Escalante: An Environment for the Rapid Construction of Visual Language Applications

Jeffrey D. McWhirter and Gary J. Nutt

CU-CS-692-93 December 1993

University of Colorado at Boulder

Technical Report CU-CS-692-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Escalante: An Environment for the Rapid Construction of Visual Language Applications

Jeffrey D. McWhirter and Gary J. Nutt

December 1993

Abstract

Escalante is an environment that supports the iterative design, rapid prototyping and automatic generation of complex visual language applications with a modest amount of effort. It enables the application developer to specify the application and its interface by defining its data model and the corresponding visualization model (using a visual specification environment). The data models are general graph models, while the visualization models are relatively unconstrained graphics; this enables the user interfaces to represent a broad set of presentations and views while adhering to a general framework in which well-defined behavior can be easily specified. Once the data and visualization model have been defined, Escalante will generate a program that implements the data model and the viewing mechanism using a fixed control mechanism; the resulting program can be enhanced to incorporate arbitrary application software. The approach enables a surprising range of interfaces to fit within the meta model; the paper characterizes the spectrum of the domain by describing different example applications (including some quantification of the effort required to construct each example).

1 Introduction

Graphical user interfaces are a key component of many modern computer applications. The graphical representation of the state of an application and the ability to manipulate that state through its representation has the potential to greatly facilitate human computer communication. *Visual language applications* (or environments) are specifically dependent on this type of user/application interaction. Using a visual language application one constructs and manipulates a *visual program* which is based on the constructs and properties defined by a *visual language*. In these systems the state of the application is tightly coupled with its representation, blurring the distinction between application and interface. Visual language applications serve to facilitate human computer communication in many areas including simulation, modeling, visual programming, software engineering, educational systems and network design (e.g., [1]).

Figure 1 shows a screen snapshot of an example system built using Escalante. This application, *BooleanCircuit*, supports the construction and direct manipulation of boolean logic circuits and is based on a visual language composed of *AndGate, OrGate, NotGate, OnOff* and *Connection* elements. Changes to the input of a circuit are made directly through the OnOff nodes. These changes are propagated through the circuit with the gates applying their respective boolean operations. We will use this example throughout the paper to illustrate the process of creating applications with Escalante.

The specific problem Escalante addresses is the development effort to implement a visual language application. Traditionally, software support for the development of user interfaces focuses on *how* the user interacts with the interface rather than on *what* the user acts. There are increasing efforts to address issues concerning the *what* of applications, ranging from the very general (e.g., HUMANOID [11], UIDE [10]) to the more focused (e.g., Unidraw [12]) to the very specific (e.g., Edge [8]). There exists a tradeoff between the overall applicability of an environment and the degree of support provided by the environment for a particular application. The more general approaches are applicable to a broad range of applications but the degree of support offered to a developer for a particular application is limited. The more restricted approaches can greatly facilitate the development of certain types of applications

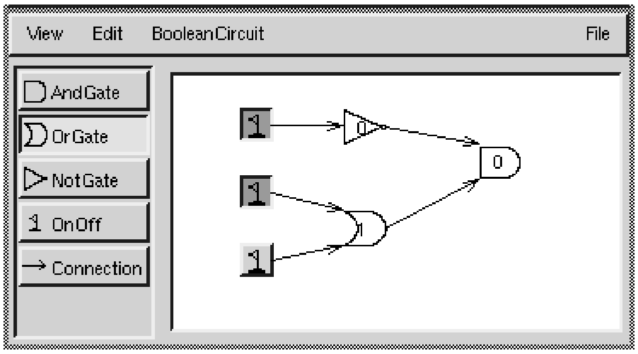
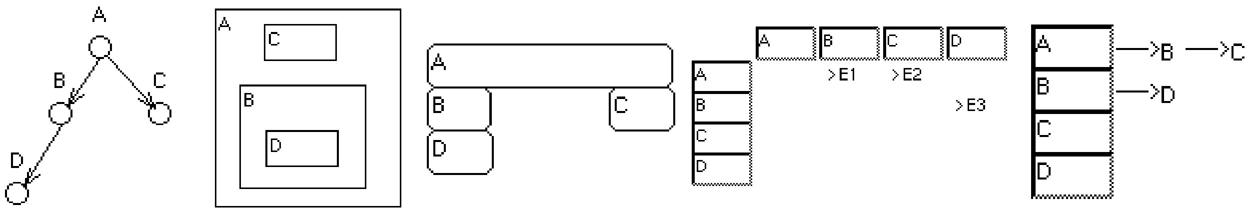


