Visualizing Information Repositories on the World-Wide Web

Mark Minas
Computer Science Department (IMMD II)
University of Erlang en
Martensstr. 3, 91058 Erlangen
Germany
minas@informatik.uni-erlangen.de

Leon Shklar
Bell Communications Research*
445 South St.
Morristown, NJ 07960
U.S.A.
shklar@cs.rutgers.edu

Computer Science Department
Rutgers University
New Brunswick, NJ 08903
U.S.A.

Abstract

The main objective of the proposed high-level *Visual Repository Definition Language* is to enable advanced Web presentation of large amounts of existing heterogeneous information. Statements of the language serve to describe the desired structure of information repositories, which are composed of metadata entities encapsulating the original data. Such approach helps to avoid the usual relocation and restructuring of data that occurs when providing Web access to it. The language has been designed to be useful even for unexperienced programmers. Its applicability is demonstrated by a real example, creating a repository of judicial opinions from publicly available raw data.

1 Introduction

Information technology is commonly predicted to be expanding faster than any other. Large amounts of data are already available on international data-networks, e.g., the Internet and, in particular, the World-Wide Web (WWW). Two main categories of data have to be distinguished in this context:

(1) Data already prepared for the WWW, i.e., formatted using Hypertext Mark-up Language (HTML), etc.

(2) Unformatted, heterogeneous data, e.g., legacy data, or data processed not primarily for WWW use, e.g., judicial opinions from the U.S. Supreme Court.

Until recently, the only way of homogeneously integrating the second kind of data into the WWW was to reformat the data and place it at a WWW site. Such approach is not very practical because of the amount of human and computing resources it requires for the initial conversion and maintenance of information. As a solution, we have presented *InfoHarness*—a system that provides rapid access to large amounts of new and existing heterogeneous information through WWW browsers without any relocation or restructuring of data [14]. *InfoHarness* imposes logical structure on raw data by analyzing it and generating metadata to encapsulate the original information. *InfoHarness* offers an opportunity not only to integrate existing heterogeneous information into the WWW but also to support new sophisticated presentation of data already available on the Web. *InfoHarness* can thus be looked at as a user-adjustable, parameterized, high order filtering scheme for arbitrary information.

Initially, the generation of *InfoHarness* metadata entities, i.e., information on how the original data is to be interpreted, filtered and composed, has been controlled by a textual modeling language. In this paper, we present a new visual language VRDL (Visual Repository Definition Language) replacing the textual language. There are two main reasons why the introduction of the new visual front-end serves to benefit the usability of the language:

(1) VRDL is easier to comprehend than the original textual language. The language design is inspired by Nassi-Shneiderman diagrams which are quite popular when teaching programming to novice programmers. Although results from "real-life" experiments are not yet available, we expect non-programmers to be comfortable with our visual language.

(2) Using an automatic generator of diagram editors [11], we have built a graphical editor dedicated to syntactic editing in VRDL. This way, the user gets maximal help when using the language.

In the following section of the paper we briefly discuss the *InfoHarness* object model and the repository definition language. Section 3 gives a description of our work on applying *DiaGen* to mapping the language constructs into

*Now with Pencom Systems, Inc., 40 Fulton St., New-York, NY 10038, U.S.A.

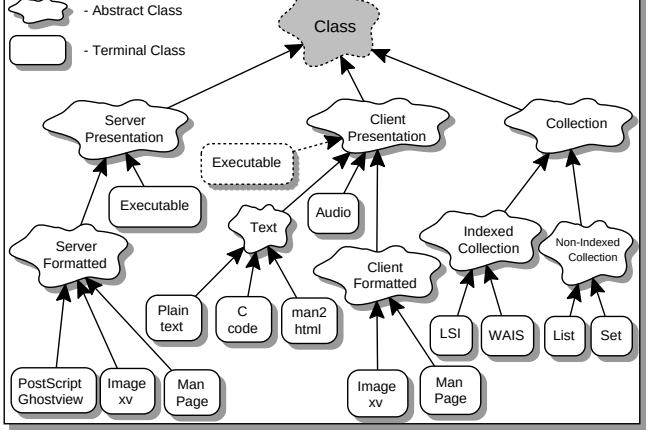

Figure 1: *InfoHarness* class hierarchy

interactive diagrams. Section 4 provides a discussion of related work. We conclude the paper with a brief summary and our future plans.

