Design and Implementation of Programming Environments in the Visual Programmers Workbench

EXTENDED ABSTRACT

Robert V. Rubin, James Walker II
GTE Laboratories, 40 Sylvan Road, Waltham, MA, USA 02254

Eric Golin
Department of Computer Science, University of Illinois,
1304 East Springfield Avenue, Urbana, IL, USA 61801

Abstract

Diagrams play a central role in software engineering. They are used for specifying design elements such as requirements, database models and concurrent systems. Families of diagrams form visual languages, and creating such diagrams constitutes visual programming. The Visual Programmers Workbench (VPW) addresses the rapid synthesis of programming environments for the specification, analysis, and execution of visual programs. A language-based environment for a specific visual language is generated in VPW from a specification of the syntactic structure, the abstract structure, the static semantics and the dynamic semantics of the language. VPW is built around a model of distributed processing based on shared distributed memory. This framework is used both in defining the architecture of the environment and for the execution model of programming languages.

The Visual Programmers Workbench has been used to experiment with visual programming environments for several visual languages. These include entity-relationship-attribute diagrams, dataflow diagrams, finite state diagrams and Statecharts. Much early experience with VPW was gained by generating an environment for a telephony application based on a PetriFSA language. This language is a combination of Petri Nets and finite state automata, and is applicable to the design and specification of concurrent distributed systems.

This paper describes the design of the Visual Programmers Workbench and our experience using it. The remainder of this section discusses our model of visual language specification and gives the architecture of the Workbench. The next section contains an example application created by the Workbench, a visual programming environment for the PetriFSA language that is used to design a telephone switch. The final section compares our work with related systems and discusses the merits of our approach.

The Language Definition Model

The Visual Programmers Workbench is designed around a model of visual languages. This model serves as a framework for visual language specification. The language specification customizes VPW to the application language. Our model of language specification is divided into four components: the syntactic and abstract structure of the language and the static and dynamic semantics.

The syntactic structure specifies the visual appearance and structure of the language. A concrete syntax describes programs as they appear to the user. The lexical elements of a visual language are the basic display items,
such as lines, polygons and text. These elements are combined into a two dimensional arrangement to form a visual program. The concrete syntax determines how the lexical elements may be grouped to form programs in the language.

The Visual Programmers Workbench uses a picture layout grammar to specify the syntactic structure of a visual language [4]. Picture layout grammars are based on attributed multiset grammar. A multiset grammar is similar to a context-free grammar, except that the right hand side of a production is considered to be an unordered collection of symbols, rather than a string. The language generated by a multiset grammar is a set of multisets. An attributed multiset grammar is a multiset grammar which has been augmented with parsing attributes. A production in an attributed multiset grammar is a triple \((R, s, c)\), where

- \(R\) is a rewrite rule \(N \rightarrow M\), where \(N\) is a non-terminal symbol and \(M\) is a multiset of symbols.
- \(s\) is a set of semantic functions, which map from the attributes of the right-hand side to the attributes of the left-hand side.
- \(c\) is a set of constraints defined over the attributes of the right-hand side which indicate when the production is valid.

The abstract structure specifies a model which reflects the underlying structure of the language. This model may abstract out the details of the concrete syntax. The model is embodied in an abstract representation for programs. The abstract structure is the basis for both static and dynamic semantic processing of visual programs. The abstract structure is defined as an object graph. An object graph can be represented by the pair \((O, R)\) where \(O\) represents a set of program objects, and \(R\) is a set of relations on \(O\) specifying interactions between the objects.

Object graphs are an object-oriented representation based on graph structures which provide a natural mechanism for representing the abstract structure of a visual language [5]. An object graph type defines a data structure. An object graph is an instance of an object graph type and may be an aggregate of other object graphs. VPW represents the abstract structure of a visual program by constructing an object graph. Because VPW is designed to support a broad range of language semantics, multiple abstract structural representations are utilized.

The static semantics specifies the static properties of the language, as well as the static processing performed by the environment. The classic notion of static semantics is to express context-sensitive properties of the language, such as type-checking. The context-sensitive properties of a visual language can be specified by associating multi-sorted equations with productions in the picture layout grammar or relations in the object graph.

We follow the broadened concept of static semantics to include the processing of visual programs that may extract, analyze, or synthesize static properties. For example, drawing a new layout for a diagram, verifying the correctness of a program, and computing a query of a visual object are all considered to be forms of static semantics. Such static semantics are specified using an extended form of action routines [6] that define the relationships that must hold among objects and external methods. Static action routines operate in a shared distributed memory called a tuple space as derived from the Linda programming language [7].