Figure 1: Boolean Circuit Application

```
N = (A,B,C,D)
E = (E1(A,B),E2(A,C),E3(B,D))
```

(a)



(b) (c) (d) (e) (f)

Figure 2: Graph Model Representations

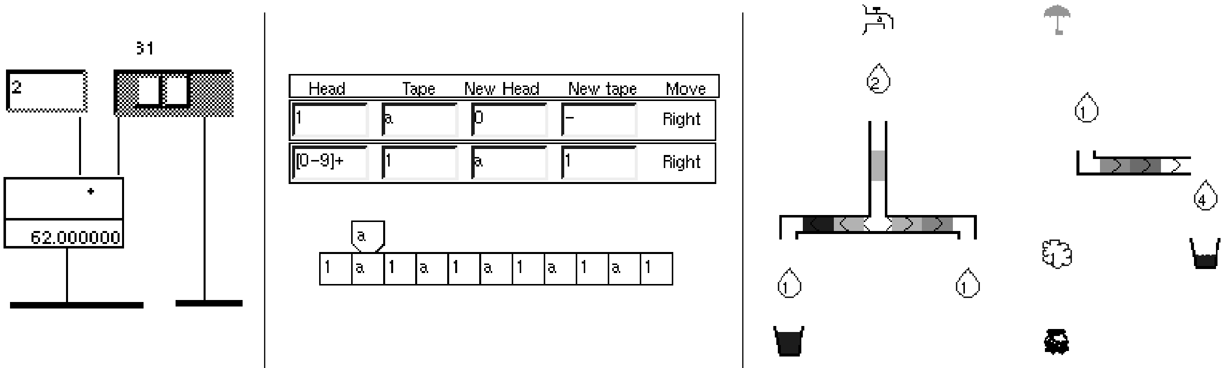
but their domain is limited. Current software technologies do not provide the level of domain specific support required to *rapidly* construct highly functional applications for a broad range of visual languages. Our approach is to focus on applications for graph model based visual languages, providing an environment that allows a system developer to rapidly construct applications with a minimal amount of programming.

In the following sections we discuss the class of systems we address and the problems these systems pose to the application developer. We then give an overview of Escalante, describing its component architecture and the language specification environment *GrandView*. Related efforts in this area are then discussed. The results of this work are presented through the description of a set of applications that have been built using Escalante.

2 Visual Language Applications

Escalante supports building applications for visual languages that are based on object-relationship abstractions (e.g., nodes and edges). We define languages such as these to be *graph model based*. The characterization of the domain as graph models reflects the form of the underlying language constructs, not any particular representation of those constructs such as circles and arrows. For example, Figure 2 shows six different representations of the same underlying graph model constructs. As seen in this figure, graph models can be represented in many different ways (e.g., textual list, directed graph, containment, spanning hierarchy, adjacency matrix and adjacency list).

The constructs that make up visual languages and the uses those visual languages are put to ranges from the simple to the complex. For example, Figure 3 shows a set of visual languages, including a dataflow language (*DFlow*), a *Turing Machine* simulation language and a water flow simulation language (*Waterworks*). As seen in this figure, visual languages can be made up of complex language constructs that exhibit complicated and dynamic graphical representations, spatial relationships, complex behavior, etc.



