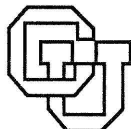
A Characterization Framework for Visual Languages

Jeffrey D. McWhirter
and
Gary J. Nutt

CU-CS-586-92 March 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Characterization Framework for Visual Languages [1]

Jeffrey D. McWhirter
and
Gary J. Nutt
University of Colorado

CU-CS-586-92 March 1992

University of Colorado at Boulder

Technical Report CU-CS-586-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

A Characterization Framework for Visual Languages [†]

Jeffrey D. McWhirter
and
Gary J. Nutt
University of Colorado

March 1992

Abstract

Visually oriented systems implicitly incorporate a *visual presentation*, a logical *data model* of the presentation, and an underlying *system* to support various operations on the data model in the context of the presentation. Visual system designers often do not explicitly separate these components, requiring that the presentation, data model, and underlying system be designed as a monolith. This discourages experimentation with visually oriented systems since it tends to require that a substantial effort be expended to prototype different variants of a visual language system. It also tends to obscure visualization issues with system issues and data model issues. This paper introduces an approach in which graph models are used to define the data model (including structure and appearance of the model), and in which the model specification can be used to automatically generate the system components that support the model and its presentation. The approach relies on a *conceptual framework* to encapsulate general model components and behaviors; any particular visualization/model instance is defined by selecting and enhancing components and behaviors within the framework. In this paper we describe the framework and illustrate its use for defining a spectrum of visually oriented systems.

1 Introduction

Constructing a complete system that uses a visual language to program a computer's behavior is an imposing task. The use of visualization suggests that the user is not expected to be a programming expert, and further that the programming language may reflect fundamental aspects of the user's domain of knowledge. The development of the system environment in which the resulting program is to be "managed" (e.g., entered into the machine, edited, stored, compiled and executed or interpreted) is a separate task in constructing the overall visual language facility.

The goal for this research is to facilitate visual language system development by providing an environment in which the language designer can rapidly build system prototypes, with different visual presentations, for visual languages that are defined in terms of graph models. Specifically we support the specification and generation of the graph model components of these systems within the context of a fixed user interface template (window system) and system environment (visual language editor). Our work enables the visual language developer to easily experiment with different

[†]This work was supported by US West Advanced Technologies.

structural and visual aspects of visual languages and is intended to shorten the path between language conception and system realization.

In recent years there has been considerable interest in the development effort related to visual language support systems. This work has ranged across formal visual language generation and compilation [11, 12], user interface toolkits [32, 13], graph model specific systems [27, 31, 22, 24, 23, 28, 5, 3, 21], visual language parsing [29, 17, 20], and other associated efforts [30, 25, 4, 10]. Our work differs from these efforts in that we focus specifically on the data model and its representation for graph model based visual languages. We address a wide spectrum of structural and visual characteristics of graph models including issues concerning the use of the language within the programming environment, e.g. abstraction mechanisms, element filtering, event propagation, and program navigation. The work is intended to enable systems and models to support the language, rather than to force the language to conform to them.

Human-computer dialog can be cast in a linguistic model or an implementation model [1]. In the linguistic model, there is an input language and an output language, each having meaning and form components. The meaning component can be further subdivided into conceptual (the cognitive model for the user) and functional (semantic behavior) components.

Visually oriented systems may incorporate many different user interface metaphors to define the input language but all use a visual representation to implement the output form component, a logical data model that defines the output meaning, and an underlying environment/system to support various operations on the data model in the context of the presentation. Visual system implementations often do not explicitly separate these components, requiring that the input/presentation, data model, and underlying system environment be developed as a single unit. This discourages experimentation with visually oriented systems since it tends to require that a substantial effort be expended to prototype each presentation and conceptual model. It also tends to confuse visualization issues with system and data model issues.

This paper focuses on a system architecture in which the components are designed individually, led by the data model definition. The *data model characterization framework* is the conceptual basis of the process; it is an environment in which to describe the constructs which make up the data model of a visual language, visual representations of those constructs (within a fixed window system), and certain aspects of the use of the language (within a programming environment).

There are two major contributions that the characterization framework makes: the first is utilitarian – the framework is the basis of a rapid prototyping tool for visual language research. The capabilities of the rapid prototyping tool are a direct reflection of the characterization framework, i.e., the size of the domain of visual languages to which our system is applicable depends on the descriptive power of the framework. The second contribution of the framework is conceptual – it provides a means for describing and studying graph model based visual languages. It allows one to think about the structure of the underlying language model, the visual representation of the underlying language model, and the behavior of the visual language within the programming environment as distinct, but interrelated, entities. Designers often blur the lines between the data model, the visual representation, and the programming environment. Making the distinction between these facets of a visual language can lead to a more precise understanding and definition of the language as well as enabling one to experiment with different visual representations and methods of user interactions with the language.

The next section of the paper describes our view of visual languages, what they are, and how they are used. In Section 3 we present some example graph model based visual languages and

discuss some of the structural and visual complexities that graph models exhibit. In Section 4 we describe the characterization framework. We summarize our work and discuss the ongoing system implementation efforts in Section 5.

2 Visual Languages

During the last decade, visualization as an output language, coupled with direct manipulation as an input language, has enjoyed growing success as a fundamental technology in contemporary human-computer dialog. Our work on the Direct Computer Usage Project [15, 16] has led us to the view that *visual languages* can be thought of as one type of language for dialog between humans and computers – one that is biased toward the human's knowledge domain (and hence is appropriate for direct usage by nonspecialist users). This suggests that the language must have a familiar *appearance* at the interface as well as a familiar and predictable *behavior*.