2 Building and Maintaining Information Repositories

An important advantage of our approach is in providing access to a variety of heterogeneous information without making any assumptions about its location and representation. This is achieved by generating metadata and associating it with the original information. We begin with describing the object model, proceed to a discussion of method sharing between *InfoHarness* objects, and conclude with reviewing statements of the textual Information Repository Definition Language (IRDL).

2.1 Object Model

The most basic concept in our approach is that of an encapsulation unit, which is defined as a metadata entity that encapsulates a portion of the original information of interest to end-users. An encapsulation unit may be associated with a file (e.g., a *man page*), a portion of a file (e.g., a C function), a set of files (e.g., a set of related judicial opinions), or an operation (e.g., a database query). For example, a C file and a function that occurs in this file may be encapsulated by different units because, in different contexts, each may present a unit of interest.

An *InfoHarness* Object (IHO) is defined recursively as one of the following:

(1) A simple object, composed of a single encapsulation unit.

(2) A collection object, composed of a set of references to other *InfoHarness* objects (its children).

(3) A composite object, combining a simple object and a set of references to other objects.

Simple and composite objects always store the logical address of the encapsulated data, together with typing information that determines decompression, decoding, decryption and data presentation methods (the latter is always present). Each object may contain an arbitrary number of additional attributes (e.g., owner, last update, security information, etc.). Composite IHOs also contain references to other objects (their children). A sample composite object may encapsulate this paper's abstract, combined with a set of references to simple objects that encapsulate text, *html*, and *postscript* versions of the full paper.

Collection objects are composed of references to their child objects and may also point to independent heterogeneous indices constructed from the portions of data encapsulated by these child objects.

2.2 Method Sharing

This section describes the *InfoHarness* class hierarchy (Fig. 1). The abstract class hierarchy has been constructed to support the execution of queries against independent indices and the presentation of the results as well as to support the run-time presentation of encapsulated information. The abstract classes provide method sharing between groups of terminal classes, which may be instantiated as *InfoHarness* objects. Examples of terminal classes include encapsulators of external viewers and indexing tools.

Each instance of the *Collection* class stores a set of parent-child relationships between the *InfoHarness* objects. In addition, instances of the *Indexed Collection* class are associated with a physical index that is used at run-time

to select members of the collection. The distinction between the *Server Presentation* and the *Client Presentation* classes is in the execution site of the presentation tools while the rationale for their subclasses is in the choice of these tools.

The *Server Presentation* abstract class helps group objects, for which the encapsulated information is accessed at run-time by running a process at the server. The subclasses of this class are *Server Formatted Data* and *Server Executable*. The subclasses of *Server Formatted Data* provide access to raw data by executing external viewers at the server but displaying windows at the client. Instances of the class *Executable* serve to encapsulate application programs that get executed at the server. The subclasses of the *Client Presentation* class include *Client Formatted Data*, *Text*, and *Audio*. For instances of terminal subclasses of *Client Presentation*, data is first transferred to the client and then accessed by running external viewers at that client. For security reasons, we did not initially provide support for the *Client Executable* class, but have recently started using the Java [8] technology to make it available.

Most of the data types may be defined either as a subclass of *Server Presentation* or as a subclass of *Client Presentation*. The exceptions are *Audio* and *Text* which are always defined by instantiating the *Client Presentation* class.

We say that the abstract class hierarchy in *InfoHarness* is *stable* because we do not foresee any immediate need for additional abstract classes to model different kinds of presentation. Whenever an appropriate terminal class is defined, it inherits data access and representation methods from an existing abstract class. This, of course, does not preclude evolutionary changes of the class hierarchy. The *InfoHarness* class hierarchy is *open* in that new terminal classes may be defined to accommodate the vast variety of information.

2.3 The Repository Definition Language

This section presents the basics of the textual language which is used as a backbone for our visual language described in Section 3.

The two main components of the language are responsible for introducing new types and for building information repositories. The type definition component is responsible for generating methods that serve to support new kinds of data and new indexing technologies. It is intended to be used primarily for applying *InfoHarness* to new domains and is not discussed in this paper. Conversely, the structure definition component is responsible for generating metadata entities that impose the desired logical models on raw data. It is intended for a wide range of repository administrators and even sophisticated end-users who want to create their own repositories. To support the latter, we have concentrated our efforts on providing an advanced interactive graphical interface.

