Take pity on the user ! the RED customization package

Vincent Prunet

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Take pity on the user!
The RED customization package

Vincent Prunet

N° 2103

Novembre 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

1993

ISSN 0249-6399

Take pity on the user! The RED customization package

Vincent Prunet *

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Croap

Rapport de recherche n ° 2103 — Novembre 1993 — 31 pages

Abstract: X applications will not enter the market of PC users unless they provide a uniform and convenient customization mechanism. The *RED* (*Resource Editor Description*) language lets the application designer specify abstract objects that convey a conceptual model of the application. Values for abstract resources of this model may be specified either from *RED* text files or interactively using a generic *RED* resource editor. The editor's user interface is generated from the abstract model. Thus, conventional users easily customize applications.

Key-words: Window System, Resource editing, User Interface, RED, X11, Centaur.

(Résumé : tsvp)

*SEMA Group c/o INRIA Sophia Antipolis, Vincent.Prunet@sophia.inria.fr

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Ayons pitié de l'usager!
Paramétrisation des ressources avec RED

Résumé : Les applications X ne pénètreront le marché bureautique qu'à la condition de proposer un mécanisme de paramétrisation des ressources uniforme et convivial. Le langage *RED* permet au concepteur d'une application de spécifier des objets abstraits à partir du modèle conceptuel de l'application. Les valeurs des ressources associées à ces objets peuvent être spécifiées dans des fichiers ou interactivement à l'aide d'un éditeur de ressources générique dont l'interface graphique est générée en accord avec le modèle abstrait. L'édition des ressources est ainsi mise à la portée de l'usager non spécialiste.

Mots-clé : Système de fenêtrage graphique, Edition de ressources, Interface utilisateur, RED, X11, Centaur.

1 Introduction

X [1] has a powerful resource policy, which makes *Intrinsics* [2] based applications that exploit the X resource manager extensively highly customizable. Unfortunately, very few tools provide a high level interface to the resource mechanism, and the customization of an Xt application is still reserved to experienced X users.

Application writers sometimes ease suffering by providing an embedded resource editor, specifically tailored for the application. The advantage of this approach is that the concrete object hierarchy is hidden behind a conceptual model better suited to users. But we would rather not have to write a dedicated resource editor for every application. Moreover, the list of customizable resources cannot be modified without altering the application code. Another disadvantage is that distributed applications cannot be customized consistently. Lastly, embedded resource editors may exhibit different user interface models, and be inconsistent with each other.

Editres [3] is probably the first generic resource editor to be offered to Xt users. With *Editres*, one can dynamically edit the resources of a running application. When any visible object in the running application is selected with the mouse, *Editres* automatically computes the path leading to this object in the application widget tree. This path may be used to build an X resource specification. *Editres* allows interactive resource experimentation on a running application, which dramatically decreases editing time. With *Editres*, competent Xt users, and not just application developers, can customize an application. But even with *Editres*, normal users are excluded from this practice since the application's conceptual model remains obscure.

Custom [4], the IBM resource editor, targets naive users. Natural language text replaces resource patterns. A friendly user interface makes color or font selection very convenient. *Custom* abstracts somewhat from low level resource specifications by pretty printing resources understandably. Users may provide their own resource presentation declarations to accommodate their vision of the application. *Custom* is very handy, but fails to abstract sufficiently from the X resource pattern language.

The *RED* language (for *Resource Editor Description*), presented in this paper, lets the application designer specify abstract objects that convey a conceptual model of the application. Values for abstract resources of this model may be specified either from *RED* text files or interactively using a

generic *RED* resource editor. The dynamic configuration of the editor resembles that of *Custom*, but the *RED* specification file contains mostly abstract information intended to forge the user's perception of the application. Setting a resource on a single abstract object may result in modifying several concrete application components, but this is hidden from the user.

A *RED* compiler and a prototype of a *RED* resource editor have been implemented. The resources of any *X Resource Manager* based application can be specified using this editor.