Thus, we identify three important aspects of visual languages: what the language does, what it looks like, and how it is used. These aspects describe the underlying data model, the visual representation of the model, and the programming environment that supports the use of the language. The underlying constructs and their representation embody the traditional notion of the separation between image and meaning. In recent years this duality has been been discussed more formally by Bertin [7] and Chang [11] among others. In the realm of graphical user interfaces the Model/View/Control paradigm of Smalltalk embodies these three aspects of visual languages [2].

The *data model* of a visual language is the set of constructs from which a visual program is created. The model specifies the constructs, the behavior of the constructs, and how they are put together to form a program. The data model encapsulates the meaning of the language. The complexity of the data model is dependent on the complexity of the intended problem domain.

The graphical (output) *representation* of the data model is a mechanism that facilitates the understanding and use of the language. It is the vehicle by which the system conveys information concerning the data model to the user, and it provides a context for the input language to the system. Both the data model and the task of bridging the gap between the data model and the user have a direct effect on the complexity of the visual representation of the language.

The *programming environment* need not be an integral component of the visual language itself, rather it should enable the use of the language. Due to the nature of the medium in which visual languages are represented, there is a need for a system to support the creation and manipulation of visual programs. The diversity that visual languages exhibit in terms of their data model, the representation of the data model, and the user interactions with the language imply the need for language specific support systems.

Considerable effort may be required to develop a visual interface that satisfies the particular needs and requirements of the visual language and the end user. The need for a language specific support system is a drawback from the perspective of the language developer, due to the development effort required to realize the system for a particular visual language. From the perspective of the end user one can view the need for a language specific programming environment as an advantage. A language specific interface can provide better user support by providing mechanisms that support such activities and functionality as the construction of complex programs, program manipulations, animation, etc. For visually complex languages, mechanisms to support the creation and understanding of the visual program are crucial to the overall usefulness of the system.

Each of these aspects of visual languages are important and contribute to the overall usefulness of the language. The goal of a visual language and its programming environment should be to achieve the best fit between the problem domain, the underlying data model, the visual representation of the model, and the end user. The language will not be useful if the underlying data model does not match the problem domain. If the visual representation does not accurately convey the meaning of the language the usefulness of the language is diminished. The visual representation is a vehicle that conveys information about the underlying constructs. If the visual representation is misleading or does not convey the appropriate information then the user will find it difficult to understand the abilities of the language and its applicability to the problem domain.

It is important to be able to address the underlying constructs of a visual language outside the realm of any one particular representation of those constructs. Independent of the quality of the visual representation, if one does not have the appropriate constructs to solve a problem then the language will not fulfill its intended purpose. While the medium in which these underlying constructs are described and manipulated is visual, one should be able to think about them outside of the context of visual representations. By first focusing on the underlying constructs and their applicability to a particular problem domain one can better ensure that the language will be applicable to the given problem domain.

A visual language is often based on a particular representation of the underlying constructs. Likewise there is a particular set of interaction methods, i.e. a fixed support system, that a user may employ to manipulate the language. We believe that this rigidity in visual representation and interactions can limit the usefulness of a visual language.

The focus on one particular visualization limits the ability to achieve the best fit between the underlying constructs and the representation as well as between the language as a whole and the user of the language. While visualization is important for understanding and manipulating the language, there is no reason to believe that one particular visualization is necessarily the best way to perceive the language. Graphics systems are rich in expressive abilities; one should take advantage of the expressiveness of the medium to explore a wide range of possible visual representations to achieve the best fit between problem domain, visual language, and user.

A fixed method of user interaction can also be a detriment to the ability of the user to understand and make use of a visual language. Graphical user interfaces offer a wide variety of language interaction mechanisms. In the development of a visual language support system these mechanisms should be explored to provide an environment which provides the necessary support required by the user and the visual language. The dependence of a visual language on a support system and the development effort required to realize a support system are major factors in limiting the ability to experiment with different language visualizations and methods of user interactions with the language.

3 Examples

Visual languages often rely on graph models to define the data model (and sometimes to define the representation). One simple example is a visual language for project management based on PERT charts; the data model defines the logical behavior of the PERT chart and the popular representation of a PERT chart uses triangles, boxes, and arcs. The "environment" typically supports the creation and modification of a PERT model, and the analysis of the chart for resource usage, critical path, etc. (based on data model definition).

Graph models embody an object/relationship abstraction with a visual representation of the abstraction. In the simplest case, the object/relationship abstraction can be thought of as a set of nodes representing the objects and a set of edges that represent the relationships. However, graph model based visual languages can also exhibit additional complexity , e.g., within the underlying constructs of the language, the representation of those constructs, and the use of the language within the programming environment. In this section we will briefly describe various aspects of a few visual languages to identify some of the complex structure and representation that they may exhibit.

Raddle and VERDI Raddle [14] is a graph model intended to represent coordination in distributed systems. Raddle models are composed from *teams*, *roles*, *behavioral expressions*, and *interactions*. A model describes how teams communicate and synchronize to accomplish an overall function. A Raddle model can be defined using a formal, textual description; there is no specific graphical representation of a Raddle model.

VERDI is a visual language system built on the Raddle data model [18]. Figure 1 illustrates how VERDI represents a Raddle graph model. The outer box in this figure represents a Raddle team and contains a set of roles. In turn each role contains a subgraph made up of *places* (denoted by circles), *blocks* (not shown), *returns* (ground symbol), *boxes* (double ended rectangles) and *links* (directed edges). *Tokens* are used to represent the flow of control among the components which make up a role and are shown as small, filled circles. There are also implicit groupings of boxes within a team termed *N-party interactions.* In the figure the three boxes labeled "ping" are all members of the same N-party interaction. The two boxes labeled "pong' are members of a different N-party interaction.