(a) DFlow (b) Turing Machine (c) Waterworks

Figure 3: Example Visual Languages

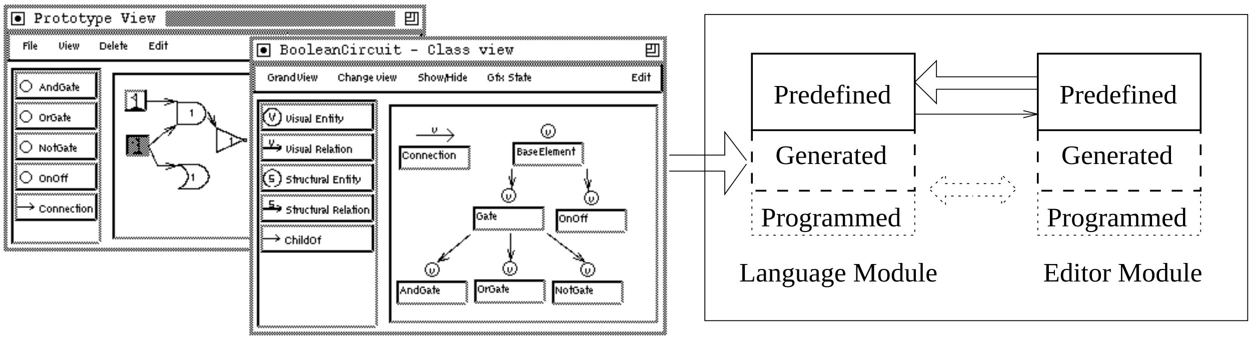
The complexity of a visual language application varies with the syntax and semantics of its associated language. The functionality embodied by these systems may range from simple editing tasks to generation of external artifacts (e.g., code generation) to complex and dynamic simulations. The *use* of a visual language within an application introduces a wide range of issues that have to be addressed. A visual program is not a static entity. The user interacts with the application, creating, deleting, moving, and copying elements of the visual program. A visual language application may also make use of multiple language representations, multiple windows, visual abstractions, etc. These factors of language and application complexity are addressed in the development effort using Escalante.

2.1 Developing Visual Language Applications

The development of a visual language application is a difficult task. A recent study points out that, on average, approximately fifty percent of the implementation effort for a wide variety of applications was devoted towards the user interface [7]. This observation may be conservative for visual language applications. The highly graphical and interactive nature of these applications tends to place emphasis on the interface, blurring the distinction between the application and the interface.

The effort required to implement a visual language application impedes the entire process of language and application development. Iterative design and prototyping of the evolving language and application is a difficult task. The development effort interferes with the ability to refine the language and application in order to provide a good fit between the problem domain, language, application and user. The investment of time required to implement an application can affect the overall usability and usefulness of the application and may even preclude application development.

The goal of Escalante is to provide a high level of support to the developer of a visual language application and to enable rapid application development with minimal programming. Our approach is to provide a rich substrate that is applicable to a broad range of visual language applications and provides deep system support for those applications, including



(a) GrandView Specification Environment (b) Target Application Architecture

Figure 4: Escalante Architecture

the underlying application data model, representation of the data model and an extensive editing component. A visual specification environment facilitates the development process by allowing the application developer to focus on the definition of the target visual language and to ignore many of the implementation details and complexities of the system substrate. Using Escalante, the developer can explore new language constructs, representations and behaviors. Escalante allows the developer to expand the range of potential language and application features in order to create usable and useful applications.

3 Escalante

The development of Escalante has been guided by a conceptual *language characterization framework* [6] which provides a cohesive and general way in which to describe the constructs and characteristics of a diverse set of visual languages. The ability of this framework to describe the constructs and properties of visual languages is crucial for providing coverage for both the breadth and the depth of the domain. Space constraints limit our ability to explain all of the aspects of Escalante and its potential uses. In [5] we provide a detailed description of Escalante and the process of creating applications with it.

3.1 System Architecture

Escalante consists of three components: base language module, base editor module and the GrandView specification environment. Figure 4 shows the development process and a conceptual view of the target application architecture. Applications built using Escalante are composed of a language module and an editor module. The language module contains most of the language specific functionality required within an application, encapsulating the application data model and the representation of the data model. The editor module is the interface between the user and the underlying language module, providing to the user the ability to create, inspect and manipulate a visual program.