2 A small example

The example we present is based on the *xcalc* [5] calculator. This *Athena toolkit* application emulates a pocket calculator, with two possible modes, *ti* or *hp*, corresponding to a TI-30 or an HP-10C calculator.

2.1 Xcalc architecture

The widget tree[1] of the application depends on the emulation mode. In *ti* mode, it is:

```
XCalc xcalc
        Form ti
                Form bevel
                        Form screen
                                Label M
                                Toggle LCD
                                Label INV
                                Label DEG
                                Label RAD
                                Label GRAD
                                Label P
                Command button1
                Command button2
                Command button3
                ...
```

[1] The widget tree was collected using *Editres*.

Command button38
Command button39
Command button40

where each line contains the widget class followed by the widget name.

The top-most *Form* widget, *ti*, contains the lcd display (*bevel* and sub-widgets) and as many *Command* widgets as there are keys on the calculator.

In *hp* mode, the top-most *Form* widget is named *hp* instead of *ti*. There are only 39 buttons in *hp* mode.

In both modes, keys are named according to position, not function. So digit *5* is represented by the widget *button27* in *ti* mode or *button17* in *hp* mode.

2.2 High level customizations

Customizations often have a semantic foundation: objects are customized according to function, not position. However, the associated resource values may also depend on object position (especially resources such as colors or fonts), to ensure the best contrast with sibling objects and the overall consistency of the application.

Here is an example of a semantic rather than position-based resource specification:

- a *Turquoise* background color for the numeric keypad,
- and a *DarkSlateGrey2* background for all other keys.

The following X resource implementation of this specification takes into account the *xcalc* application hierarchy and the semantics of each button (for both emulation modes):

```
XCalc.Form.Command.background: DarkSlateGrey2
XCalc.ti.button22.background : Turquoise
XCalc.ti.button23.background : Turquoise
XCalc.ti.button24.background : Turquoise
XCalc.ti.button27.background : Turquoise
XCalc.ti.button28.background : Turquoise
```

```
XCalc.ti.button29.background : Turquoise
XCalc.ti.button32.background : Turquoise
XCalc.ti.button33.background : Turquoise
XCalc.ti.button34.background : Turquoise
XCalc.ti.button37.background : Turquoise
XCalc.hp.button7.background  : Turquoise
XCalc.hp.button8.background  : Turquoise
XCalc.hp.button9.background  : Turquoise
XCalc.hp.button17.background : Turquoise
XCalc.hp.button18.background : Turquoise
XCalc.hp.button19.background : Turquoise
XCalc.hp.button27.background : Turquoise
XCalc.hp.button28.background : Turquoise
XCalc.hp.button29.background : Turquoise
XCalc.hp.button36.background : Turquoise
```

This specification deserves a few comments:

- It is barely readable. Actually, one cannot guess from this specification what keys are affected.
- It is long. Compare the size of this resource code with the length of the plain English specification.
- It cannot be maintained. The semantic information initially specified —that all numeric keypad buttons share the same background color— was lost during the translation into X resource specifications.
- The user must have a sound knowledge of the application hierarchy. *Editres* partially relieves this burden.
- Identifying and writing down all of the above resources was a lengthy, error-prone process, to be repeated for both emulation modes.
- Finally, we remark that generally, conventional users do not fully understand the X resource pattern matching algorithm and consequently, do not manipulate the precedence mechanism correctly. The above specification probably could not have been written by the average user.

A more concise but equivalent specification would be:

```
numeric_keypad.background = Turquoise ;
keypad.background = DarkSlateGrey2 ;
```

which is how we customize the application in *RED*.

Miracle? Not exactly. The long list of widgets that implement the numeric keypad still has to be specified. But instead of being specified by the user, this list is built, once and for all, by the application designer or an experienced user. Furthermore, conventional users can customize the application without ever knowing which toolkit implements it.

Appendix A lists the complete *RED* specification for *xcalc*. Section 3 provides an introduction to the *RED* specification language. Section 4 presents *RED* based tools intended for the end user.