Raddle (the graph model on which VERDI is based), is relatively complex in its internal structure and in the visual representation of the internal structure. In addition to the various constructs (team, role, token and N-party interaction), there are also various types of relationships that exist among these constructs. For example, the relationships between a team and its set of roles, and between a role and the components which make up the role, denote the component makeup of the team and the roles. A team contains a set of roles, and a role contains a set of places, blocks, and returns. The relationships among the components of a role denote control flow among those components. The token construct may carry data and denotes those nodes within a role that are currently active. There are rules that govern how these various constructs can be combined to form a correct graph. For example, teams only contain roles, roles only contain places, blocks, boxes, links and returns. A role must contain one, and only one, place. Links may only connect the elements that make up a role. These are but a few of the rules that define the correctness of the VERDI graph model. Describing and instantiating a full complement of structural rules for Raddle is nontrivial; a more complex graph model could be a difficult task.

The visual representation of the VERDI graph model conveys the implicit meaning of the constructs and relationships which comprise the model. The relationships which exist between a team and its roles, and a role and its subgraph, are shown as visual *containment.* This representation conveys the notion of one node belonging to (contained by) another node. The relationships among the components which make up a role are shown as directed edges. These relationships reflect the control flow among the components. The grouping of boxes into interactions is not explicitly shown; rather, boxes with the same labels denote membership in a common interaction.

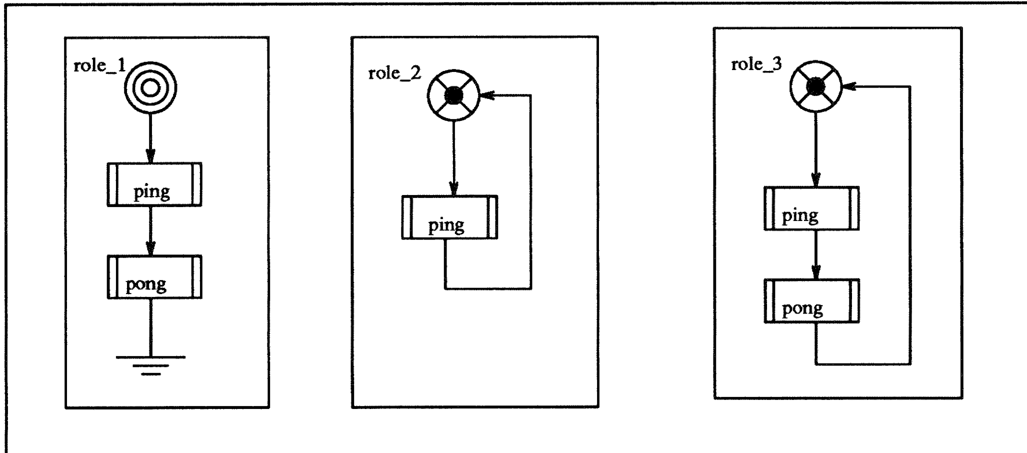


Figure 1: VERDI Example

The CAEDE CAD Modeling System The CAEDE [9] system is a software CAD environment. Figure 2 is an example CAD design developed within CAEDE which exhibits only a subset of the complete CAEDE graph model. Like VERDI, the CAEDE graph model exhibits a wide variety of structural and visual complexity. The node labeled "agent pool" is an *agent* node; the nodes labeled "dispatcher", "agent_1" and "agent_2" are *task* nodes. The node ports or sockets associated with the agent and task nodes are *entries* into their associated nodes. The annotations along the edges of the graph represent data flow along those edges. The annotations labeled "no_user" and "no_agent" serve as guards to their associated entry nodes. The points labeled "free" and "wait" are *plugs* or *labeled connection points.* There are various types of relationships that exist within this graph model. These relationships are represented using both containment and directed edges.

A specification for CAEDE models must capture the node types in a model, and the relationships among the various nodes. Considering that the example represents just a subset of the overall CAEDE graph model, the specification of this visual language and the development of the programming environment to support this language would be a difficult task.

The *plurbius* Visual Programming Environment Figure 3 is an example visual program from the *plurbius* system [33]. The basic component on which the language is based is the *machine.* A machine can be a *doer* or a *tester.* In the figure the node that contains the three nodes, and the nodes labeled "procl" and "/", are doers. The fourth node, labeled with "=", is a tester. Machines may have any number of *hoppers* and in the case of doer machines a *spout.* In the figure the hoppers are the fixtures on the top of the machines. The spouts are the fixtures on the bottom of the machines. Hoppers and spouts reflect the input and output data, respectively, of machines. Machines may be *composite*, i.e. made up of other machines. In the figure the machine visually containing the three nodes and the node labeled "procl" is a composite machine. The solid edges are termed *pipes* and define the data flow among the machines. The dashed edges are called *wires* and define the control flow among the machines. Machines also have sink and source sockets (shown

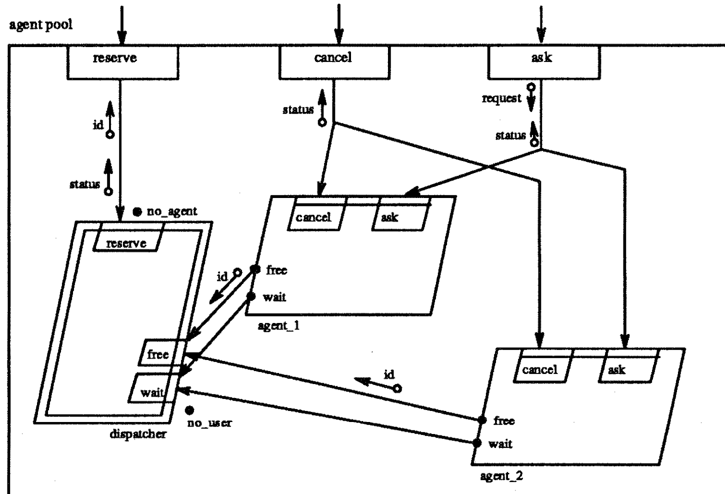


Figure 2: CAEDE Example

as triangular ports on the sides of the machines), used in conjunction with the wires to define the control flow within a program.

