GUI.

Formal techniques may allow one to perform systematically, or even automatically, validation and verification activities like testing, simulation and model checking [4] and to prove that the modeled systems possess desired properties. Hence, the validity of the design may be assessed before the development phase takes place.

Many formal methods have been proposed for GUI design [12], such as transition diagrams, Petri Nets [1], formal grammars [14], process algebras [13], temporal logic [3]. However, most methods get unwieldy as the system complexity grows (i.e., they are beneficial only for small systems or single components). Often they are not amenable for automatic verification techniques and only support simulation and testing. As pointed out by Shneiderman [16, p. 159]: “Scalable formal methods and automatic checking of user interface features would be a major contribution”.

We propose a new method, combining 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 [7]. Each automaton is an object, described by means of a grammar, specifying a small part of the scene, such as a window or a widget. Automata may share common behavior and hence be seen as instances of general models. The automata interact by sending and receiving communication events in order to realize the expected global behavior.

The VEG approach allows the automatic generation of efficient code from the specification of the interface, and its integration with commercial design tools. The various automata are implemented with 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.

Apart from the merits of individual notations, one of the major obstacles in the industrial spreading of formal methods 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 experts only. Advances in automatic verification techniques, such as model checking, may change this state of the affairs, making in principle "pushbutton" verification possible for various systems specified with automata.

In general, however, a specification or a program has to be "abstracted" to be amenable for automatic verification with tools such as model checkers. In fact, the number of states of even simple programs is usually by far too high for model checkers. Abstractions are hard to obtain and there is no guarantee that the verification results for the abstracted system are meaningful for the original one. In the VEG toolkit, however, the abstracted version can be derived automatically from the original specification and its meaning is very close to the original one. In various cases, the abstracted specification is exactly the one used to produce the application: the uniqueness of the specification insures the coherence between the application and its formal model. In our experiments, we found that even very large GUI may be easily checked, since usually the number of their states is much smaller than the current limits of model checking technology.

Verification and validation activities in VEG are based on Spin [8], 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 simulation and test case generation. The support for verification in VEG may also help in checking features of an interface and in detecting requirement errors. For instance, a Quit button in an Editor application should be reachable from every state. This means that the GUI will never run into a configuration where a user will no longer be allowed to quit. 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 you can write into it. Also this kind of (temporal) properties can be easily verified with Spin.

The paper is organized as follows: Section 2 introduces the basic VEG notation, Section 3 extends the notation, Section 4 shows an example of modular design, Section 5 illustrates automatic verification in VEG and Section 6 reports on the implementation. Finally, Section 7 describes related works and Section 8 draws a few conclusions.

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 graphical components (see Fig. 2.1): the central component is a small widget representing a juggler while the other two are buttons, labeled Go and Stop respectively, to control the juggler's behavior. When the window is started, the juggler is idle. When the Go 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 button (represented by the small cross at the topmost, right-hand corner of the window) to destroy the scene.

