Experiences with a Formal Method for Design and Automatic Checking of User Interfaces

Alessandro Campi, Eliseo Martinez, Pierluigi San Pietro

Dipartimento di Elettronica E Informazione
Politecnico di Milano
P.za Leonardo da Vinci, 32
20133 Milano, Italia
{campi, martinez, sanpietr}@elet.polimi.it

1 INTRODUCTION

This paper addresses our experience in using and developing the VEG (Visual Event Grammars) toolkit for the formal specification, verification, design and implementation of graphical user interfaces.

In fact, VEG follows the traditional Seeheim's model, decomposing a GUI in three parts. The Presentation Part takes into account how the different components of the GUI are depicted and laid out (e.g., colors, borders, sizes and locations); in VEG this is delegated to an external presentation design toolkit. The Dialog Control Part specifies all possible interactions among GUI components (their dynamic behavior, i.e., sequences of actions taken in response to a user-initiated event). The Application Interface Part specifies how the GUI gets linked to the application that effectively does the work requested by the user, e.g., by making the Dialog Control Part to invoke suitable application methods.

The VEG notation of the toolkit is used only to describe the dialog control of GUIs, rather than their presentation: a VEG specification is completely independent of the actual layout of the GUI. Separation of presentation and behavior allows the reuse of the same logic with different presentations (e.g., in multi-interface systems such as banking systems allowing access through automatic tellers or web pages).

Dialogs in VEG are specified by means of modular, communicating grammars, which have an intuitive visual notation, supported by a visual editor called Dialog Control Editor (DCE). 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.

The VEG notation is based on describing a set of rules, each one defining the behavior of a component in a different state. The typical structure of each rule is:

<current state> ::= <guard> <communication> <visual action> <goto next state>

where the <current state> is the logical state of the GUI component, <guard> is the occurrence of one or more events (e.g., a user pushing a button), <communication> is sending events to other components, <visual action> is the specification of a visual change (to be programmed based on the particular presentation toolkit). Since the notation is simple (and visual), users may not be aware they are manipulating grammars, since each grammar rule simply describes sequences of interactions. Hence, even though the notation is completely formal, it may be appreciated also by people without a special mathematical background.

VEG implements also another kind of separation, namely *separation of data and control*. Every VEG specification is composed of a "purely syntactical" part, describing the event-driven behavior of the GUI (the control), and of a "semantic" part, implementing data manipulation. 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 Java. A *semantic function* has some *variables* (or semantic attributes) as arguments and computes a value as result. Values can be passed to visible 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. Hence, a VEG specification is actually a form of *attribute grammar*, a technique invented by D. Knuth in 1968 to make design of compilers more rigorous.

One of the main problem in complex GUI design is their *verification*: 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. However, in VEG, 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. Verification in VEG is completely automated and it is based on a safe (and very natural) abstraction of VEG specifications, which basically eliminates the data manipulation part (but still allows very detailed specifications). This addresses one of the major obstacles in the industrial spreading of formal methods, i.e., the perceived difficulty in their use. The price to be paid for this is that only certain kinds of properties (typically, safety properties) can be verified, while other properties (typically, liveness properties) cannot: simplicity is obtained at the cost of sacrificing generality.

In our opinion, one of the key points of GUI model-based design must be *automatic code generation*: predictably, we found no interest of programmers in models that have to be reprogrammed by hand, or in model that have to be extracted from existing code. Instead, they are mainly interested in simple, reliable visual techniques to quickly design a UI. VEG supports code generation, by using a customized parser generator (based on ANTLR). Since the layout is not described in VEG, the complete generation of a GUI is possible if the dialog model is linked to an actual presentation of the GUI. In the VEG environment, this link is based on a visual tool, called Visual Platform Editor, allowing the association of VEG events with presentation events. In fact, the VEG notation does not specify how user events are linked to the low-level events of the presentation. For instance, a VEG event <*go*> is not explicitly connected to a left-click of the mouse on a PushButton labeled Go, but this connection can be visually described by the VPE using button activation, menu selection or keystrokes shortcuts.