3 The *RED* specification language

A *RED* specification is logically divided into several sections, and may be physically split into several files. All of these sections *but one* are to be written by the application designer or by an experienced X user (a system administrator, for example).

The section written by the user contains his or her own preferences. This section may be edited and modified using any text editor, but user preferences are usually generated by using a *RED* resource editor.

Let us go deeper into the *RED* language. There are four kinds of sections in a *RED* specification:

1. the abstract object model for the application. An application is represented as a set of *RED* objects. Each object belongs to a *RED* class. Each class has a list of named resources (properties). An object may have values assigned for each resource of the corresponding class.

   Subclassing is encouraged: a class may inherit resources declared by several other classes.

   Resources, classes, and objects can be annotated with any kind of information (including documentation messages and semantic links between objects).

2. the specification of default resource values for the *RED* objects defined in the previous section.

3. user defined customizations for the same *RED* objects.

4. the mapping of *RED* objects and resource values to those of the underlying window system toolkit.

Sections 1, 2, and 4 are generally specified by the application designer. Section 3 is generated under the user's control.

Sections 1, 2, and 3 do not depend on the customization mechanism used by the actual application. It should be noticed that every piece of information used or specified by the user is independent of particular the window system toolkit.

In the following sections, we describe the four section types in detail.

3.1 The abstract object section

The abstract object model consists of resource declarations, class declarations, and object instantiations. Objects are typed and their resources are constrained to be those listed in the corresponding class declaration.

3.1.1 Resources

RED resources must be declared. A resource declaration consists of a resource type name, a representation, and optional annotations. The resource type name is later used within class declarations. Various operations such as resource value range checking or specialized resource editing (using color editors or font browsers) depend on the resource type.

Primitive resource types are *Color*, *Font*, *FileName*, *String*, *Boolean*, *Integer* (and subranges), *Cardinal*, and *Pixmap*.

Examples:

```
resource Geometry = String ;

resource Digit = [0..9] ;
```

Enumerations, also called *SelectOne* or *SelectMany* depending on the number of selected items, allow for discrete range values. All possible values must be listed in a symbolic form that is independent of the format expected by the target resource manager.

Examples:

```
resource SimpleGeometry = {UpperLeft, UpperRight,
                           BottomLeft, BottomRight} ;

resource Caps = {SmallLetters, CapitalLetters} ;

resource Alphabet = set of {A, B, C, D, E, F} ;
```

Annotations are optional (and appear in parentheses):

```
resource EmulationMode = {HP, Ti}
    (doc = "Emulation mode:",
           " HP emulates an HP-10C",
           " Ti emulates a TI-30") ;
```

3.1.2 Classes

A class declaration associates a class name, a class structure, and annotations. A class structure consists of a possibly empty list of superclass names followed by a list of resources (with their names and types) and renamings.

Examples:

```
class Object = {
   foreground, background : Color ;
   backgroundPixmap : Pixmap ;
   font : Font
   }
   (doc = "Common object resources") ;

class TopLevel = {
   simpleGeometry : SimpleGeometry ;
   geometry : Geometry ;
   iconicOnStart : Boolean
```

```
   } ;

class Dictionary = Object, TopLevel {
   alphabet : Alphabet ;
   caps : Caps
   } ;
```

The first example includes an annotation specification. The second features user-defined resource types. The third shows inheritance from superclasses.

Renaming

Name conflicts between resources may be resolved by renaming or hiding some inherited resources. Other conflicts may be solved automatically by applying the precedence rule and then the unification rule.

- Precedence rule.

  A resource is either local or inherited. Local resources have a higher precedence than inherited resources. Inherited resources that have the same name as a local resource are not visible. Renamed resources have the same precedence as local resources. Consequently, a renamed resource cannot have the same name as a local resource, and a renamed resource takes precedence over inherited resources with the same name.

- Unification rule.

  Identically named resources inherited from a common ancestor class and resource (via different paths) are considered the same resource.