The language and editor modules are made up of a predefined base component coupled with generated and programmed language specific components. The predefined components encapsulate general functionality and behavior of visual language applications. The generated components are created through the GrandView environment and encapsulate language specific functionality. There are fixed interactions defined between the predefined language and editor modules. For future reuse of the language module we have limited its functional dependencies on the editor module. The programmed components of the language and editor modules are created manually by the developer and are used to modify or extend system capabilities and functionality that are not provided by the predefined and generated components. It has been our observation that non-trivial visual language applications can be created with minimal manual programming. The manual programming required typically involves the implementation of language specific functionality (i.e., language semantics). For example, the programmed component of the BooleanCircuit application in Figure 1 involved implementing the semantics of the boolean operations of the Gate elements.

3.2 Language Module

In Escalante a visual language is not simply a set of graphical marks or images, rather, it is made up of a set of constructs that exhibit certain properties (including graphical images). To provide explicit support for a diverse set of language constructs and properties we make the distinction between the constructs and the characteristics of those constructs. Language constructs are generalized into a simpler form - *entities* and *relations*. This generalization reflects the characterization we have made of graph models as object-relationship abstractions.

An entity represents a thing within a language (e.g., node, graph, subgraph, aggregation). A relation concretely defines some relationship (e.g., edge, member of a graph, containment) between two entities, termed the *tail* and *head*. We use the entity/relation characterization as a means to make explicit the possibly implicit or abstract, constructs and relationships that occur within visual languages.

To provide support for the behavioral mechanisms of visual languages we have generalized the functionality and behavior one encounters in specific visual languages and applications and encapsulated this functionality as general mechanisms in the entity/relation constructs. These mechanisms include propagation of fixed events (e.g., deletion, movement), propagation of functions, propagation of attribute values, visual dependencies and spatial constraints.

The actual implementation of the language module is based on a collection of static classes organized as an inheritance hierarchy. A *visual program* is a collection of objects instantiated from these classes. Figure 5 shows the classes which make up the predefined component and a set of generated classes that make up the constructs used in the BooleanCircuit application. There are three groups of predefined classes: visual elements, structural elements and graphic primitives.

The *GraphObject* class implements an *attribute value mapping mechanism* which is used to propagate attribute values from one object (source) to the attributes of other objects

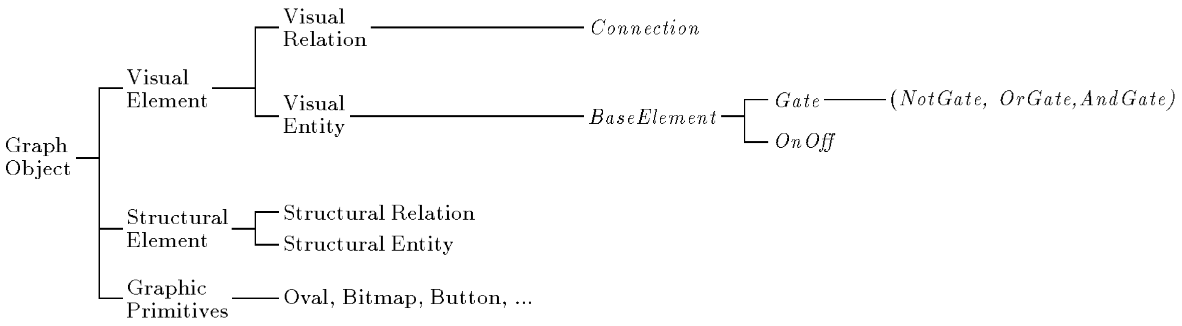


Figure 5: Language Hierarchy

(targets). One can place any number of *attribute filters* between the source and the target that allow for the modification of the values and the control of the attribute mapping process.