2 THE VEG TOOLKIT

The VEG toolkit is designed to complement traditional presentation tools such as those provided by Java Workshop, Jbuilder, NetBeans 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 of a GUI using the VEG toolkit is depicted in Figure1.

The user interface developer interacts with the Dialog Control Editor (DCE) to write a VEG specification. DCE is a visual tool, which allows the UI developer to design graphically the behavioral aspects of a GUI. The grammar rules of VEG are described by means of a very simple and intuitive graphical representation, directly based on the (attribute) grammar rules. Each GUI component corresponds to a so-called VEG *model* describing its behavior. Within each model, UI developers may define a set of symbols, representing all conceptual events and actions the model deals with, as well as other symbols to represent the model state. Then, for each possible model state, one or more rules describes which events are acceptable, and what the desired response is. Different kinds of views for system components are offered, allowing a UI developer to concentrate on the relevant details. The specification can also be validated with syntax and type checking.

Following a UI developer request, DCE may produce two different specifications: the first one is an XML encoding of the VEG specification; the second one is the translation into the language of the SPIN model checker (Promela).

The UI developer 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 UI developer, but it is essential for safety critical applications. Knowledge of Promela and of Spin is not required.

To produce the real application, the XML description of the interface is used by the Code Template Generator (based on the ANTLR parser generator) to build templates of the controller classes. These classes consist of a set of communicating parsers and semantic evaluators, with some parts yet to be filled with code.

The Visual Platform Editor (VPE), the component devoted to link the controller to the presentation, builds the final version of the code, automatically generating all the sequences of enabling and disabling

of components necessary to make the graphical interface compliant with the VEG specification. VPE takes as input the XML description of the specification, the java templates of the Code Template Generator, and the java files of the presentation. Its output is the complete application, composed of the same files of the presentation enriched by the code that allows the controller to guide the visual interface and the code of the controller.