The dynamic semantics specifies the execution properties of the language. The dynamic semantics may be used to express program interpretation, simulation, monitoring, and dynamic verification. Dynamic semantics are specified using action routines [6] where an external method may, for example, include a definitional interpreter, or a debugger. Dynamic action routines also operate in tuple space. The evaluation of action routines are triggered by the matching of a template equation to new tuples placed by action routines in tuple space.

Logical Architecture of VPW

The Visual Programmers Workbench is organized around a specification of the language and its functions, and a set of tools that process the language specifications. Figure 1 shows the logical architecture. Here we describe the process by which a diagram is turned into a computationally meaningful abstraction.

The dominant mode of interaction with the environment is through the editor. A programmer defines a program by using the editor to produce a picture.

The first step in producing a program is recovering the structure of the picture. The spatial parser takes a specification of the languages syntax and the picture as its input and produces a concrete structure, the parse graph, as its output. The parse graph describes the syntactic structure of the program.

A set of mapping functions transforms the parse graph into multiple abstract representations. Each mapping function transforms a specification of the parse graph into an abstract representation. The abstract representations are all derived from the syntactic structure of the source program. VPW relies on three separate mechanisms to define a mapping to an abstract syntax: AWK [8], DCG's in Prolog [9], and synthesized attribute evaluation [2] over the parse graph.
A Visual Programming Environment

We have used the Visual Programmers Workbench to generate a customized visual programming environment for a Petri Net/FSA language, which we call the PetriFSA language. This language has long been used to specify communications services on paper [10], but, to this point, has not had a supporting environment. Using the VPW, we were able to rapidly generate a visual environment that supports editing, viewing, querying, executing, and animating PetriFSA diagrams. Then, using the generated visual programming environment, we were able to design and construct a prototype of a telephone switch.

The PetriFSA Language

The PetriFSA language is a state-transition language augmented by concurrency, similar to ESTELLE [11] and other distributed communication languages described in the literature. A PetriFSA diagram is similar to an FSA diagram in that it contains a set of states, one initial state, one or more final states, and a set of labeled transitions connecting the states. A PetriFSA diagram differs, however, in that a state can be executable, and a transition contains both a trigger condition and an action.

The PetriFSA VPE

The dominant use of the PetriFSA environment is in editing a diagram, and executing the application represented by that diagram. In this environment, editing a diagram involves creating the diagram, followed by a cycle of parsing, querying, and modifying the diagram.

An example PetriFSA diagram is shown in Figure 2, derived from [10]. The figure shows a portion of a service script. The major elements in the diagram are states and transitions. A state may have an associated invariant (not shown in the figure), which are conditions that are true while the process is in that state. When a process is in a state, all of the outgoing transitions emanating from that state are enabled. A transition has an associated trigger condition and action. The trigger condition may test for some boolean condition, or may await the arrival of an external message. When an enabled trigger becomes true, the transition is activated, and the associated action is executed. An action is a sequence of methods or tokens. Multiple states and transitions can be active at any time, allowing concurrent behavior to be expressed.

Creating PetriFSA Programs.

Diagrams are drawn using Xpic, an object-oriented graphics editor built on top of the X Window System. The editor was extended to receive and send messages using the Message Backplane described below. Editor actions such as highlighting an object can be invoked through the Message Backplane.
In this figure, the initial state is Idle. When the trigger Off Hook is received, the action Dial Tone is executed, and a transition is made to the state Dialing.

In Figure 2 we see a PetriFSA diagram for a telephone application as the user has drawn it using the editor. The structure of the PetriFSA diagram is recovered by the spatial parser. Figure 3 contains a portion of the grammar defining the concrete syntax of the PetriFSA language.

Once the system has parsed the PetriFSA diagram, the parse graph is transformed into an object graph. Figure 4 contains a schema for a PetriFSA object graph. The relations and their attributes are defined declaratively on a per-production basis. Tuples may contain sort-specific attributes, or other program points.

obj (pfsaState, ['source', 'eval']).
obj (pfsaInitialState, ['source', 'boolean']).
obj (pfsaFinalState, ['source', 'boolean']).
obj (pfsaTransition, ['source', 'target', 'transition', 'trigger', 'action']).

reln (edge, ['source', 'target']).

Analysis of PetriFSA Programs. The PetriFSA environment includes several capabilities for viewing, analyzing and querying a diagram. These capabilities are outlined below.

In Figure 5, the user has applied a graph layout algorithm from the Workbench. This layout is particularly useful in presenting the hierarchical nature of a diagram.