Again, a specification for this model must distinguish among doers, testers, hoppers, spouts, pipes, and so on. Further, the composite relation among machines should be reflected in data model so that it can be appropriately represented at the user interface.

The GRACE/CS Visual Specification Language Figure 4 shows an example, taken from [19], of the GRACE/CS visual specification language. This example also represents only a subset of the full GRACE/CS graph model. GRACE/CS is based on the formal notion of a *recursive graph* or *R-graph.* A R-graph allows both edges and nodes to contain subgraphs. GRACE/CS is used to specify the structure of concurrent systems. As shown in the figure, this language exhibits a relatively high degree of visual and structural complexity. The node labeled "iobuffer" visually contains a subgraph. The edge labeled "synchronize" also visually contains a subgraph. There are node ports and as well as nested node ports. The data model specification must reflect each of these relationships.

VERDI, CAEDE, *plurbius*, and GRACE/CS have full programming environments (systems) supporting the data model and representation. We have cited these models to illustrate the complexity of the data model and its implied representation for few visually oriented systems we have seen in the literature. While we have only discussed the static structure and visual representation associated with these models, the rules for constructing each are much more complex than interconnecting a set of nodes with edges. Some relationships are represented by containing one object within another; others by hierarchy, and others by different types of edges.

There are also dynamic aspects associated with the interpretation of these graph models, dealing with how a user creates and manipulates a model within the programming environment. For these languages to be useful, issues concerning scalability, model navigation, abstraction, and creation

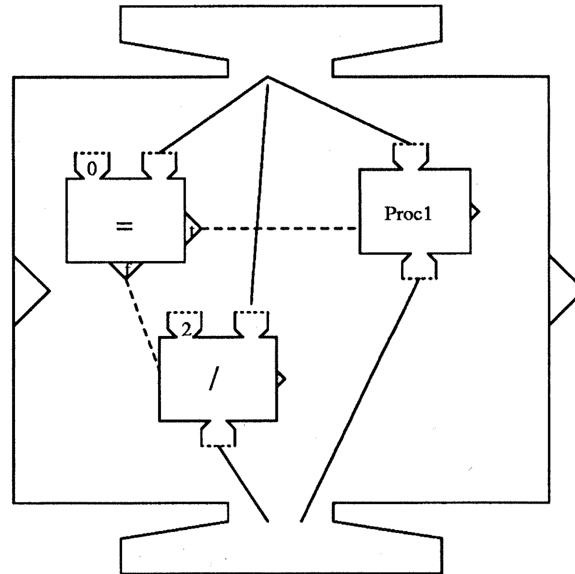


Figure 3: *plurbius* Example

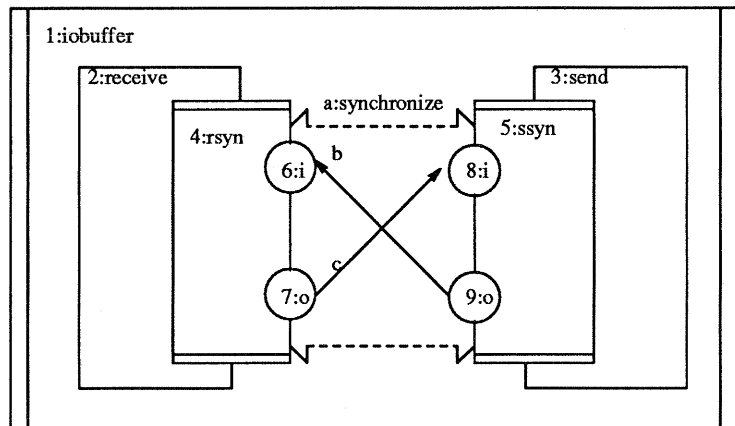


Figure 4: GRACE/CS Example

have to be addressed within the programming environment. Addressing these system issues can lead to further complexity in realizing the visual language and programming environment.

We illustrate how the characterization framework can be used to defined rules of the type needed to build a visual language system of the complexity of those cited in this section. As a pedagogical tool (we have no firsthand experience with the system), we use the VERDI example to explain how the framework is used.

4 Characterization Framework

While we advocate the separation of visual representation, data model, and supporting system, we recognize that the data model defines several aspects of acceptable visual representations. Therefore, we tend to think about data models as addressing the purely *structural* aspects of the behavior (e.g., connectivity, graph syntax, etc.) and also several *visual* aspects of the model (e.g., constraints on its appearance, the effect of deleting a connected node, etc.).

In this paper, the graph model characterization framework is the basis for describing the structural and visual aspects of graph models as they will be operated on within the context of an interactive graph editor. Here, we do not address issues directly related to the dynamic execution or interpretation of the model, i.e., the meaning of the model. Rather, we focus on the specification of the constructs (and their properties) that define a graph model; these specification can be used to generate a graph editor that supports the particular characteristics of the graph (data) model. This section gives a general description of the characterization framework; a more detailed description of the framework appears in [26].

The general goal of the characterization framework is to be applicable to a large set of graph models. Since graph models can exhibit a large degree of structural and visual complexity, the specific goal is to be able to automatically generate the support system for arbitrarily complex graph models.