The structure definition component provides an executable specification for building and maintaining information repositories. It combines features of a structured programming language with non-procedural support for data encapsulation, set operations, and content-based indexing. A program is a sequence of declarations followed by a sequence of statements:

```
<program> := BEGIN; <body> END;
<body> := <decls> <stmts>
<decls> := <decl>; [<decls>]
<stmts> := <stmt>; [<stmts>]
```

2.3.1 Declarations

A declaration may be a type declaration or a variable declaration:

```
<decl> := <type_decl> | <var_decl>
```

Type declarations are required for both collection types and data types. Collection types are associated with particular indexing technologies (WAIS, LSI, or user-defined). As for data types, we distinguish between the encapsulation types (e.g., `TXT`, `C`) that help utilize proper data encapsulation methods, and the presentation types that ensure proper run-time presentation of the encapsulated data (as well as proper pre-processing when building indexed collections). Of course, methods for all declared types must be available from the type library. The generation of methods and their association with new type names is discussed in [15].

```
<type_decl> := <type_name> [,<names>]
<type_name> := <encap_name> | <pres_name> | <encap_pres_name> | <coll_type>
<encname> := ETYPE: <name>
<presname> := PTYPE: <name>
<encap_pres_name> := TYPE: <name>
<coll_type> := TYPE INDEX: <name>
<names> := <name> [,<names>]
```

Within a variable declaration, the SET qualifier determines whether the interpreter returns a handle to a single element or to a set of elements. If the handle identifier represents a set, it may be iterated upon using iteration statements that are discussed later in this section. The language supports two built-in types: IHO (for the *InfoHarness* objects) and STRING. At this time, values of individual attributes are always treated as strings, therefore, we did not introduce any arithmetic types. Our current work on supporting complex attribute structures is discussed in [13].

```
<var_decl> := VAR [SET] <built-in>: <vars>
<built-in> := IHO | STRING
<vars> := <var> | <var>, <vars>
<var> := <item> | <set>
<item> := <iho> | <string>
<set> := <iho_set> | <string_set>
```

2.3.2 Expressions

At this time, object and string expressions (which are omitted here) are the only kinds of expressions possible (both may be either set or item expressions):

```
<expr> := <item_expr> | <set_expr>
<item_expr> := <item> | <iho_expr> | <string_expr>
<set_expr> := <set> | <iho_set_expr> | <string_set_expr>
```

As discussed in Section 2.1, the encapsulation of raw data is performed through encapsulation units, which are contained within simple and composite objects. The encapsulation process is controlled by type methods that determine the desired interpretation of raw data. For example, a C file may be treated as a text file and encapsulated by a single object. Alternatively, it may be treated as a C program, which would result in generating a set of objects, each of which encapsulates an individual function.

Encapsulation is controlled by the encapsulate statement of the language. It requires the encapsulation type, and either one or more locations of raw data, which are either legal Uniform Resource Locator (URL) expressions [2], or previously defined simple or composite objects. Note that the use of an object in this context would only make sense if it had been created with a different encapsulation method.

```
<iho_set_expr> := <encapsulate>
<iho_expr> := <combine_expr> | <index_expr>
<encapsulate> := ENCAP[SULATE] { <encapsulation_type> | * } [ <presentation_type> ]
                 { <URLs> | <iho> | <iho_set> }
```

As defined in Section 2.1, collection objects do not directly encapsulate raw data, but are composed of references to other objects. While simple objects may be created directly from encapsulation units, a collection object may only be created from a set of objects. If a collection object is indexed, it contains information about the index location and about the proper query method. The creation of collection objects is performed by the index operation that takes a collection type (LSI, WAIS, etc.), a set of objects, and a desired location:

```
<index_expr> := INDEX <collection_type> <iho_set_expr> <file URL>
```

Creation of composite objects is performed through the combine operation that takes an object and a set of references to other objects:

```
<combine_expr> := COMBINE <iho> <iho_set_expr>
```

Given the provision for sets in the language, there has to be a way to compute set unions and intersections. The union operation is used to merge two sets, to add an item to a set, or to convert a single item into a one-item set; the intersection operation is used to compute common members of two sets:

```
<set_expr> := "{" <item> [,<set_expr>] "}" | "{"<set_expr>,<set_expr>"}"
            | "{"<set_expr>&<set_expr>"}"
```

2.3.3 Statements

A statement may be an assignment, a set iteration, or an input/output operation. Assignment operations may be performed on both item expressions and set expressions. Selective access to individual set elements is provided by the iteration statements. Simple input/output directives promote the reuse of existing metadata entities.

```
<statement> := <assign> | <forall> | <input_output>
```

Assignment statements support the assignment of individual items and sets of items, where an item is either an object or a string. It is possible only to assign set expressions to set variables and individual items to item variables. Object expressions may be only assigned to object variables and string expressions may be only assigned to string variables.

```
<assign> := <var> = <expression>
```