Context checks for consistency and completeness in a diagram are easily captured by placing an algebraic sort specification on the structure of an object. An example is given in Figure 6. The sort specification imposes type conditions on the abstract structure of the object. The specification defines a set of relations that contain a set of attribute-sort tuples, where the sorts are the type specification for the objects. To evaluate these equational theories we rely on the unification algorithm extended to join a many sorted-algebra with subsorts, as reported in [12].
Object Sort Specification

```
pfsaState :=
    state: String accept: Boolean |
    state: String accept: Boolean eval: Boolean |
    state: String accept: Boolean final: Boolean

    check::
        unique (pfsaState.state) ∈ EnvState
```

```
pfsaTransition :=
    trans: String trigger: String action: String |
    trans: String trigger: MSG action: MSG |
    trans: String trigger: MSG action: Method

    check::
        trigger ∈ EnvMSG
        source ∈ EnvMethod
        target ∈ EnvMethod
```

Verify Specification

```
∀ State:
    if not State.accept
        reach (State, '?').

∀ State:
    if State.accept
        reach (pfsaDiagram.initial, State).
```

Several checks cannot be captured exclusively through a sort specification. These checks require context-sensitive environment or state information relative to the entire program. For example in Figure 6 contains several context-sensitive checks that are local to the object-sort specification. One such check fulfills the consistency criteria that every State have a unique id. Another local sort-dependent checks guarantee that routines are properly defined in the environment.

A mechanism is also included in the Workbench for capturing context-sensitive consistency and completeness checks that are global to the diagram, and not specifically local to an object sort. Such checks are captured as verifiers, and are expressed independently of a sort specification. The examples in Figure 6 capture the completeness criteria that all accepting states be reachable from the initial state, and that all states that are not accepting states contain an edge to another state.

The Workbench provides a graph algebra to allow the construction of views in the diagram. The graph algebra includes standard and extended set operations for constructing views of the Information Base. These "set" functions are useful in defining views over the graph. These views can be isolated projections of the graph, or may represent compositions of projections over the graph. In Figure 7 a "fisheye" query (defined by Furnas in (13)) formulated in the graph algebra allows us to better abstract the nature of the hierarchy in a diagram.

```
Figure 6: Portion of PetriFSA Algebraic Specification

A pfsaState is defined to contain an attribute id of sort String, an attribute accept of sort Boolean, and an optional attribute eval of sort Boolean. The object pfsaTransition has attributes source and target that are systematically related to the sort of the Transition.
```

```
Figure 7: PetriFSA Diagram after Fisheye Query

We have issued a fisheye query with a low threshold value, giving us a sparse graph. If we were to raise the threshold, we would see more of the directed connections. Fisheyes provide a view of a structure much like a "fisheye" camera lens. Objects close to a point of interest are shown in great detail, and objects that are further away are shown in successively less detail. A fisheye view is defined as Connect(G, G) with respect to G and API, D. The function API is defined as the distance from a node to a designated root. The function D is defined as the distance from a node to a designated point of focus. The degree of interest (DOI) for a node is then defined as