Fig. 2.1: A Juggler

A simple VEG specification of a Juggler window is the following one:

```
Model SimpleJuggler
   Axioms start
   start ::= \createScene idle
   idle ::= <go> \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[1], and it is composed of a set of *grammar rules*. Each rule may have a left-hand side: a variable name (also called a *state* or a nonterminal symbol), such as *start*, *idle* or *juggling.* Each rule always has a right hand-side, describing the behavior of the model when in that state. Rules with an empty left-hand side are called *ubiquitous* rule and may be applied in every state.

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

An input event is actually an abstraction of low-level events or even low-level event sequences, as they already exist in Motif or Java. For instance, the <*quit*> event models reception of a closing message received from the window manager, while <*go*> 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 <*go*> is not explicitly connected to a left-click of the mouse on a PushButton labeled Go. 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 <*go*> and <*stop*> input events could have been synthesized from button activation, menu selection or keystrokes shortcuts. In fact, the precise aspect of a "button" of 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 could be activated, disabled and enabled. Examples of these components for the Go and Stop "buttons" are Pushbuttons and Menu entries. Since the same VEG specification is compatible with many different layouts and portable on many 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.

Visual actions are the GUI responses in the interaction with the user. For example, the visible action *createScene* is used to create visual objects of the platform. Visual actions correspond to suitable callbacks, defined in a Semantic Library, and they can 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.

2.1.1 Models, instances, visual objects, creation and destruction of scenes. Each model must be considered akin to a class declaration in object-oriented programming languages. Hence, it must be instantiated before use. Each instance, also called a *VEG object*, has a name and its own state, but obeys the same rules of behavior.

The semantics of a model such as the simple Juggler is as follows: when an instance of the model is in some state, and some input event is triggered, then the corresponding rule is considered. Visual actions are performed and the state changes to the next state of the rule. The initial state is the axiom specified at 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 Enabling and Disabling Widgets. In the Juggler example, the *go* 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 *go* 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. Nymeyer [9] 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.

Two general 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. Shading, as done by Nymeier, is precisely a visual counterpart of the disabling. Instead of disabling, more general actions can take place 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 shade the component or raise a

[1] 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 5.

warning, etc. (see for instance [1]).

2.1.3 BNF format and syntax-diagrams. The textual notation of formal grammars that was shown here is introduced only for explanation purposes, but it is not always very convenient. Actually, there is a prototype of a visual tool for entering and editing formal grammars represented with a form of syntax diagrams. Internally, grammars are represented by an XML document, which of course is not expected to be directly edited or read.

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*, which is convenient for automatic verification:

1) the right-hand side of every rule is composed of one or more alternatives;
2) for each alternative there is at most one nonterminal, called the *next state,* which is at the rightmost position in the alternative.

For instance, a rule to describe that in a state *ready* it is possible to receive either a <go> or a <stop> and then to go back to the ready state again can be written as:

```
ready ::=       <go> \startJuggling ready
        | <stop> \stopJuggling ready
```

2.2 A Refined View on the Example

The aim of this section is to introduce and illustrate some of the other VEG concepts, mainly modularity, 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 naming scopes, 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. Parallel composition has an interleaving semantics: when a component is activated, the other components are not allowed to collect events until the first one has completed a state transition.

To illustrate these features, we show how the Juggler specification could be broken down into modules, by specifying separately the control of each component. The level of abstraction is lowered for the purpose of explanation of our model, by means of explicit modeling of each visual component of the juggler. This also shows that our tool allows GUI designers to consider different levels of abstraction, even in the same specification.

A model called Activator is in charge of controlling a button. 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. The model can be described in VEG with the following specification, that 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 possibly by the source name, and always by the name of the event, such as in *?enable*. Output communication events (i.e., emitted messages) start with an exclamation mark, may contain the destination followed by the name of the event, such as *!activated*. The (optional) name of the source or of the destination may also be a parameter whose actual value is specified at creation time, as shown in the Activator of Section 4.

The juggler can be described by a model as follows:

```
Model Juggler
        Axioms idle
        idle ::= ?go.activated \startJuggling !stop.enable
juggling
        juggling::= ?stop.activated \stopJuggling
!go.enable idle
End Juggler
```

A juggler, when idle, expects to receive a communication event *?go.activated*, i.e., an event *activated* coming from a SimpleActivator called *go*. Next it starts juggling and sends an event *enable* to a SimpleActivator called *stop*, going to the state *juggling*. The definitions of *go* 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 *go* and becomes idle again. Notice that the ability of specifying the source of communication events may avoid name clashes of events among different models.

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(
                 juggler = Juggler.idle,
                 go = SimpleActivator.enabled,
                 stop = SimpleActivator.disabled)
          running
running ::=<quit> \destroyScene
End Box
```

The *launch* operator is used to denote parallel composition: some children processes are named and started in the specified state. In the example, an instance (called *juggler*) of the Juggler is created and initialized in the *idle* state, and two instances (*go* and *stop*) of SimpleActivator in the *enabled* state and in the *disabled* state, respectively.