The visual element classes, in conjunction with the graphic primitives, encapsulate the state and functionality related to representing, selecting and manipulating language constructs on the screen. The actual image of a visual element is derived from instances of the graphic primitive classes. The *VisualRelation* class provides the ability to define *location constraints* which allow one to define spatial constraints such as containment and adjacency.

The structural element classes have no visual representation, rather they provide a means to create applications that consist of more than one visual representation of some common set of language constructs. A structural element serves to connect or group a set of visual elements. For example, each of the representations shown in Figure 2 are a part of the same application. Each visual element in one particular representation is related to other visual elements in the other representations through a common structural element.

The graphic primitives are used to define the actual representation of a visual element. This set of classes includes simple graphical images such as bitmap, line and rectangle; classes that allow one to group and layout collections of graphic images; and *widget* graphics such as fields, buttons, text views, and menus that allow for direct user input to a visual element. The attribute mapping mechanism is used to define mappings between attribute values within a visual element and attributes of the graphics objects which form the image of the element. These graphical attributes include color, text value, pen width, menu entry, button state, etc. Mapping attributes between a visual element and the objects that make up the representation of the element allows for directly manipulating the internal state of the element through its representation as well as providing a means to automatically represent the internal state of the element.

3.3 Editor Module

The main construct of the predefined editor component is the *EscalanteView* class. This construct encapsulates a wide range of visual program editing capabilities including: the creation, deletion, and copying of language elements; graphical editing capabilities such

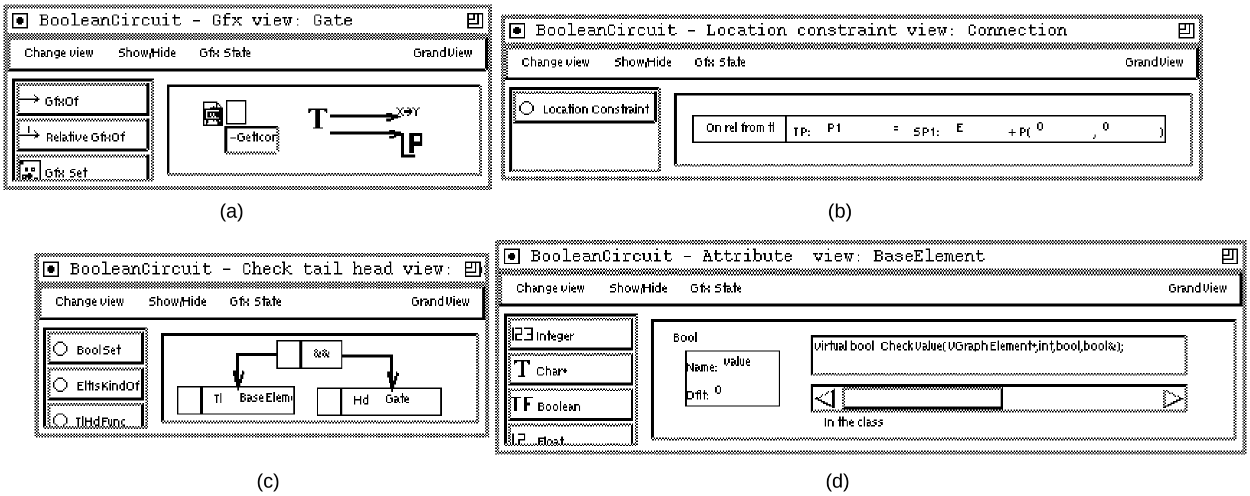


Figure 6: GrandView Language Specification Environment

as moving, resizing, scaling, alignment and simple layout; and grouping and manipulating groups of elements. There is a framework provided for creating online help. N-level undo/redo of element creation, deletion and movement is supported. One can copy/paste and export/import components of a graph. Very flexible mechanisms also exist for multiple views, viewing subgraphs and filtering out the display and selection of elements. The generated editor module is a template class, derived from the EscalanteView, that can be tailored to fit the particular needs of an application. Methods defined in the EscalanteView can be overwritten in order to modify or extend the predefined functionality.