Selective access to set members is provided by the `forall` statement of the language. The `SUCH THAT` clause of the statement supports selectivity by excluding members that do not meet the boolean combination of conditions. Each condition may be either a logical comparison or a pattern matching expression. Regular expressions are defined as usual and are not further explained.

```
<forall> := FORALL <item> IN <set> [SUCH THAT <conditions>] "{" <statements> "}"
<conditions> := <condition> | (<conditions>) | <condition> <bool_op> <conditions>
<condition>  := <item> <comp_op> <item> | <string> <match_op> <regular_expr>
<bool_op>  := AND | OR | NOT
<comp_op> := == | !=
<match_op> := =~ | !~
```

The role of the input/output statements is to support reading input information, as well as storage and retrieval of generated metadata entities. Notice that any legal URL may be used to specify an input location but only local URL expressions [2] may be used to specify output locations.

```
<input_output> := <write> | <read>
<write> := WRITE <vars> [<file URL>]
<read> := READ < vars> [<URL>]
```

2.4 Example

In this section, we discuss how to best search and present the judicial opinions from the U.S. Supreme Court that are available at `ftp.cwru.edu`. Here, information related to a single case may be distributed between multiple files. The example impressively demonstrates how information not prepared for the WWW can be effectively presented on the Web.

Given the location of the original information, the desired run-time presentation of individual cases, and the desired indexing technology, the following steps are required to generate the repository of the Supreme Court cases:

(1) Create simple objects that encapsulate individual judicial opinions (one per file). The encapsulation method should determine the case numbers for the opinions and store them as attributes of the encapsulating objects.

(2) For each object created in step one, find other objects related to the same case, encapsulate them together using the encapsulation type 'Case', and exclude them from any further consideration. The presentation method for this type should be responsible for generating internal hyperlinks to individual opinions and external hyperlinks to related information (the Supreme Court photo, brief biographies of the judges, etc.).

(3) Create an indexed collection of the objects created in step 2 using the Latent Semantic Indexing (LSI) technology [4].

These steps are implemented by the IRDL program in Fig. 2. The equivalent VRDL program is shown in Fig. 3. It assumes that the LSI indexing technology is supported and that the encapsulation and presentation methods for the types Court and Case are available in the *InfoHarness* type library.

The first statement of the program serves to encapsulate individual opinions located at `ftp://ftp.cwru.edu/hermes/ascii/` and assigns the generated set of simple objects to 'ItemSet'. The next two statements initialize 'Processed' and 'CaseSet' variables. 'Processed' is used to accumulate objects from 'ItemSet' that should be excluded from the further consideration, while 'CaseSet' is used to group together objects

```
BEGIN
  TYPE INDEX:  LSI;
  TYPE:        Court, Case;
  VAR IHO:     PrimItem, SecItem, LSI_Collection;
  VAR SET IHO: ItemSet, CaseSet, Processed, CaseItems;

  ItemSet =
       ENCAP Court "ftp://ftp.cwru.edu/hermes/ascii/";
  Processed = {}; CaseSet = {};
  FORALL PrimItem IN ItemSet SUCH THAT
       (PrimItem NOT IN Processed)
  {
    CaseItems = {};
    FORALL SecItem IN ItemSet SUCH THAT
     (ATTR PrimItem "CaseId" == ATTR SecItem "CaseId")
    {
      CaseItems = {SecItem, CaseItems};
    }
    Processed = {CaseItems, Processed};
    CaseSet = {COMBINE (ENCAP Case CaseItems) CaseItems,
               CaseSet};
  }
  LSI_Collection =
        INDEX LSI CaseSet "file:/tmp/db/SupremeCourt";
  WRITE ItemSet, LSI_Collection;
END;
```

Figure 2: IRDL program for representing the U.S. Supreme Court cases at `ftp.cwru.edu`