```
DOI(node) = API(node, root) - D(node, focus)
```

The Workbench also provides a universal-relation query language interface (14) to the objects and relations in the Information Base. Using the universal query language, a user or a semantic action routine can retrieve information from the diagram database, effectively querying the model of the diagram. Queries are specified exclusively over the attributes using the extended relational operators. The universal query language interpreter is ca-
pable of automatically computing joins of the relations over the attributes. Figure 8 illustrates a graph-based query over the attributes of the diagram.

Figure 8: PetriFSA Diagram after Relational Query

We have issued a query to identify all states and transitions that share a specific trigger.

Runtime Environment of PetriFSA Programs. The runtime environment is characterized by dynamic functions operating over multiple static and dynamic representations of language and program structure. At runtime, static objects may be referenced by the dynamic functions, and new dynamic objects are introduced by these functions into the environment. For example, in the runtime environment, the animation function reacts to the execution trace of the interpreter by querying the parse graph to derive the graphical location of the new state. The runtime environment provides the mechanism for coordinating the dynamic uses of objects in these producer-consumer relationships.

To support the dynamic sharing of objects the VPW includes a Message Backplane based on shared distributed memory, also called tuple space [7]. The Message Backplane resembles a memory-resident database—processes update and retrieve information from tuple space. Tuple space resembles a bulletin board, in that processes may post queries that are resolved when the information becomes available. Computation in this tuple space model is three party and anonymous; processes communicate by placing objects in tuple space and retrieving objects from tuple space; they do not communicate by sending messages directly to each other. To obtain an object, a process registers an evaluation tuple called a template equation in the Message Backplane. Only when a process sends an action equation to the Message Backplane that matches the template is an action routine invoked. In fact, the logical architecture is defined as a set of template equations over the static objects in the environment. The distributed memory model requires that functions only have knowledge of the objects over which they operate. As objects are produced, they are distributed and evaluated in the shared memory.

Interpreters for languages in the VPW are specified based on a notion of object definition, (as in Garden [15]) where object types evaluate directly. In Figure 9 we present an object graph system that defines the state evaluation. The interpreter executes by stepping through the state and transition objects. A state is evaluated as a consequence of notification that a transition leading to that state was correctly evaluated. Transfer of control is a function of responding to a template equation in the Message Backplane. Likewise, computing a result is an assertion of data to the Message Backplane.

EvalState(state) {
    MSGsend("STATE %s", state->label);
    if (state->eval)
        MSGsend("EVAL %s", state->eval);
}

MSGquery("PFSA TRIGGER %s"):
{
    EvalTrigger (trigger);
    t = find transition (trigger);
    MSGsend("TRANSITION %s", t->label);
    MSGsend("PFSA ACTION %s", t->action);
    EvalState (t->target);
}

Figure 9: Dynamic Semantics of Object Graphs

Figure 10 describes the run-time architecture of the system. Notice how we have several programs executing concurrently through the Message Backplane. One program is a telephone interpreter, and another represents a switch interpreter. Several telephones rely on the telephone interpreter. In this example, the telephones are connected by a common switch, which is also a PetriFSA diagram.

Figure 11 shows several active telephone applications, and scopes configured to monitor triggers. The phones are X window applications, where graphical interactions result in placing tuples in the Message Backplane. The telephones also initiate queries on the Backplane through which they communicate with a switch. Picking up a telephone results in a trigger being placed on the
Backplane. This trigger is selectively broadcast across the Backplane and handled by the transition evaluation routines that execute the switch and phone diagrams. As the switch and telephone diagrams execute, the diagrams are animated by flashing and highlighting the component that is being evaluated. The editor is integrated into the Message Backplane with an animation facility that uses the relations derived from concrete semantics to define a correlation scheme between the dynamic execution and the diagram. The message scope provides an execution trace of the tuples that were processed by the BackPlane.

Discussion

The Visual Programmers Workbench is a toolkit for the construction of visual programming environments, with an orientation towards distributed applications. VPW takes a comprehensive approach to software engineering, providing support for design, specification and debugging of applications, all within a visual framework. This comprehensive support stems from the provision of multiple views of a program.

The significant features of the Visual Programmers Workbench are:

1. **comprehensive approach:** VPW provides the language designer with tools to specify all aspects of the language. The framework for visual languages used by VPW consists of four types of specifications: the syntactic structure, the abstract structure, the static semantics and the dynamic semantics.

2. **distributed processing:** The Visual Programmers Workbench is built around a model of distributed processing based on Linda [7]. This framework is used both for the architecture of the visual programming environment and for the execution model provided for defining the dynamic semantics of a visual language.

3. **grammar-based language specification:** The syntactic structure and appearance of a visual language is specified with a picture layout grammar. The visual programmer treats his diagrams as source programs. He can create and manipulate them as pictures using a general purpose graphics editor. The syntactic structure of a visual program is recovered using a spatial parser.

4. **abstract representations:** VPW provides mechanisms for mapping from the syntactic structure of a visual program to an abstract structural representation, called an object graph. VPW supports multiple abstract representations of programs. Object graphs provide a natural and convenient representation for structuring, querying, and manipulating multiple views of program structure.

5. **static processing:** VPW provides a framework for defining and analyzing static properties of visual programs. A graph-algebra allows the construction of views, and a universal-relation interface permits the formulation of queries over the attributes of the diagram. In addition to the classical static semantics of programming languages, this framework naturally incorporates user-directed static processing as an aid to program understanding.
6. **dynamic semantics:** VPW allows the dynamic semantics of a visual language to be specified and uses this specification to execute visual programs. Facilities that can be extended by language specifications are also provided for examining the dynamic properties of visual programs, such as monitors and debuggers.

Several other researchers have had similar goals as VPW. The Garden system [15] is a meta-environment for visual programming, where a language specification is used to customize the environment (given as an object graph system); the visual syntax and the dynamic semantics. The visual syntax is described through a mapping from the abstract structure to a picture model. This approach is less flexible than using a grammar and is only suitable for picture generation. Creating visual programs is done with a structure editor, rather than using a parser.

Another similar system is the SIL/ICON compiler [16], which generates icon-oriented systems from language specifications. The concrete syntax of a visual language in the SIL system is specified using a picture grammar, and a parser based approach is taken to visual input. The abstract structure of a program is represented using a conceptual graph, which is essentially a special form of object graph. SIL does not contain broad support for such programming activities as debugging and static analysis.

The Visual Programmers Workbench represents an effort to extend the research on traditional software engineering and design environments to the domain of visual programming. Here we extend this work with a spatial parsing mechanism to recover the structure of programs, an object graph framework, a unification based approach to type-checking, a universal-relation and graph-algebra based approach to static semantics, and a Linda-based relational runtime model. Only by providing tools for the construction of complete environments can the full power of visual programming be realized. Our experience with the Visual Programmers Workbench has shown this approach to be practical.

References