3.4 GrandView

The principal tool used to create a visual language application is GrandView and its associated visual language *Grand*. Referring to Figure 4a, the developer uses GrandView to define the constructs and characteristics of a target visual language. Since Grand is a visual language, this specification process is itself an instance of visual (meta) programming. The language specific modules of an application are generated from the specification. Language or application semantics not supported by GrandView can be added to the generated language and editor modules. The generated and programmed components are compiled and linked with the predefined modules to realize the final system. For applications that do not require manual programming the generation process and final system construction is handled automatically. Once the specification is complete a working application can be realized in a matter of minutes.

Users interact with GrandView through different views of a language specification. The fundamental views within GrandView are the *Class View* and the *Prototype View*. GrandView supports other views of the Grand specification, depending on the aspect of the specification on which the developer focuses at any given time. Figure 4a shows the Class and

Prototype Views of the BooleanCircuit specification. Figure 6 shows a set of other views of this specification.

Class View In the Class View a set of class specification elements are used to define the target language classes. The *Child Of* relation is used to construct an inheritance hierarchy of these elements. In Figure 4a we see the specification of the language classes that make up the BooleanCircuit example. The *BaseElement* class is derived, by default, from the *Visual Entity* class. The *Gate* and *OnOff* classes are derived from the BaseElement class. The *AndGate*, *OrGate*, and *NotGate* are subclasses of *Gate*. The *Connection* class is derived from the *Visual Relation* class.

Prototype View A *prototype* approximates the behavior of a specified language construct. Prototypes differ from the actual implementation of the generated language construct in that certain aspects of the specification are not implemented in the prototype even though they would be provided in the generated construct. The Prototype View, shown in Figure 4a, allows the user to instantaneously see the results of a language class specification, including the inheritance of properties defined in base class specifications, prior to code generation. For the case of the BooleanCircuit example one can prototype all but the application of the boolean operations on the input to a Gate and the specification of the legal tail and head pairs of the Connection relation. The ability to prototype an evolving specification considerably shortens the cycle of specification, realization and refinement of a visual language.

Alternate Views Figure 6 shows the set of alternate views that complete the majority of the BooleanCircuit specification. Figure 6a shows the *Gfx View* for the Gate class. This view allows for the specification of arbitrarily complex graphical representations for an element. The Gfx specification for the Gate class consists of a *BitmapGfx* to define the basic image and a *TextGfx* to represent the state of a gate. Figure 6b shows the *Location Constraint View* for the Connection specification. This specifies that the tail point of a Connection relation is equal to the east point (i.e., right middle) of the tail element of the relation. Using the *Check Tl/Hd View*, shown in Figure 6c, one can visually define the legal tail/head elements of a relation. The specification shown defines that the tail of a Connection relation can only be a BaseElement and the head can only be a Gate. Figure 6c shows the *Attribute View* for the BaseElement class. The Attribute View allows for the specification of the attributes for the language class. GrandView supports other views (not shown) that allow for the specification of menu entries, attribute mappings within relations, attribute mappings between structural and visual elements, event propagations within relations and defining groups of connected elements. One can also define the default creation of relations between elements based on element type and the spatial relationships (e.g., contains, under) between the elements.

4 Related Work

In this section we limit our discussion to those systems that provide specific support for the development of visual language applications.

Unidraw [12] is a framework that provides very general support for a wide variety of visually oriented applications (i.e., graphical editors). The drawback is the amount of coding required to realize an application, Unidraw does not provide the specific support needed for rapid application construction.

Extensible graph editors such as Edge [8] and T/GE [3] provide support for a narrow domain of languages and interfaces. Systems such as these typically have a set of predefined language constructs (e.g., node, edge and graph) that provide some degree of basic functionality. Our work falls into this general category of systems. The distinguishing feature of Escalante is the size and complexity of the domain which it is applicable to and the level of functionality offered to a system developer. Escalante provides a high level of support for a much wider range of visual languages and applications than the systems under discussion.