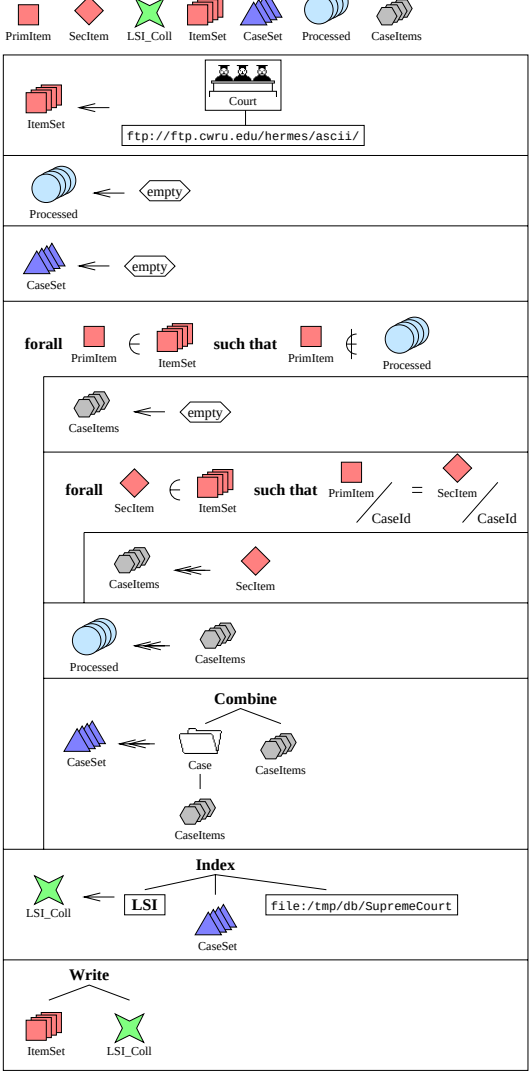

Figure 3: VRDL program for the program in Fig. 2

that encapsulate the cases. Next, the `forall` statement serves to iterate over the objects in 'ItemSet' and uses 'Processed' in the `such-that` condition to avoid assembling the same case for every member opinion.

Objects that are related to the same case are determined using the 'CaseId' attribute, which is set by the encapsulation method for the type 'Court'. All objects related to the same case are grouped together using the encapsulation type 'Case'. We then use the `combine` operation to create composite objects that both encapsulate all case-related information and contain references to simple objects encapsulating individual opinions. Finally, when all opinion objects are grouped together, an indexed collection is created for objects in 'CaseSet', and the collection object is assigned to the 'LSI_Coll' variable. 'LSI_Coll' and 'ItemSet' are stored; they implement the information repository properly formatted for providing sophisticated Web access to the original information.

Fig. 4 shows a Web page after searching for decisions with keywords *drugs*, *violent*, and *crime*, i.e., using the index referenced by the object stored in 'LSI_Coll'. The result of the query is a list of the members of the set in 'CaseSet'. Since each member is a composite object, we see not only a hyperlink for its content but also hyperlinks for the individual opinions. The dynamic Web page for the case object which is referenced by the first hyperlink in Fig. 4 is shown in Fig. 5.

3 VRDL

In this section, we describe the visual counterpart to IRDL, the Visual Repository Definition Language (VRDL) primarily designed for unexperienced users. The next subsection explains the sources of VRDL. Then, the graphical elements of VRDL are explained before we briefly introduce our VRDL editor.

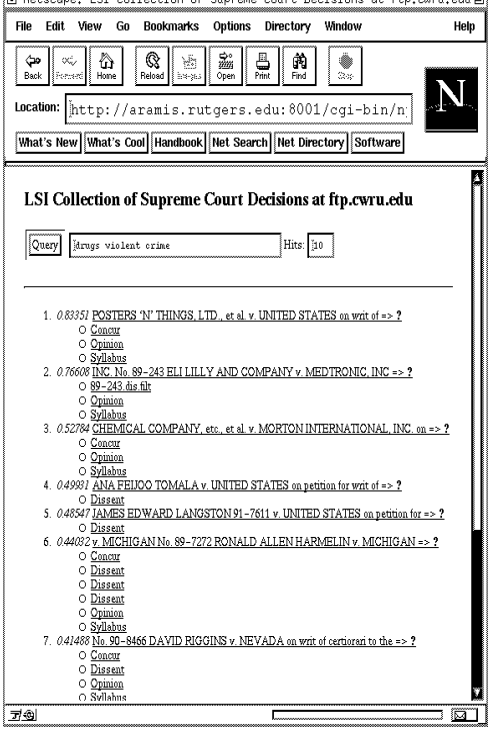

Figure 4: Query interface and selected member objects of the collection object 'LSI_Coll'

