VDL — a View Definition Language

Mats Johnson, Erik Borälv
Center for Human-Computer Studies
Uppsala University
Lägerhyddvägen 18
S-752 37 Uppsala
SWEDEN

VDL is a language for specifying user interfaces. The basic approach of the language is specifying attribute/value pairs for a hierarchy of user interface objects. The language is declarative and puts an emphasis on expressive power through the use of inheritance, parameters and variables.

1. Introduction

When describing a graphical user interface it is useful to use attribute/value pairs. The exact look and behaviour of a certain user interface object can be described using a set of such pairs. A complete user interface consists of a hierarchy of interface objects, and a description of such an interface can therefore be a hierarchy of sets of attribute/value pairs.

1.1. The X Resource Manager

An example of using attribute/value pairs for describing a user interface is the X Resource Manager, Xrm [4]. A specification might look like this:

```
top.foo.bar.foreground: red
```

The first part specifies where in the hierarchy this attribute/value pair belongs, in this case in the *bar* object, within the *foo* object, within the *top* object. The next part is the attribute name, *foreground*, followed by the value, *red*.

Strings in the hierarchy specification refer either to the names or the classes of the components.

In order to make the specification language more expressive, wildcards are allowed in the first part, the hierarchy specification. A “?” matches any single component name or class, and a “*” matches any number of components, including none.

When more than one specification would be applicable, the more specific take precedence. A specification using wildcards could look like this:

```
*bar.foreground: green
```

This means that any component called *bar* will have the *foreground* set to *green*, unless overridden.

Shortcomings. A specification of a user interface using the X Resource Manager becomes very large. The hierarchy specification to the left of the colon is repeated identically in a large number of lines. The line-oriented style makes the specification rather unreadable.

The expressiveness gained by using wildcards is not powerful enough. There are no “modules”.

It is normally not possible to specify the component hierarchy using Xrm. It is assumed that the widget hierarchy is created programmatically.

There is however a library called Wcl [7] which adds the possibility of specifying the hierarchy in a resource file. It basically works by adding the resource *wcCreate*, which defines the class name of the widget, and the resource *wcChildren* which has as its value a list of names of children to be created.

```
Mri.wcChildren:         push
*push.wcCreate:         XmPushButton
*push.labelString:      Hello World
*push.activateCallback: WcExit
```

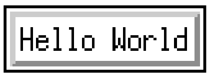

Figure 1: Example of a Wcl specification and the output it generates.

Xmt [5] is another library which adds the same capability in a somewhat different fashion.

1.2. Other tools

“Graphical” tools such as TeleUSE [6] allows the specification of the widget hierarchy in a declarative way, just as Xrm and Wcl. On the surface they seem to provide an attractive direct-manipulation interface to the user interface specification. There are usually several levels of access possible to the specification, as well as some kind of test mode where buttons can be pressed and menus pop up.

The highest editing level looks the same as the resulting interface will, but is not “live”, instead some editing is possible. The next level is a view of the widget tree structure indicating the names and types of the widgets. Finally there is a text representation of the specification (in principle the same level as Xrm with Wcl).

For setting attributes of widgets there is another view consisting of dialogue boxes with very long lists of attribute/value pairs. These dialogues are accessed through one of the hierarchy views.

The high level, direct manipulation, WYSIWYG editing is in practice only used on demos and exhibitions. The problem is that such views are simply not appropriate for the designer who need more information about and quick access to the widget hierarchy. The high level interfaces also cannot possibly provide direct access to the many attributes.

In practice the hierarchy view together with the attribute list dialogue boxes are used. This makes for lots of clicking and scrolling through lists of attributes. It is also difficult to achieve good modularisation and reuse.

User interface specifications are complex, almost as complex as computer programs. To cope with the complexity certain mechanisms are needed which do not lend themselves easily to graphic representation. Just as with programming languages it may be that it is more efficient to use a textual representation.

2. The View Definition Language

VDL uses the same basic concept as Xrm. The present implementation generates resource files for use with Wcl. In order to overcome the problems with the X Resource Manager, VDL adds structure, inheritance, parameters, variables and functions.

The actual syntax used in the examples, and in the current implementation is influenced by our use of Perl [3] as the implementation language. This syntax is not necessarily the best or the most elegant.

2.1. Attributes

The most basic construct in the language is the definition of a constant value for an attribute.

```
foreground:red
```

The value of the attribute *foreground* is defined to have the value *red*.