In this example, the Box is only a *container* without interactions, except for the event <*quit*>, in charge of launching the other components and of initializing the scene. In other cases, 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...
End JugglerPackage
```

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

2.3 Containers and Groups

The *children* of a VEG object, at a given instant, are defined as the components launched by the object and which did not terminate yet their execution. When a VEG object has children, it is called a *container*. Some special VEG constructs may be used to simplify communication among the children and between the container and its children. A container can multicast 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, it is possible to keep track of one or more sets of selected children. Thus, a container may also broadcast communication events to a selected set *x* of children, called a *group,* or to its complement ~*x*. This may be accomplished by means of a set of predefined semantic actions and predicates, called *group selection mechanisms*.

The action */AddTo(x)* adds, to the group *x,* the child that sent the latest communication event to the container, while the action */RemoveFrom(x)*, removes it from the group *x*, if present. The semantic predicate *isEmpty* may be used to check whether a group is empty.

For instance, the fragment *?selected /addTo(s) !s.enable !~s.disable* first waits for an event called *selected.* When a child C sends the event, C is added to the group *s*. Then an event *enable* is sent to all the children in the group *s,* and hence also to C, and an event *disable* is sent to the remaining children.

It should be noticed that group selection features, while being realized with semantic actions and predicates, are essentially finite state, and hence do not hamper the possibility of automatic verification of VEG specifications (as long as the number of children is bounded). In fact, they are useful shorthands to avoid an increase of the rules and states of a model.

3 MANAGING SEMANTIC ACTIONS

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. These kind of utility routines are of course independent from a GUI design, but the result of a routine may influence the behavior to a significant amount. Semantic *functions* are designed for modeling this role. They are collected in a so-called Semantic Library and are written in a programming language such as C, C++ or Java.

A *semantic function* has some *variables* (or semantic attributes) as arguments and computes a value as result. Values can be associated with input events, states and communication events and passed to visible actions and other semantic functions. 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. For communication events, values may be used to interface the models in a parallel composition.

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

Semantic functions are divided into semantic *predicates* (i.e., boolean functions) and semantic *actions* (the other functions). Semantic predicates can be used as the guard in a syntactic alternative of a production, as shown next, and hence may have a direct effect on the behavior of a GUI.

3.1 Example of Attribute Specification

A *Login Session* is a dialog where a user is required to enter a username and a password 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.

It is possible to add variables and their definitions to the VEG notation. The result is an attribute grammar-style declarative specification, which does not necessarily imply an ordering of variable evaluation.

A VEG specification of the Login Session may be the following one, where pushing the button labeled Ok after filling the two textfields is modeled with an input event *<enterData>*. The actual values of the user name and of the password are two string attributes *userName* and *passwd* of the state *login.* An integer attribute called *n* keeps track of the number of login attempts. The parts that are related to the attributes are italicized.

```
Model LoginSession
        start  ::= \createScene login
                    n.login := 0;
        login  ::= <enterData> verify
                    n.verify := increment(n.login);
        verify  ::=
        /if_loginCorrect(userName.enterData,passwd.ente
rData)
        launch(session = Session.start)
    /elsif_notTooMany(n.login) \message("Retry") login
                    n.login := n.verify;
        /else  \message("Login Failed") \destroyScene
End LoginSession
```

We assume that there is one model called *Session*, which is launched in the state *start*. The semantic predicates *loginCorrect* and *notTooMany* are used to discriminate, by means of a */if.../elsif.. /else* construct, among the various alternatives of the rule for *verify*. When the functions *loginCorrect*, *notTooMany*, *increment,* to be included in a Semantic Library, are properly implemented in a existing language such as Java, the VEG toolkit is able to generate an integrated LL(1) parser/semantic analyzer.

4 HIERARCHICAL DESIGN OF SCALABLE COMPONENTS

This section shows how to combine some VEG constructs for modular design of graphical interfaces, to obtain a hierarchical description of visual communicating modules.

Each component of a typical GUI offers a set of options, or choices, to the user, and in addition some tools for entering numerical of alphabetical data. At the top level, there is frequently a menu bar, providing a set of menus. Each of the menus contains entries that may themselves be menus. Observe that these menus or menu entries are not independent: a menu may be disabled (or even be absent) depending on the current state of the GUI, triggered by choices made in other menus.

This hierarchical organization admits many visual variations: option panes may be used in order to gather sets of choices, sets of options may be shown as lists and numerical values may be entered with scales or scrollbars.

We illustrate the modular design in VEG of these typical GUIs by means of the overall description of a simple text editor. This is useful to show how to describe menus with interacting items, where each item should be viewed as a menu itself.

The interaction with the text is done with the menu entries (or similar components, such as buttons in toolbars, accelerators or contextual menus). Each entry triggers an action that is responsible for achieving some task. Moreover, some of the entries (and thus of the tasks) may be interdependent. A typical example is the standard File Menu (where "Save", "Save As", "Close" are entries that are not always activated) or the Edit Menu.

It is important, for modularity, that the communication is between the menu and its items, and not directly between items, else it is hard to extend or revise the menus. In this sense, a menu is a container, and its elements are the menu items. Communication is between the container and its items; the container is in charge of the communication with other containers and/or with the text component.

The 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 when in any 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
                        ::= ?client.disable disabled
                    | ?client.enable enabled
End Activator
```