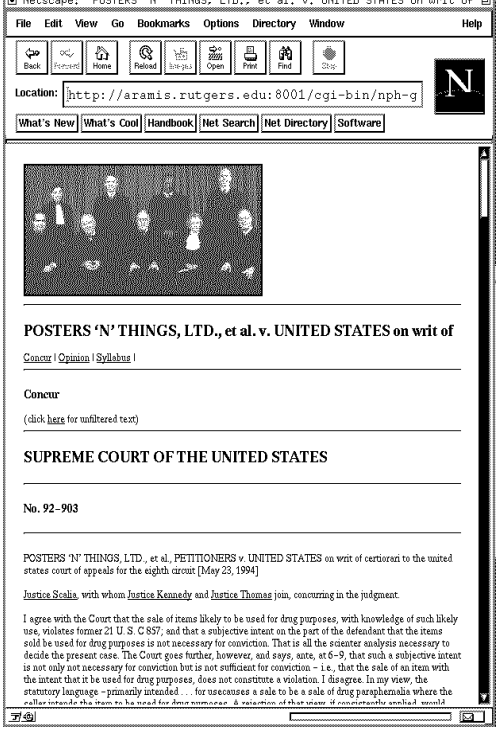

Figure 5: Web page for a case object in set 'CaseSet'

3.1 Visual Paradigm

For VRDL, we have chosen a concept similar to representing simple programs: Nassi-Shneiderman diagrams (NSDs) [12]. Each program statement is represented by a rectangle containing the statement. Sequences of statements are translated to stacks of corresponding rectangles. Loops are displayed as rectangles containing the loop control expression as well as the loop body. Moreover, particular graphical constructs are used for alternatives. Fig. 3 shows a visualization of the IRDL program in Fig. 2 inspired by NSDs. Correspondence of IRDL declarations as well as statements and their graphical representation, in particular, the appearance of the `forall` statement, is suggestive. Fig. 3 demonstrates visualizing the structure of IRDL programs (by mutually arranging rectangles) as well as individual statements. Before going into details, we discuss advantages of this approach:

(1) NSDs have proved to be useful for program presentation, especially when teaching algorithms. In particular, the popularity of NSDs in lessons for novice programmers holds promise for the task of making VRDL accessible to non-programmers.

(2) NSDs scale for large problems. They provide a bird-eye view when "zooming out" of the diagram. Contents of single statements might get lost, but the overall structure remains and even becomes more apparent. Furthermore, because of the their block structure, NSDs provide easy means of abstraction: complex blocks can be represented as simple statements. The complex structure is then elaborated in another NSD. In conventional programming languages, this is equivalent to using procedures.

(3) The structure of an IRDL program does not change when represented in VRDL. The essential difference is using the two-dimensional presentation and graphical symbols to simplify perception. However, text is not avoided in VRDL following the experience of the visual language community.

3.2 Graphical Elements of VRDL

In order to emphasize that VRDL is not a completely new language, but only employs a visual representation, we describe VRDL language elements in the same sequence as the corresponding IRDL statements in Section 2.3. We restrict our discussion to an informal one. The formal specification of the underlying constraint hypergraph grammar is omitted here for conciseness. A detailed discussion can be found in [9].

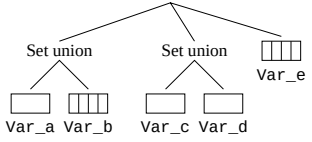

Figure 6: VRDL representation of complex set operations

3.2.1 Declarations

As restricted to the scope of this paper, types are simply enumerated in IRDL and thus redundant.[1] Type declarations are therefore omitted for the current version of VRDL. However, variable declarations are necessary and greatly simplified by using graphics. VRDL programs start with a declaration block. Each variable is represented by an icon along with its name. Shape and color provide easy distinctions between different variables. Set objects are indicated by stacks of icons. `IHO` is the default built-in type. Therefore, it is not specified in Fig. 3.

3.2.2 Expressions

Expressions have an inherent tree structure which is made explicit in VRDL. String constants and variables are primitive expressions and are presented, respectively, by plain strings and as described in Section 3.2.1. Structured expressions, e.g., `COMBINE (ENCAP Case CaseItems) CaseItems`, are displayed as trees with their keywords as root nodes. Fig. 3 shows the corresponding visualization of this example. Note that we have chosen a special syntax for representing the important `encap` statement. The encapsulation type is presented as an icon at the root of the expression tree. The icon has to be provided with the type in the type library. Another example of such a statement is the first one in Fig. 3.

Set union and intersection are also displayed as trees with 'Set union' and 'Set intersection' as root nodes. As shown in Fig. 6, the number of children is not fixed, and using the tree representation, complex set expressions are easily visualized.