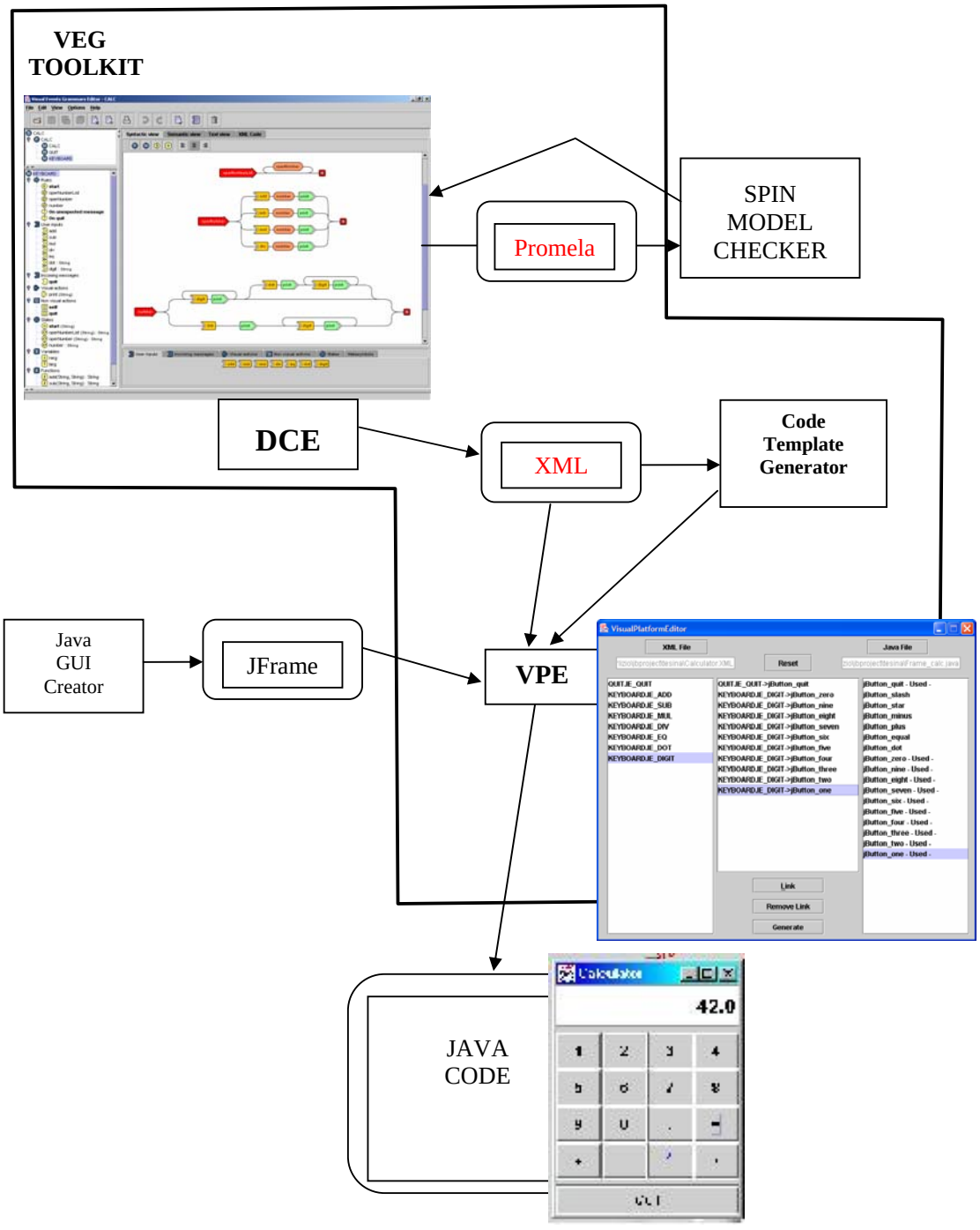

Figure 1 The VEG development process.

3 EXPERIENCES AND LESSONS LEARNED

Realistic applications have been specified, verified and implemented in VEG. Various experiments have found no difference in performance (i.e., response time and memory) between the VEG-generated code and Java code hand-written to implement the same application. The developers of a 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.

One may ask why the toolkit was, in the end, not successful and programmers are not using it, if the ideas were valid. This is a common fate of many similar projects, and we think that some reflections on it may have an interest also for the model-based UI design community. It also illustrates a typical case of failures in the application of formal methods in industrial settings.

Various difficulties were encountered since the conception of the ideas, born in 1999 during an Esprit project 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. Industrial users implemented a first version of the toolkit.

At project completion, many unexpected difficulties were encountered.

First, even though technicians and programmers in the project found the toolkit very interesting and supported its adoption, their management decided that model-based UI building is not a strategic issue and aborted internal adoption. The rationale was that, more or less, programmers can already program GUIs and investments to improve quality and reliability would not pay off (this may be called the *market share* effect: it is often better to be the first on the market with a buggy product than the second with a good one).

Second, publication of the results was much delayed because of a patent application, which at the end actually collapsed for contrasts among the partners. This kind of contrasts on proprietary software and techniques are very common, and a commitment from the start to a *free software* or even to an *open source* style would have been much more fruitful. Open source technology is particularly suited to academic or academic-industrial projects, since it may in principle allow faster spreading and adoption of a technology: proprietary tools can only be pushed, often with difficulties, by specialized tool vendors.

The final difficulty was originated by the combination of the two above: the abandonment of the project by the toolkit implementers and the adoption of a non-open source technology made maintenance and improvement very hard when the project ended. This task was left to one of the academic partners (Politecnico), but universities are not always very well equipped to deal with software development. The toolkit at the end was so difficult to be fixed that had to be reimplemented. The reimplementation is actually still in progress but a new release is imminent (the original code generator is still in use, but its eventual replacement is already foreseen).

REFERENCES

All relevant references to VEG, including papers referencing related works, may be found on the VEG website: http://www.elet.polimi.it/upload/campi/veg/

Acknowledgments. Among the many people who have contributed to the design of VEG and of its toolkit, we specially thank Jean Berstel, Stefano Crespi Reghizzi, Gilles Roussel and Emanuele Mattaboni.