A ubiquitous 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.).

As an example of modular design, consider a simple Edit Menu, composed of the entries Cut, Copy, Paste. At the beginning, none of these entries is enabled. A Cut or a Copy operation requires that some text is selected. A Paste requires that a Cut or a Copy operation have been executed before. Each operation makes an action on the text (doCut, doCopy, doPaste), which is not described in VEG. Each entry of the menu communicates with the menu itself, signaling its activation and receiving the enabling/disabling

commands, while the menu communicates with the text component. After launching the various entries, the EditMenu may be in one of four states: *running*, waiting for some text to be selected, while the three entries cut, copy and paste are disabled; *cutcopy*, when some text has been selected, with cut and copy enabled; *paste*, when the paste entry is enabled and the others disabled (some text was cut or copied but currently it is not selected); *full*, when all the entries are enabled (some text was cut or copied and the text is currently selected).

```
Package EditMenu;
Import BasicComponents
Model EditMenu
   start ::= launch(
       cut = Activator(EditMenu).disabled,
       copy = Activator(EditMenu).disabled,
       paste = Activator(EditMenu).disabled)
       running

   running ::= ?Text.select !cut.enable !copy.enable
                  cutcopy

   cutcopy ::= ?Text.select cutcopy
       | ?Text.deselect !cut.disable !copy.disable running
       | ?cut.activated !Text.doCut !cut.disable
                  !copy.disable !paste.enable paste
       | ?copy.activated !Text.doCopy !copy.enable
       !paste.enable full

   paste ::= ?Text.select !cut.enable !copy.enable full
             | ?paste.activated !Text.doPaste paste

   full ::=        ?Text.select full
     | ?Text.deselect !cut.disable !copy.disable paste
     | ?cut.activated !Text.doCut
                  !cut.disable !copy.disable paste
     | ?copy.activated !Text.doCopy  !copy.enable full
     | ?paste.activated !Text.doPaste
                  !cut.disable !copy.disable paste
End EditMenu
```

The EditMenu communicates with the Text, which for brevity is not described here. The interaction among menu entries may similarly be described for the usual File menu. Dialog boxes with lists of choices (such as a Format specification box) may also be managed in a similar way.

5 AUTOMATIC VERIFICATION

A primary goal of verification activities is to ensure that a specification is consistent, i.e., it handles every combination of states and events that may occur.

Consistency may take various aspects. In particular, it implies that, given some state of a model, there should be a rule that is applicable for every event that may be received by the model in this state. If this rule is missing, then some situation has been forgotten by the designer, such as the case of a component sending an event that is never consumed. The run-time effect of such an error is that the application is blocked, i.e., a *deadlock* occurs. Thus, deadlock detection is a major goal to be achieved.

Consistency also means that the specification does not contain states that are *unreachable*, reflecting design concepts that are unusable, such as the case of a component waiting for an event that will never be sent.

Deadlock-free specifications may still have logical errors, because they may behave different from what was intended, such as the case of a login session where no password identification is done. This kind of error may be detected by validation activities, such as simulation, animation, and verification of (temporal) logic properties, which are also possible with the VEG toolkit.