2.2. Structure

In VDL, the description of the sets of attribute/value pairs for the complete hierarchy of components forming a user interface is broken up into templates, or *styles*, describing a part of the hierarchy. A definition of such a style consists of a name, a parameter list and a sequence of definition items. It is an ordered sequence; items that come later in the sequence override earlier ones.

The simplest kind of definition item is one that defines a constant value for an attribute.

```
foo:-
  foreground:red,
  background:yellow.
```

In this example, a style *foo* is defined, with two definition items, specifying that *foreground* is *red* and *background* has the value *yellow*.

2.3. Values of other attributes

The value of an attribute can be defined to be the same as the value of another attribute.

```
foo:-
  foreground:val(background).
```

The value of *foreground* is defined to be the same as the value of *background*.

2.4. Variables

Variables are like attributes, except that they do not generate any output themselves, but only act as placeholders.

```
foo:-
  colour=red,
  background:val(colour).
```

In this example, the variable *colour* is defined with a value of *red*. The attribute *background* uses the variable *colour*. This results in *background* having the value *red*. There is no attribute *colour* generated.

2.5. Hierarchy inherit

Variables and attributes access values at the same level of hierarchy, unless an explicit argument is given to *val* indicating how many levels up in the hierarchy the value should be fetched from.

```
foo:-
  background:val(colour, 1).
```

Here *foo* is defined to have the same *background* as its parent component in the hierarchy.

2.6. Inheritance

A style can inherit all attributes (and variables) from another style.

```
bar:-
  foo, background:green.
```

Here the style *bar* is defined, inheriting attribute/value pairs from *foo*, and adding a pair of its own. Definitions later in the sequence take precedence over earlier ones, including inherited definitions.

2.7. Hierarchy

The hierarchical structure is specified in VDL by naming a child and providing a definition.

```
foo:-
  cld(a, bar),
  cld(b, bar, background:green).
```

This defines a style *foo*, describing a hierarchy with two children, named *a* and *b*. Both children inherits attributes from the style *bar*, but the second child *b* declares *background* to be *green*, overriding any value inherited from *bar*.

2.8. Bigger example of the above

2.9. Parameters

There is also the option of using parameters when inheriting. Parameter values are accessible inside the whole definition of a style, but not from anything inherited.

```
foo(bg):-
  foreground:red, background:val(bg).
bar:-
  foo(blue).
```

The style *foo* is defined to take one parameter, *bg*, and uses it for the value of background. The style *bar* inherits from *foo*, passing a parameter of *blue*. The result is that *bar* gets a *background* of *blue*, and a *foreground* of *red*.

2.10. Functions

Sometimes the value of an attribute needs to be a function of some other value.

```
foo:-
  font:[weight(val(font), bold)],
  foreground:
    [darker(val(background), 30)].
```

Here the *font* is defined to be the same as before, but *bold*, and the *foreground* to be the same colour as the *background*, but darker. In this example the function definition is expressed in Perl [3], since this is the implementation language of our prototype.

3. CAOS in VDL

VDL was designed to be useful on its own, but also to support the CAOS [1] model of dialogue design.

3.1. What is CAOS

CAOS is an implementation model for object oriented user interfaces. It extends the well-known PAC [2] model with more detailed knowledge of the application, resulting in less effort to implement dialogues, and more power in the underlying support system. This in turn results in less expensive user interfaces with higher quality, and increased consistency.

In the CAOS (and PAC) model there are three important parts: *presentation*, *abstraction* and *control*. The presentation corresponds to the hierarchy of user interface objects described by VDL. An abstraction is an object (in the object-oriented sense) that represents the data that should be presented to the user. The controls are objects responsible for maintaining the mapping between presentation and abstraction.

3.2. Support for CAOS in VDL

In order to support CAOS the hierarchy described by VDL must be annotated with control objects. Knowledge about the abstractions is also used by VDL to improve expressiveness.

Structure. Using CAOS means that it is known what abstraction is going to be presented by a given layout. For example, a layout meant to present an employee is only meaningful when used for exactly that purpose. Trying to use it for presenting a purchase order will result in chaos. Therefore it makes sense to structure the layouts according to abstraction.

This structure is achieved by appending the name of the abstraction to be presented to the name of the VDL style.