One approach is to postulate a library that contains *all* possible constructs and characteristics of graph models, then to use these library modules within the characterization framework. This approach is infeasible, since we know of no way to ensure that the toolkit is robust and complete. This approach would ultimately require that designers treat the framework as a library of tools to which any designer can add individual constructs.

The approach we take is to provide fundamental mechanisms that can be used to generate a spectrum of structural and visual properties, but which attempts to minimize the assumptions made about specific aspects of graph models. Within this framework we do not explicitly support such model specific constructs as subgraph, port, token, label, etc. Rather, we provide a set of general constructs and mechanisms that can be configured to describe and realize these constructs as well as a wide variety of other graph model constructs, characteristics, and behaviors. This approach differs from the library approach in that no particular components are provided, yet it is easy to construct many arbitrary components with the facilities that are provided. As we are unable to prove that our toolset is complete, we argue for its completeness by describing the rationale behind the framework, and by illustrating its usage with examples.

We concentrate on two properties of the framework: simplification of the constructs which make up a model, and the development of general mechanisms that act on these constructs. These two aspects will allow the characterization framework to support a wide variety of complex graph models.

4.1 Framework Constructs

The general idea of using a toolkit to construct the data model is well supported by objects and type hierarchies, e.g., see [8]. The object-oriented paradigm allows one to encapsulate constructs and behavior in the toolkit (type hierarchy), and to be able to adapt and use those constructs and their associated behavior through the use of subclassing, inheritance and type polymorphism.

A graph model can be described as a set of object types (classes). An instance of the model is then a collection of objects instantiated from these object types. The characterization framework incorporates basic classes for attributed **GraphElement** types, **Entity**, and **Relation**. *All* constructs and relationships within a graph model are cast into this entity/relaiton framework. Entities represent atoms (and molecules) within a graph model, and a relation defines some meaningful relationship among objects within the model. The objects that a relation connects may be entities or other relations. By abstracting all model specific constructs and relationships into this entity/relation framework we have a simple mechanism to describe and view the component structure of graph models. It frees us from thinking about specific model constructs such as subgraph, token, node, port, whose behavior is generalized into the entity/relation framework.

There exists entity and relation types that are common across all graph models. These common graph elements are concrete reflections of constructs that exist within the conceptual notion of a graph model. For example, we represent the generic behavior of a graph by an entity of type **Graph** within the framework. An instance of a **Graph** entity represents the graph as a whole. The **Graph** instance contains behavior to control and manipulate the elements that make up the graph. The conceptual relationship of an element *belonging* to a graph is represented by the relation type **MemberOfGraph**. Instances of this relation type are used to connect the instance graph object to the objects that make up the graph. Furthermore the conceptual notion of an element containing an attribute is reflected by the relation type **ElementToAttr**.

To illustrate the use of the entity/relation framework we can define a particular interpretation of the VERDI graph model discussed in Section 3. The set of entities, E, and the set of relations, R, which make up the VERDI graph are defined as follows.

E = {VERDIGraph, Team, Role, Place, Return, Box, Block, Token, Interaction, Label}
R = {MemberOfTeam, MemberOfRole, MemberOfInteraction, Link, ToToken, TeamToInteraction}

The **VERDIGraph** entity type is the concrete realization of the abstract concept of the graph. An instance of this entity represents a particular VERDI graph (inheriting behavior from the base **Graph** class. This object instance controls and accesses the set of constituent objects which make up the graph as a whole. Instances of the relation type **MemberOfGraph** are used to reflect the abstract relation of belonging to a graph. These relations connect the **VERDIGraph** instance to the set of objects that belong to the graph.

Instances of the **Interaction** entity type represent the N-party interaction within VERDI. This entity is used to group a set of **Boxes** into an interaction. Instances of the relation type **MemberOfInteraction** exist between the **Interaction** object and the **Box** objects which make up the **Interaction**. Instances of the relation type **TeamToInteraction** exist between a **Team** and the set of **Interactions** associated with that **Team**.

The **Label** entity type is used as an attribute for those object types that require a textual label. In this particular case the entities **Team**, **Role**, and **Box** contain an attribute of type **Label**. The relation type **ElementToAttr** is used to provide a relation between an attribute of a graph element

and the graph element itself. In this example, instances of type **Team**, **Role**, and **Box** would have an incident relation of type **ElementToAttr** that connects the objects to their **Label** instances.

The relation type **ToToken** defines the relation between those entities that may contain a **Token** object and any **Token** object that exists on the entity. The remaining entity and relation types in this example specification represent their respective VERDI constructs as discussed in section 3.

4.2 Framework Mechanisms

The second important aspect of the framework is the idea of allowing various *mechanisms* to be applied within the general entity/relation framework. The behavior of these mechanisms can be controlled by the language/system designer to achieve the desired model behavior.

These general mechanisms allow one to define particular structural, visual, and system behaviors for the general set of entities and relations that make up the constructs of a graph model. It is through these mechanisms that one can conceptually transform the general entity/relation framework into model specific constructs. We view model specific constructs such as subgraph, token, label and port as entities and relations. One uses these mechanisms to achieve the particular desired behavior of these constructs, i.e., to convey the implicit meaning of those constructs.

By viewing a graph model in general terms of entities and relations one is able to think about mechanisms that are useful for entities and relations as a whole and not be distracted by model specific issues. If a mechanism is useful for the general case of entities and relations, then it also useful for the specific cases of subgraphs, tokens, ports, labels, etc. One can also observe model specific behavior and generalize that behavior into the entity/relation framework. This generalized behavior can then be applied to achieve other specific behaviors.