The picture parsing approach, as exemplified by the the Palette [2] system, involves the use of modified graphical editors as the means to specify a visual program. The image that is created within the editor is parsed, using a *picture layout grammar*, to derive the actual language constructs. This approach enables the rapid development of applications without the overhead of interface creation. The drawback to the approach is that there is limited domain knowledge within the interface. The interface is based on a set of graphical constructs, not language constructs. The range of functionality within applications is limited.

The AgentSheets system [9] is used to create applications for visual languages that are composed of grid based, communicating agents. AgentSheets is not a general purpose toolkit for visual language application development. Rather, its focus is on the creation and exploration of this particular language paradigm. Escalante has been used to create applications based on this language paradigm (e.g., Figure 3c).

VAMPIRE [4] is an environment that supports the construction of iconic programming systems with the particular focus on the semantics (i.e., execution) of iconic programs. The definition of the semantics is accomplished by specifying attributed graphical rules that define transformations on the visual program. The emphasis of VAMPIRE is on language semantics not on general user interface construction. Escalante does not provide explicit support for defining the execution of visual languages.

5 Experiences

Escalante has been under development for approximately two years and consists of approximately 36,000 lines of C++ (including GrandView). Escalante is built using ET++ [13], an application framework for user interface development.

Escalante has been used to create a number of visual language applications including the screen snapshots shown in this paper. Other systems include a Petri net editor, a dataflow graph editor that supports compiler optimization research, a mockup for a commercial visual parallel programming environment and various workflow systems (i.e., office automation systems). Escalante has also been used to construct an application that simulates constant acceleration forces acting on moving bodies. These systems have been built by both the developers of Escalante as well as other, less experienced, users. However, we cannot draw accurate conclusions concerning Escalante based on the experiences of other users. These users worked with an early version of Escalante (with minimal documentation) and had little or no previous experience with visual languages or user interface development.

We now describe a set of applications built using Escalante. Each of these applications was created by the first author and are used to illustrate the variety of applications one can construct using Escalante. In this discussion we give estimates of the application development effort in terms of the overall time, the number of generated classes and the amount of manual programming required to implement the application. While these estimates cannot give a full and complete measure of the ease (or difficulty) of creating applications with Escalante they indicate that Escalante can facilitate the rapid construction of complex applications.

5.1 BooleanCircuit

The majority of the BooleanCircuit application of Figure 1 was created from the specifications shown in Figures 4 and 6. Approximately 30 lines of code had to be written to implement the application of the boolean functions (i.e., and, or and not) to the input values of a Gate. All other functionality (e.g., representation, value propagation) was achieved through the code generated from the Grand specification.

5.2 Multiple Representation

Figure 2 is a simple example of a multiple representation application. Each of the graphs shown in the figure are part of the same application. Adding elements to a graph in one window causes corresponding elements to be added to the graphs in the other windows. This application was constructed in less than a day with less than 30 lines of manual coding. The generated language module consists of 22 classes.

5.3 DFlow

The DFlow language in Figure 3a allows one to define computations by constructing dataflow graphs. Values can be directly manipulated using input fields, sliders, buttons, etc. These values are propagated along relations and can be mathematically combined (e.g., added, divided) by the *ComplexValue* element. This application was built in less than one half a day and consists of nine generated language classes combined with 60 lines of manually written code that implements the mathematical combination of values by the ComplexValue element.

5.4 Turing Machine

Figure 3b shows a Turing Machine visual language. This language allows the user to construct and simulate a Turing Machine. The rules of the Turing Machine are defined by adding Rule elements to the table shown at the top of the figure. This is accomplished through a single menu command. Rules govern the state, input and output of the Turing Machine. During simulation the *Head* element searches for a Rule that matches its state and writes new values to itself and the *Tape* element it is above. It then moves itself according to the Rule. This application was built in less than one day with approximately 100 lines of code involved to implement the simulation of the Turing Machine.