Model checking [4,8] is a powerful and useful technique, allowing the automated verification that a finite state machine verifies a given property, such as deadlock-freeness, usually specified in a temporal logic language. Model checking is thus the ideal tool to verify consistency, 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: the number of possible states of the system must be significantly reduced: even if a model checker may sometime check systems with 10^{60} reachable states or more, this is actually equivalent to less than 200 bits of state space. 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 Spin. In fact, when studying formal features of a user interface, the actual values of the parameters passed to and from the semantics 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. However, the presence of semantic predicates introduces a loss of precision, since they are used as guards of a branch cannot be evaluated anymore: the choice of the branch to be followed becomes nondeterministic.

Moreover, most predefined semantic actions may be translated into the finite state constructs of Promela. For instance, the selection mechanism, which is implemented with predefined semantic actions, is also modeled by our translation. For example, this permits to check that sending an event to an empty selection set does not produce a deadlock. Also other semantic actions, such as the number of attempts in the login dialog may be easily modeled in Promela. On the other hand, user defined semantic actions (ad hoc coded in Java) are ignored by our translation and treated as part of the terminal alphabet. However, the set of predefined semantic actions can be extended to deal with common semantic actions that can be modeled in Promela.

In order to define the translation from VEG to Promela,

a VEG package P can be formalized as a transduction [2] T from an input alphabet of input events to an output alphabet of visible actions and semantic actions. If the transduction T depends on the values of the semantic variables, the transduction T is called *semantically-branching*.

Denote with S the transduction obtained from T by replacing semantic branching (i.e., /if ... /else constructs) with all possible continuations (i.e., replacing the /if .../else constructs with the alternative operator |). T is semantically-branching precisely when $T \neq S$.

Significant information may be automatically obtained for S using verification programs. Some information about T may be deduced in view of the following statement.

Statement 1. The following properties hold:
1) if a state is unreachable in S, then it is unreachable in T;
2) if S is deadlock-free, then T is deadlock-free.

Notice that the converse properties may not hold when $T \neq S$. Also, part (2) of the statement assumes that the semantic and visible actions do not have an internal deadlock themselves (such as a nonterminating code).

Often, S is finite-state (e.g., the Juggler example) and hence it can be verified by using model checking techniques. The number of states of S 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 S, however, may not be finite-state. This is the case when the number of VEG objects in a specification may not be bounded by any integer (e.g., the user is allowed to instantiate any number of copies of a window). In this case, S is called an *unbounded* transduction.

If S is not finite-state, it cannot be verified using standard model checking techniques. In this case, we introduce an 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* S_N of the transduction S. The transduction S_N can be verified by model checking when N is not too large. The verification results for S_N are in most cases extendable to S and to T:

Statement 2. Let S be an unbounded transduction.
1) If a state x is unreachable in some S_N, then x is unreachable in S (and thus in T);
2) if there is a deadlock in S then there is a deadlock also in some S_N.

Statement 2 cannot give any guarantee that the results of a verification on a finite-state version S_N may be extended to an unbounded S. For example, a deadlock in S may occur only when a certain number k of instances of a model have been created: checking only S_N 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 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 package, although they cannot be automatically generalized.

5.1 Applications of Spin

During the experiments that various developers and we performed, many errors have been found and corrected by using Spin. Once the error is found, Spin builds a counter-example 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 instead visible.

Some of the errors detected with Spin would probably have escaped even an accurate testing phase. For instance, in the specification of a Notepad-style editor, 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 the error was immediately found by deadlock detection, *before* designing the layout and linking it to the VEG specification.

Also validation activities are well supported by Spin. In the above example of the Notepad editor, there was 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 became disabled after its activation, even though the text remained selected. The problem was that the controller did not send any 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 the following state invariant (with obvious meaning):

```
state(text,selected)->
state(cut,enabled)&&state(copy,enabled)
```

This error would be found also with traditional testing, but with much higher costs of detection and correction.