These mechanisms allow us to minimize the assumptions we make regarding graph model behavior yet still provide the support for various types of functionality if it is desired. For example, the assumption could have been made that when an entity is deleted all of the incident relations are also deleted. This assumption for a particular behavioral characteristic of graph models would have limited the applicability of the framework. It would have disallowed any alternative behavior desired by the language/system designer. By providing a general mechanism regarding the existence of entities and relations whose behavior can be controlled by the language/system designer we allow a wider range of possible model and system behavior.

The mechanisms discussed here can be used to support a variety of specific model behaviors when they are applied to different model constructs. We have developed a set of general mechanisms that deal with such functionality as structural rules, event propagation, visual constraints and display control. Any of these mechanisms can be applied to any of the entities or relations within the framework, not simply the examples given here.

4.2.1 Structure Rules

The structure rules that we have developed address how entities and relations can be connected. We include three different structural rules as base mechanisms: incident relation, connected element, and incident element. The incident relation rule defines the legal number and types of incident relations of a graph element. The connected element rule defines the legal number and types of connected elements of a graph element. The incident element rule defines the allowable head and tail pairs of a relation.

One can define a wide variety of graph model syntax using this set of rules. For example, this rule can be used to specify the VERDI rule that a **Role** must have one and only one **Place**. The incident relation rule can be used to specify that VERDI **Team** entities may only have incident relations of type **MemberOfTeam**, **MemberOfGraph**, and **ElementToAttr**. Applying the connected element rule to the **Graph** entity one can define the legal number and types of elements within the graph as a whole. Thus, many syntactic rule requirements for graph models can be satisfied using this set of simple rules expressed as mechanisms.

4.2.2 Event Propagation

The general event propagation mechanism is defined within a relation; the definition allows the designer to define *event dependencies* that allow one to control the propagation of selected actions along graph element/relation paths within a graph model. For each event type there exists a set of four flags within a relation: the flags define the scope (left operand, right operand, or both operands) of the event whenever it occurs in the left operand of a relation or the right operand of a relation. The event propagation mechanism is used to define existential dependency, location dependency, copy dependency, and cut dependency.

Existential Dependency Existential dependency propagates the occurrence of an element's destruction event. Using this dependency the designer can specify that the existence of a relation depends on the existence of the head (left operand) or tail (right operand) of the relation. For example, if the existential dependency on the head node is set in a relation then on deletion of the head node the relation is also deleted. One can also specify whether the existence of the head or tail is dependent on the existence of the other element. For example, in the VERDI graph model one may wish to delete the components of a **Role** on deletion of the **Role**. One may define this behavior using the existential dependency mechanism within the **MemberOfRole** relation that exists between the **Role** and its set of components. One could also use this mechanism within the **MemberOfGraph** relation to remove all of the elements of a graph on destruction of the graph.

Location Dependency Location dependency propagates the event of changing the location of some element. This can be used for having edges "track" nodes whenever the node is moved. In the VERDI example it may be desired to move the constituent elements of a **Team** object or **Role** object when the object is moved. Using this dependency with the **ElementToAttr** relation between the VERDI entities and the **Label** entity one can cause the **Label** object to move when its associated object is moved. In combination with the **MemberOfGraph** relation one can shift the entire graph by moving the **Graph** object.

Copy Dependency The copy dependency propagates the event of copying an element. In the VERDI model, one could specify that copying a **Role** implies copying all of the elements of a role. Copying a **Place** implies the copying of any **Token** on that **Place**.

Cut Dependency Cut dependency is similar to existential dependency. The affected elements are removed, but not deleted, from the graph.

4.2.3 Visual Constraints

We have developed three visual constraints: containment, location, and display. (These constraints explicitly appear as part of the data model, but refer to the visual representation.) They are specified within relations and define some constraint on the visual state of one element based on the visual state of the other element.

Containment Constraint A containment constraint specifies that the affected incident element of the relation visually contains the other element of the relation. In VERDI, the relations of type **MemberOfTeam** and **MemberOfRole** specify a containment dependency for the **Team** and **Role** objects respectively.

Location Constraint The location constraint specifies that the location of the affected element of a relation is constrained to some logical viewing area with respect to the location of the other element of the relation. Using this constraint within the **ElementToAttr** relation one could specify the layout of the *Label* attribute for the **Team**, **Role**, and **Box** entities within VERDI. One could also define the location of the **Token** instances using the **ToToken** relation. One could constrain parts of an entire graph to particular areas of the screen using the location constraint within the **MemberOfGraph** relations.

Display Constraint The display constraint defines that the actual display of one element is dependent on the display of the other element. It also defines that the display of the relation is dependent on the display of the connected elements. For example, one could use this constraint to specify that when a **Team** instance is not displayed then the **Role**s which make up the **Team** are also not displayed. Likewise one could apply this constraint to control the display of the components of a **Role** based on the display state of the **Role**.

4.2.4 Display Control

Display control provides finer control over what is being displayed than the display constraint. Each relation contains flags that define whether the head and tail of the relation can be shown. For a graph element to be shown none of its incident relations may have the display control flag associated with the element set. This control mechanism is placed within a relation because it allows one to have more flexible control over the display of an element, i.e. more than one relation can control the display of a single element. Using this control mechanism within the **MemberOfGraph** relations one could develop a filtering mechanism that controls the display of certain elements of the graph based on element type. In the VERDI example one could turn on or off the **Label** objects using the display control within the **ElementToAttr** relation. Also within VERDI one could turn on or off the display of the **Roles** within a **Team** or the elements of a **Role** within a **Role**. A visual abstraction hierarchy could also be implemented using this mechanism. The display control mechanism offers the ability to develop a wide range of mechanisms that are useful for managing, manipulating and viewing graph models.