3.2.3 Statements

VRDL program structure is visualized in an NSD-like manner. Sequences of statements are translated into a stack of rectangles, each representing one statement. Fig. 3 showing the VRDL representation of the IRDL program in Fig. 2 makes clear how different statement types are presented. Note that we have introduced a simpler syntax for the assignment statement when adding objects and sets of objects to sets. In this case, we are using a double arrow instead of a single arrow for regular assignment. Note also the visualization of the `forall` loop, which contains the loop body as a sub-structure and the loop head analogous to its IRDL representation.

3.3 Interaction with the Graphical Front-End

In other work, we have described *DiaGen*, a system that supports the automatic generation of graphical editors for specific applications from formal specifications [11]. VRDL and a graphical editor have been described by such a specification.[2] *DiaGen* has been used for generating a graphical editor for editing VRDL programs from this specification. The generated editor has the following main features:

(1) The editor provides syntax-directed editing. The user can select VRDL statements together with declarations from the menu bar. Complex programs are constructed by combining such statements in a direct manipulation style: the user grabs a statement and drags it to a new position where it is inserted. Only syntactically correct programs would be created.

The same principle is used for expressions: they are initially instantiated as default expressions that may later be replaced with menu bar selections. However, variables are defined in statements by dragging them from declarations or different locations within statements.

(2) The editor supports the direct manipulation in many situations. E.g., declarations may be reordered by dragging variables in a declaration from one position to another. Markers between the variables of declaration trees (and also in Read and Write statements; see Fig. 3) provide hints for where to release the variables.

[1] Type declarations are only redundant at the conceptual level. One way to use them in VRDL would be to establish an interface between the editor and the IRDL interpreter. Then, for example, when creating a new `ENCAP` statement, the editor may query the interpreter for legal types, and the type visualized by an icon at the root of the `ENCAP` tree may then be presented as a menu of these types.

[2] Details of this specification can be found in [9].

Several VRDL diagrams can be edited within the same editor. The user can arbitrarily move statements and declarations from one diagram to another one in a direct manipulation style.

(3) The editor provides – as each editor generated by *DiaGen* – an automatic layout. Rectangles representing program structure automatically adapt to growing or shrinking statement contents. But, it is also possible to adjust layout by moving lines. Automatic adjustments after such modification are guaranteed to be minimal [10].

(4) VRDL diagrams are translated to IRDL programs which are processed with the existing IRDL interpreter. Thus, the editor can be used to directly define information repositories.

(5) According to the current *DiaGen* implementation, the editor is running under Unix and X11/Motif. The prototypical (textual) specification for *DiaGen* has some 2,000 lines.

4 Related Work

The access and retrieval of heterogeneous information has historically concentrated on different application areas, including software reuse, digital libraries [6], geospatial data [5], etc. Software reusability now extends beyond code to include other software assets such as specifications, designs, test cases, plans, data, and documentation [1]. The construction of digital libraries and the modeling of geospatial data require assembling a variety of media types, both structured and unstructured, and consequently ensuring ease of access and manipulation. The basic trade-off for these applications lies in balancing the cost of constructing a storage system versus the cost of locating and browsing relevant resources.

One of the most important concerns in providing uniform and transparent access to information is uniformity. Short of a uniform data representation, which has not proved to be practical, the next best thing is a uniform data modeling approach. Various methodologies exist for capturing the internal structure of heterogeneous data, e.g., [18]. Representations within hypertext systems capture basic notions of objects (nodes) and relationships (links) which are used to build complex structures (e.g., hierarchies, arbitrary networks).

Research on modeling data within a dynamic environment has identified two concepts – the set concept and the type concept. These concepts adequately capture invariant properties of data within a database architecture [17]. Our research is aimed at developing a simple, yet powerful, language that supports modeling heterogeneous information based on the underlying notions of sets and types, and at making using the language simple enough for unsophisticated programmers. Hence, we have provided a visual language for specifying structure definitions because it is targeted for a wide audience of information modelers. Whereas the concept of a simple, declarative, textual language to support modeling is not new[3], the visual language is a unique feature increasing user-friendliness.

Furthermore, VRDL is related to visual programming languages and environments as well as visual programming. The visual approach has been quite successful for restricted domains and programming tasks as well as for teaching inexperienced users. Examples include attempts to generalize spreadsheets to real (visual) programming languages [3], and *Prograph*, a visual programming language and environment currently used for moderately sized software projects [7]. This success with casual and inexperienced programmers was the motivation for our design of VRDL. We used a Nassi-Shneiderman diagram representation [12], which is not (as far as we know) incorporated in prominent software products, but which successfully serves as a visualization aid in teaching novice programmers.