When using (inheriting from) such a layout the abstraction part of the name is automatically derived from the context established by the controllers annotating the presentation.

```
Big/Employee(bg):- … .
Big/Purchase(bg):- … .
```

Here a layout for an employee is defined and another one for a purchase order. Both are named *Big*. Both layouts are referenced using the name *Big*:

```
  …
  cld(emp, ctl(Object, get_employee),
    Big),
  cld(pur, ctl(Object, get_purchase),
    Big),
  …
```

The presentation actually used depends on the context established by the controllers. Since *get_employee* returns an abstraction of class *Employee* and *get_purchase* returns an abstraction of class *Purchase*, the VDL compiler knows where to find the correct presentation.

Controls. Controls are specified in VDL in a manner similar to children.

```
Big/Employee:-
  cld(name, ctl(Object, get_name),
    Small),
  … .
```

A layout for an employee is defined here, named *Big*. It has a child named *name* with an *Object* controller. The controller has a navigation specification of *get_name*. The child inherits from *Small*.

There can be more than one controller, but all controllers must be specified before any other definition items.

VDL has access to information about the abstraction structure, and knows what abstraction a method returns. The controller specifications are used to calculate what abstraction every style is presenting. This information is then used when looking up layouts with names containing the abstraction name.

4. Wcl in VDL

In the prototype implementation there is some specific support for Wcl and Motif.

4.1. Inherit from widgets

If a form inherited from is not found, this is interpreted as a Wcl declaration of what Motif widget class should be used to create the component. This is accomplished by setting the value of the *wcCreate* attribute.

```
label(txt):-
  XmLabel, labelString:val(txt).
```

The layout *label* is defined to inherit from *XmLabel*. Since no such layout has been defined, this has the same effect as setting the *wcCreate* attribute to *XmLabel*. This technique makes it possible to create a definition for *XmLabel* later to override the builtin defaults of Motif.

4.2. Popups

In Motif there is a second set of children to a widget called popups. VDL supports the creation of popups in a manner similar to ordinary children.

```
MenuBar:-
  pup(fileMenu, Pulldown(…), …).
```

The layout *MenuBar* has a popup called *fileMenu* which inherits from *Pulldown*.

4.3. Siblings

In Motif, convenience (or confusion) functions are often used. With Wcl the names of these functions can be used as values of the wcCreate attribute, just like the names of widget classes. The problem with these functions is that they often create more than one widget. It is even so horrible that the top-level widget of the widget tree they create is not given the name it was asked to use.

```
foo:-
  cld(bar, XmCreateScrolledList,
    items:”First, Second, Third”,
    itemCount: 3).
```

Here one might expect the creation of a child called *bar* of widget class *XmScrolledList*, that gets the attributes *items* and *itemCount* set. What happens is instead that a child named *barSW* of type *XmScrolledWindow* is created. This widget in turn has a child called *bar* of type *XmList*. The attributes defined therefore end up at the wrong level.

To solve this problem the concept of sibling is introduced. If a layout defines a sibling, it will be handled like a child, except that its attributes will be output as if it was a child of the parent instead.

```
foo:-
  cld(bar, XmCreateScrolledList,
    sib(barSW,
      cld(bar,
        items:”First, Second, Third”,
        itemCount:3))).
```

This is how sibling is used to solve the problem. The behaviour of *XmCreateScrolledList* is modelled, and there is even a way of setting attributes on *barSW*. Note that there is no widget class specification in either the `sib(barSW…)` or the `cld(bar…)`.

That is already taken care of by *XmCreate-ScrolledList*. There is no point in defining any attributes on the child that contains `inh(XmCreateScrolledList)`.

5. References

[1] Mats Johnson, CAOS, an extended object oriented model for dialogue design.

[2] Joelle Coutaz, PAC, an Object-Oriented Model for Dialog Design, in Proceedings of IFIP INTERACT'87: pp. 431-436, 1987.

[3] The Perl language http://www.perl.com/perl/

[4] Adrian Nye, Volume 2: Xlib Reference Manual, 3rd Edition June 1992, 1138 pages, ISBN: 1-56592-006-6, O'Reilly & Associates, Inc.

[5] David Flanagan, Volume 6C: Motif Tools, Streamlined GUI Design and Programming with the Xmt Library, 1st Edition August 1994, 1024 pages, ISBN: 1-56592-044-9, O'Reilly & Associates, Inc.

[6] TeleUSE User Manual.

[7] Adrian Nye, David Smyth, Wcl 2.0: the Widget creation Library, The X Resource issue 2, O'Reilly & Associates, Inc., Spring 1992.