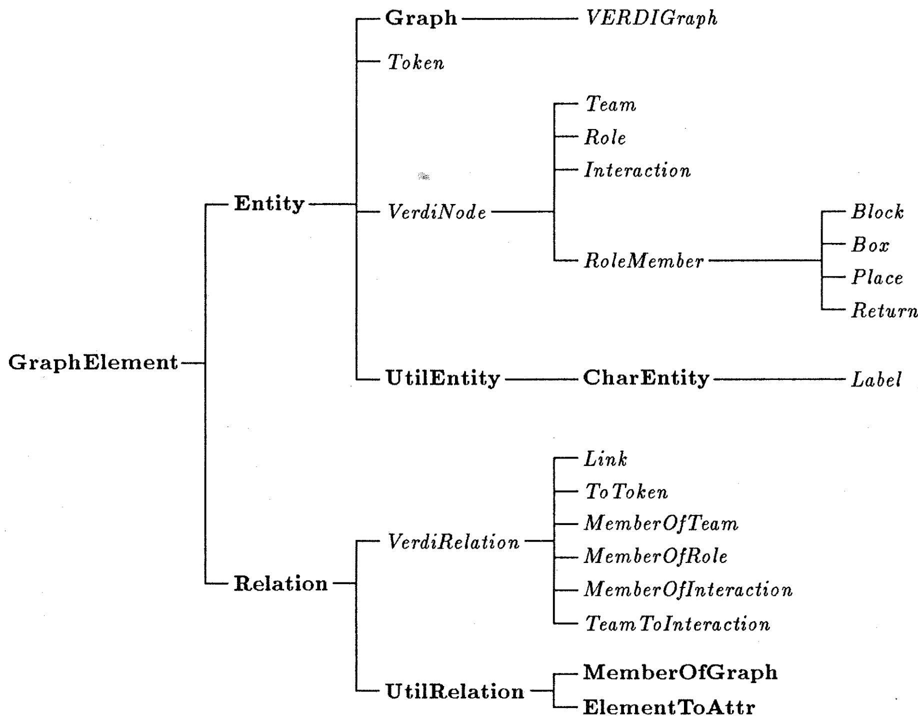


Figure 5: Framework Hierarchy

4.3 Framework Realization

We adopt a type hierarchy based approach in structuring the characterization framework. Figure 4.3 shows elements of the base type hierarchy and how a particular configuration of the elements which make up the VERDI example discussed above fit into the hierarchy. The element types shown in **bold** are those that are common to all graph models within the framework. The element types shown in *italics* are the specific elements defining the VERDI example. The base elements of the hierarchy are the **Entity** and **Relation** types. By deriving all types that make up a graph model from these base types we can manipulate all model types as either **Entities** or **Relations**. The type hierarchy provides the structural framework for describing a graph model and reflects what can be viewed as a naturally occurring hierarchy that graph models tend to exhibit. The inheritance mechanism within the type hierarchy allows one to aggregate the common characteristics of the constructs of a graph model into more abstract constructs thus providing a method of organizing the model in a logical fashion. The type hierarchy based approach provides a smooth path from the conceptual characterization framework to the instantiation of the framework within our ongoing system development efforts.

5 Summary

Visual language systems implicitly depend upon a data model with an appropriate representation and an underlying system. In Section 3, we described four visual systems published in the literature, each with a sophisticated data model and visual representation. We then explained our characterization framework for expressing a graph model definition for data models, including structural and visual properties of the language. We argue that the framework is a useful mechanism for describing a spectrum of visual languages which have an underlying data model that can be expressed as a graph model. The framework allows a designer to specify the syntax and semantics of the model without relying on an undue amount of formalism (e.g., graph grammars, attribute grammars, etc.).

While it is possible to define the syntax of the graph model using a graph grammar (e.g., see Beguelin's definition for the Phred visual language [6]), the characterization framework employs a type hierarchy with entities and relations, and the ability to define attributes using base classes. Besides easily defining the syntactic structure of the data model, the framework enables one to define a more complex set of constraints, dependencies, rules, and behaviors for the data model – rules of the complexity required to describe Raddle and the other implicit languages mentioned in Section 4. We have restricted our examples to Raddle and VERDI in the interest of conciseness for the paper, but the nontrivial relations of the other data models could also be defined using the characterization framework.

The characterization framework could be constructed using an arbitrary mechanism, e.g., a rule-based system; we have chosen to use a type hierarchy since it can be easily refined to specify new behavior on the basis of existing definitions. It is also possible to automatically define significant blocks of software to implement the behavior from relatively simple and concise directives.

In other research (to be reported elsewhere) we elaborate on our use of the characterization framework to define the data model and parts of the visual representation for a system that produces rapid prototypes of visual language editors. Briefly, the data model specification is a set of directives to the characterization framework. The directives and the framework are used to generate C++ software to represent model-specific parts of the graphic editor. These parts support such functionality as internal model storage, model I/O, consistency, event propagation, and model representation.

The generated model specific components are combined with a graph editor template to form the model specific graph editor. Drawing from the linguistic model of human-computer dialog discussed in Section 1, the graph editor template encapsulates the form component of the input language and translates it into the meaning component encapsulated in the data model. Within our system development efforts we employ a generic editor template that encapsulates a fixed input language form and a fixed translation between the input language form and meaning. If the system designer requires a variant of these fixed components than they can be modified to support the desired functionality. The specification and realization of the input form and translation between between input form and meaning is an area that is currently being addressed in [17, 13] among others.