6 IMPLEMENTATION

The notation and results presented in the previous sections are part of a LTR 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 and provide a set of tools, based on compiler technology, to be used during GUI design. These tools are designed to complement traditional layout tools such as those provided by Java Workshop or J++. 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 Fig. 6.1.

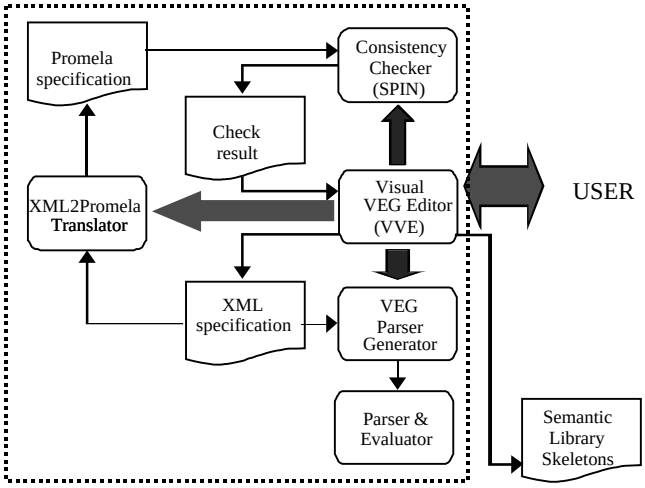

Fig. 6.1: The VEG development process.

The user interacts with the Visual VEG Editor (VVE) to write a VEG specification. Following a user request, 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 user may check the consistency of the specification, by using the Promela file as input to the Spin model checker. Model checking could be omitted by the user, but it is essential for safety critical applications. To produce the real application, the XML file is used to generate 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 input events.

Realistic applications have been specified and implemented, like a Notepad-style editor, a graph construction library and a large real application to medical software. One window of the latter is shown in Fig. 6.2.

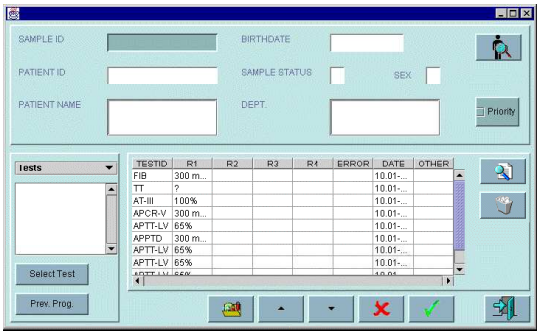

Fig. 6.2: A window of a GUI of a medical software application developed in VEG.

Various experiments have found no difference in performance between the VEG-generated code and Java code hand-written to implement the same application. The developers of the medical software application claimed overall significant time saving over previous development of a similar application. The largest benefits were, predictably, concentrated in the testing phase.

7 RELATED WORKS

The idea of applying context-free or regular grammars to the description of dialogs is not new, since it is at least as old as of 1981 (e.g., [11,14,15]. 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. Some authors introduced a special notation to supplement grammars whenever they seem unsuited to describe some features. For instance, Van den Boss [17] 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. As it was shown, 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 amount of features that are really useful in designing GUI, while avoiding to make the notation rather baroque as other approaches have often done. Grammar approaches were abandoned since they do not deal with parallelism, lack structuring constructs and fail in cleanly integrating data structure and control structure aspects of a GUI. However, we have shown that the VEG constructs support parallelism and structuring, and that the use of attribute grammars neatly blends data and control.

There are of course many formal notations other than grammars, used also for GUI design, such as Statecharts [7] and Petri Nets (e.g., [1]). While these methods have been widely applied for GUI specification and design, at least in academic research, their own modeling power is excessive (and often not well tailored towards GUI applications), 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 [10]), where test cases are generated and then checked with a test oracle. Model checking has also been applied for V&V of GUI, e.g., in [5], where an existing GUI is abstracted into a finite state version that can be checked with SMV. In general, these approaches (either test oracles or model checking) 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. As [5] says, "Selection and application of abstractions is fundamentally a creative process and this is its chief drawback." 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 V&V: in principle, the model checker could also be used to generate test cases (and test oracles) as well.