5 Conclusions and Future Work

We have combined two independent lines of research [16, 11] to produce VRDL, an interactive visual language, which supports generating object encapsulations of existing information. VRDL combines the flexibility of object encapsulation with the power and convenience of a simple interactive declarative language. VRDL diagrams, together with the raw data, determine the structure of information repositories.

IRDL was designed for programmers and has shown to be effective for building information repositories. Our objective in designing the visual language was its ease of use by non-programmers or inexperienced programmers. For this reason, we have chosen a Nassi-Shneiderman diagram-like representation, a visualization that has proved successful in teaching algorithms. We do not have experimental results on the VRDL acceptance by those user communities, but we are planning such experiments with *InfoHarness* users.

At this time, complete IRDL programs generated by the interactive editor are being passed on to our IRDL interpreter. We intend to provide a closer integration between the editor and the interpreter to support error recognition at

[3] For a detailed discussion of related languages see [14].

the initial stages of building and updating VRDL diagrams. The current VRDL front-end would be an X-Motif application. One intriguing possibility is to generalize *DiaGen* to also support generating Java [8] programs to provide a much closer integration with Web applications.

6 Acknowledgements

The work on legal applications is being performed jointly with L. Thorne McCarty at Rutgers University. The authors would also like to thank Robert Allen, Georg Heidenreich, and Christian Jacob for many helpful comments.

References

[1] V. Basili. Support for comprehensive reuse. *Software Engineering Journal*, pp. 303–316, 1991.

[2] T. Berners-Lee et al. World Wide Web: The information universe. *Electronic Networking: Research, Applications and Policy*, 1(2), 1992.

[3] M. Burnett and A. Ambler. Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual Languages and Computing*, 5:29–60, 1994.

[4] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Hashman. Indexing by latent semantic indexing. *Journal of the American Society for Information Science*, 41(6), 1990.

[5] Content standards for digital geospatial metadata. Federal Geographic Data Committee, June 1994.

[6] C. Fisher, J. Frew, M. Larsgaard, T. Smith, and Q. Zheng. Alexandria digital library: Rapid prototype and metadata schema. In *Advances in Digital Libraries*, pp. 173–194. Springer-Verlag, New York, 1995.

[7] E. Golin. Tool review: Prograph 2.0 from TGS systems. *Journal of Visual Languages and Computing*, 2(2):189–194, June 1991.

[8] J. Gosling and H. McGilton. The Java language environment: A white paper. Sun Microsystems, Mountain View, CA, May 1995.

[9] M. Minas and L. Shklar. Generating a graphic editor for VRDL using DiaGen. Internal report, Lehrstuhl für Programmiersprachen, Uni. Erlangen-Nürnberg, 1995.

[10] M. Minas and G. Viehstaedt. Specification of diagram editors providing layout adjustment with minimal change. In *Proc. 1993 IEEE Symp. on Visual Languages (VL'93), Bergen, Norway*, pp. 324–329. Aug. 1993.

[11] M. Minas and G. Viehstaedt. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *Proc. 11th IEEE Int. Symp. on Visual Languages (VL '95), Darmstadt, Germany*, pp. 203–210. Sept. 1995.

[12] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, Aug. 1973.

[13] L. Shklar and C. Behrens. Geoharness: A system for managing NASA's remote sensing data. Draft, 1996.

[14] L. Shklar, K. Shah, and C. Basu. Putting legacy data on the Web: A repository definition language. *Computer Networks and ISDN Systems*, 27(6):939–952, Apr. 1995.

[15] L. Shklar, C. Basu, and K. Shah. Portable information repositories. Draft, 1996.

[16] L. Shklar, A. Sheth, V. Kashyap, and K. Shah. InfoHarness: Use of automatically generated metadata for search and retrieval of heterogeneous information. In *Proc. 7th Int. Conf. on Advanced Information Systems Engineering (CAiSE '95), Jyväskylä, Finland*, LNCS 932, pages 217–230. Springer-Verlag, June 1995.

[17] J. Taylor. Toward a modeling language standard for hybrid dynamical systems. In *Proc. 32nd Conf. on Decision and Control*, volume 3, pp. 2317–2322, San Antonio, Texas, USA, Dec. 1993.

[18] J. Ter-Bakke. *Semantic Data Modeling*. Prentice Hall, 1992.

[19] G. Viehstaedt and M. Minas. Interaction in really graphical user interfaces. In *Proc. 1994 IEEE Symp. on Visual Languages (VL'94), St. Louis*, pp. 270–277. Oct. 1994.