Successful visual language systems have a representation that is familiar to users in a particular domain. Cognitive and behavioral properties of the users' domain cannot always be captured entirely within the visual representation, which leads to the notion of an underlying data model. It is tempting to construct visual language systems that do not explicitly separate the represen-

tation from the data model, since that can be done without reasoning through the data model behavior. In contrast, we have described a characterization framework that explicitly separates the representation, data model, and system functionality, and which encourages the designer to design the data model before focusing on the details of the system and presentation. The approach allows visual language designers to build model-independent language support facilities that can be used with a variety of data models, and for each data model to be used with a variety of different representations.

References

[1] Aaron Marcus and Andries van Dam. User-interface developments for the nineties. *IEEE Computer*, 24(9):49–57, September 1991.

[2] Kenneth E. Ayers. The MVC paradigm in Smalltalk/V. *Dr. Dobb's journal : Software tools for the professional*, 15, November 1990.

[3] Paul S. Barth. An object oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2), April 1986.

[4] Guiseppe Di Battista, Amedeo Giammarco, Guiseppe Santucci, and Roberto Tamassia. The architecture of diagram server. In *Proc. IEEE Workshop on Visual Languages (VL'90)*, pages 60–65, 1990.

[5] Stephen Beer and Ray Welland. Software design automation in an IPSE. In *1st European Software Engineering Conference*, 1987.

[6] Adam. L. Beguelin. *Deterministic Parallel Programming in Phred.* PhD thesis, University of Colorado, Department of Computer Science, May 1990.

[7] Jacques Bertin. *Semiology of Graphics.* The University of Wisconsin Press, 114 North Murray Street, Madison, WI 53715, 1983. Translated by William J. Berg.

[8] Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[9] Raymond J. A. Buhr, Gerald M. Karam, Carol J. Hayes, and C. Murray Woodside. Software CAD: A revolutonary approach. *IEEE Transactions on Software Engineering*, 15(3):235–249, March 1989.

[10] Stephen Casner. Building customized diagramming languages. In Shi-Kuo Chang, editor, *Languages and Information Systems: Visual Languages and Visual Programming*, pages 71–95. Plenum Press, 1990.

[11] Shi-Kuo Chang, Michael J. Tauber, Bing Yu, and Jing-Sheng Yu. A visual language compiler. *IEEE Transactions on Software Engneering*, 15(5):506–525, May 1989.

[12] Claudia Crimi, Angela Guerico, Guiliano Pacini, Genoveffa Tortora, and Maurizio Tucci. Automating visual language generation. *IEEE Transactions on Software Engneering*, 16(10):1122–1135, October 1990.

[13] James Foley, Won Chui Kim, Srdjan Kovacevic, and Kevin Murray. Defining interafces at a high level of abstraction. *IEEE Software*, pages 25–32, January 1989.

[14] Ira R. Foreman. On the design of large, distributed systems. In *Proceedings of the First International Conference on Computer Languages*, October 1986.

[15] Ed Freeman and Clayton Lewis. CU/USW research collaboration mid-year report: 1991, 1991.

[16] Ed Freeman and Clayton Lewis. The CU/U S West direct computer usage workshop, January 1992.

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

[18] Michael L. Graf. A visual environment for the design of distributed systems. In *Proceedings of the 1987 International Workshop on Visual Languages*, pages 330–343, August 1987.

[19] Minoru Harada and Tosiyasu L. Kunii. A recursive graph theory - as a formal for a Visual Design Language. In Tadao Ichikawa, editor, *IEEE 1984 Workshop On Visual Languages*, pages 124–135, December 1984.

[20] Richard Helm, Kim Marriott, and Martin Odersky. Building visual language parsers. In S. Robertson, G. Olson, and J. Olson, editors, *CHI '91 Human Factors in Computing Systems*, pages 105–112. Addison Wesley, April 1991.

[21] William A. Jindrich. Foible: A framework for visual programming languages. Master's thesis, University of Illinois, Department of Computer Science, Urbana-Champaign,Illinois, 1990.

[22] Anthony Karrer and Walt 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.

[23] Steve Laufmann, Rick Blumenthal, Loren Sylvan, and Ed Freeman. GNMS - A system for modeling netwrok structures. Technical report, US West Advanced Technologies, Boulder, Colorado, 1988.

[24] Stefano Mannucci, Bruno Mojana, Maria Cristina Navazio, Valerio Romano, Maria Carla Terzi, and Piero Torrigiani. Grapsin:A structured development environment for analysis and design. *IEEE Software*, pages 35–43, November 1989.

[25] J. Marks. A syntax and semantics for network diagrams. In *Proceedings of the IEEE 1990 Workshop On Visual Languages*, pages 104–110, December 1990.

[26] Jeffrey D. McWhirter. A framework and system for the specification and instantiation of graph models, 1991.

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

[28] Steven P. Reiss. Working in the ¡garden environment for conceptual programming. *IEEE Software*, pages 16–27, November 1987.

[29] Robert V. Rubin, James Walker II, and Eric J. Golin. Early experience with the visual programmers workbench. *IEEE Transactions on Software Engineering*, pages 1107–1121, October 1990.

[30] Shin Takahashi, Satoshi Matsuoka, Akinori Yonezawa, and Tomihisa Kamada. A general framework for bi-directional translation between abstract and pictorial data. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, November 1991.

[31] Walter F. Tichy and Francis J. Newberry. Knowledge-based editors for directed graphs. In *1st European Software Engineering Conference*, 1987.

[32] 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.

[33] Sterling Wight, Wallace Feurzeig, and John Richards. Plurbius: A visual programming environment for education and research. In *IEEE Workshop on Languages for Automation: Symbiotic and Intelligent Robots*, pages 122–128, August 1988.