Another aspect of VEG is that the specification is largely layout-independent: this approach is similar to [6], where interactive services using more than one interface 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.

8 CONCLUSIONS

In this paper, we have shown how to apply grammars for the specification, design, verification and implementation of GUI. Dialogs are specified by means of modular, communicating grammars called VEG (Visual Event Grammars). VEG extend traditional BNF grammars to make the modeling of dialogs more convenient.

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, to test its consistency and correctness, to detect deadlocks and unreachable states, and also to generate test cases.

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.

The complete VEG toolkit is going to be available soon as free software. 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.

Acknowledgements The VEG notation and toolkit is the result of an Esprit Long-Term Research project, called Gedisac, started in 1998 and completed in 2000. Special thanks to Marco Pelucchi, who developed the VEG toolkit and various prototypes, first as a graduate student and after while working at Txt. We gratefully acknowledge the contributions of many people to the development of the VEG ideas and toolkit. Among them, Alberta Bertin, Txt, Fabien Lelaquais, Marie Georges and Christian de Sainte-Marie, Ilog, Alessandro Campi, Politecnico di Milano. We also thank the project reviewers Gorel Hedin, Lund Inst. of Technology, and Ian Sommerville, Lancaster University, and the project officers Pierrick Fillon and Michel Lacroix, for their many useful suggestions.

REFERENCES

1. Bastide, R., Palanque, P., A Petri Net Based Environment for the Design of Event-Driven Interfaces, 16th *Int. Conference on Application and Theory of Petri Nets* (ATPN'95) Torino, Italy, 20-22 June 1995.
2. Berstel, J., *Rational Transductions and Context-Free Languages*. B. G. Teubner, Stuttgart, 1979.
3. Brun, P., XTL: A Temporal Logic for the Formal Development of Interactive Systems, in [12].
4. Clarke, E.M., Emerson, A., Sistla, A. P., Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM TOPLAS* 8(2): 244-263 (1986).
5. Dwyer M.B, Carr, V., Hines, L., Model Checking Graphical User Interfaces Using Abstractions, *Proc. 6th European Softw. Eng. Conf.*, 244-261, Sep. 1997.
6. P. Godefroid, L. Jagadeesan, R. Jagadeesan and K. Laufer, Automated Systematic Testing for Constraint-Based Interactive Service, *FSE'2000*, pp. 40-49, San Diego, Nov. 2000.
7. Harel, D. , Statecharts: a visual formalism for complex systems, *Science of Comp. Progr.* 8, 231-274, 1987.
8. Holzmann, G.J, The Model Checker Spin, *IEEE Trans. on Software Engineering*, 23(5),279-295, May 1997.
9. Nymeyer, A., A grammatical specification of human-computer dialog, *Comp. Lang.*, 21(1):1-16, April 1995.
10. A. Memon, M. Pollack and M.L Soffa, Automated Test Oracles for GUIs, *FSE 2000*, San Diego, CA, Nov. 6-10, 2000.
11. Olsen, D., Pushdown automata for user interface management. *ACM Trans. on Graphics*, 3(3):177-203, July 1984.
12. Palanque, P., Paternò F. (eds), *Formal Methods In Human-Computer Interaction*, Springer Verlag, December 1997.
13. Paterno', F., Faconti, G., On the Use of LOTOS to Describe Graphical Interaction, in *People and Computers VII: Proceedings of the HCI'92 Conference*, Cambridge University Press, pp.155-173, September, 1992.
14. Reisner, P., Formal Grammar and human factor design of an interactive graphics system. *IEEE Trans. on Software Engineering,* 7(2), 229-240, 1981.
15. Shneidermann, B. Multiparty grammars and related features for defining interactive systems, *IEEE Trans. Syst. Man Cyber.,* SMC-12, 2, 1982, pp. 148-154.
16. Shneidermann, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 3rd edition (July 1997), Addison-Wesley.
17. van den Boss, J. Abstract interaction tool: a language for user-interface management systems. *ACM Trans Prog. Lang Syst.*, Vol. 10, 1988, pp. 215-247.