5.5 Waterworks

The Waterworks example, shown in Figure 3c, is an application for constructing dynamic water systems. A *Water* element has two forms - *Drop* and *Vapor*. A Drop falls until encountering some object. If it is possible the Drop enters the object, else the Drop undergoes a *phase change* and turns to Vapor. Vapor behaves likes a Drop except it rises. Drops are produced by *Faucets* and Vapor is produced by *Kettles*. The rate at which Water is produced by Faucets and Kettles can be changed. *Pipes* accept Drops and carry up to a certain capacity. On reaching capacity Water backs up in a Pipe. The amount of Water in a Pipe is reflected by the color of the Pipe. This application was built in two days and required approximately 300 lines of code to implement the water flow and movement behavior.

5.6 GrandView

The GrandView environment has been built using Escalante (and itself) in a bootstrapped fashion with one development iteration being used to construct the next iteration. This development effort occurred over a span of nine months. The generated language module of GrandView consists of 75 classes comprising 8000 lines of code. Another 5000 lines of code were written to implement the prototyping and code generation functionality of GrandView.

6 Conclusion

Visual language applications are complex and dynamic systems. Creating an environment to support building these systems requires addressing a wide variety of issues concerning visual languages and applications. Escalante is applicable to a broad range of applications and provides a high degree of support for constructing those applications. The language specification environment GrandView overcomes many of the difficulties that arise in using a large software environment such as Escalante. GrandView allows the developer to focus on the important aspects of the target language and abstracts away many of the details that can interfere with the development process. Providing a functionally rich development environment and an amenable way in which to use the environment allows for the rapid construction of complex, highly functional applications.

7 Acknowledgments

Escalante is a realization of the Ph.D. research conducted by author McWhirter; he has been supported in this work by a grant from US West Advanced Technologies. Nutt has been supported by Bull Worldwide Information Systems and US West Advanced Technologies on this work.

References

[1] E. P. Glinert, editor. *Visual Programming Environments: Paradigms and Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1990.

[2] E. J. Golin, S. Danz, S. Larison, and D. Miller-Karlow. Palette: An extensible visual editor. In *Proceedings of the 1992 ACM/SIGAPP Symposium an Applied Computing*, pages 1208–1216, March 1992.

[3] A. Karrer and W. Scacchi. Requirements for an extensible object-oriented tree/graph editor. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Systems and Technology*, pages 84–91, October 1990.

[4] D. W. McIntyre and E. P. Glinert. Visual tools for generating iconic programming enviroments. In *Proceedings of the IEEE 1992 Workshop On Visual Languages, VL'92*, pages 162–168, 1992.

[5] J. D. McWhirter, Z. K. F. Eckert, and G. J. Nutt. Building visual language applications with Escalante. Technical Report CU-CS-655-93, Dept. of Computer Science, University of Colorado, Boulder, Boulder, Colorado, 80309-0430, September 1993.

[6] Jeffrey D. McWhirter and Gary J. Nutt. A characterization framework for visual languages. In *Proceedings of the IEEE 1992 Workshop On Visual Languages, VL'92*, pages 246–248, 1992.

[7] Brad A. Meyers and Mary Beth Rosson. Survey on user interface programming. In *CHI '92: Human Factors in Computing Systems*, pages 195–202, May 1992.

[8] Francis J. Newberry and Walter F. Tichy. EDGE: An extendible graph editor. *Software—Practice and Experience*, 20:63–88, June 1990.

[9] Alex Repenning. *Agentsheets: A Tool for Building Domain-Oriented Dynamic Visual Environments.* PhD thesis, University of Colorado, Department of Computer Science, Boulder, Colorado, 1993.

[10] Piyawadee Noi Sukaviriya, James D. Foley, and Todd Griffith. A second generation user interface design environment: The model and the runtime architecture. In *INTERCHI '93: Human Factors in Computing Systems*, pages 375–382, April 1993.

[11] Pedro Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *INTERCHI '93: Human Factors in Computing Systems*, pages 383–390, April 1993.

[12] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8:237–268, July 1990.

[13] A. Weinand, E. Gamma, and R. Marty. ET++ - an object oriented application framework in C++. In *OOPSLA'88 Conference Proceedings*, September 1988.