Renaming an inherited resource:

```
class Colored : Object {
   Object.background -> global_background ;
   background : Color
   } ;
```

Hiding an inherited resource:

```
class SubObject : Colored {
   Colored.background -> ;
   } ;
```

3.1.3 Instances

An object declaration associates an object name, a class structure, and annotations. Generally the class structure is just a class name, as in:

```
dictionary : Dictionary ;
```

However, in a few cases, it is not worth declaring a new named class for a single object instance. In such cases, the class structure may contain several superclass names and resource descriptions. The next example introduces the object *calculator*. As no other object shares the same structure, this object is declared as the unique instance of a anonymous class:

```
calculator : Object, TopLevel {
   emulationMode : EmulationMode
   }
   (doc = "Main window") ;
```

3.2 The default and user value sections

The person responsible for *RED* specifications generally provides a set of default values for the attendant abstract resources. The user may also provide a section that further tailors the application.

A value specification is a three part statement including an object name, a resource name, and a value or a reference to another resource. Both object and resource are abstract. Values are abstract data. However, depending on the resource representation type, there may be either an implicit conversion from the *RED* representation to the toolkit representation or an explicit conversion according to parameters specified in section 3.3.

Abstract values may be identifiers, character strings, booleans, or numbers. Values can also be represented by lists of identifiers in the case of *SelectMany* resource representations.

A reference consists in a resource name prepended by an object name. Replacing a value by a reference to another resource ensures that both resources hold the same value.

Examples:

```
calculator {
        simpleGeometry = BottomRight,
        emulationMode  = HP };

calculator.background = Aquamarine3 ;

dictionary.alphabet = {A, B} ;
```

Abstract values may be replaced by references to resources. The next example shows how resources of an object (here *colorPalette*) may be used as constant values in the specification:

```
colorPalette : { foreground, background,
                 highlight, shaded : Color } ;

calculator.foreground = palette.foreground ;
```

3.3 The mapping section

Abstract resources and objects are mapped to string values and resource patterns, according to the X resource manager syntax.

3.3.1 Resource values

Resource values are generally specified as character strings or identifiers. When the resource has an unbounded range of possible values (colors, fonts, ...), these strings and identifiers are also used as X values (i.e., implicit casting).

Discrete resources must have abstract values that are represented by identifiers which may be expanded into different character string values. When no translation information is provided for a given abstract value, the identifier is cast to X's character string representation of it.

Examples:

The screen location of an Xt top level widget can be specified abstractly when the accompanying mapping section includes the following specifications:

```
@ resource SimpleGeometry {
   UpperLeft  = "+0+0", UpperRight  = "-0+0",
   BottomLeft = "+0-0", BottomRight = "-0-0" };
```

SelectMany resources can be parameterized to pretty print their values with prefix, postfix, and separator character strings. The default prefix and postfix strings are null; the default separator is a space character.

Example:

```
@ Alphabet {
   prefix    = "(",
   postfix   = ")",
   separator = ", " };
```

3.3.2 Object mapping

Depending on the abstraction level of the *RED* specification, designers may choose one of three different syntactic constructs to express the mapping from the abstract representation to application patterns.

- Object coupling: An abstract object is bound to a list of X object patterns (an object pattern is any valid X resource pattern without its last component). During the generation phase, the resource name is appended to each of these patterns to produce a complete X pattern.

  Example:

  ```
  @ calculator -> '?*' ;
  ```

  A slightly different example is:

  ```
  @ calculator -> '?' & '?.ti*' & '?.hp*' ;
  ```

  which exploits the multiple pattern capacity of the mapping language. This important feature of the language allows for binding a *RED* object to any actual application objects, whether or not they can be designated by a single pattern.

- Resource coupling: An object resource is bound to a list of X patterns (including the last component, an X resource name or class). This allows for different *RED* and X resource names.

  Example:

  ```
  @ calculator.emulationMode -> '?.rpn' ;
  ```

- Value coupling: Both X patterns and resource values are hidden. This mechanism is available only for discrete resource types. A list of X resource specifications is associated to each abstract value.

  Example:

  The *dictionary* application consists of a list of *Label* widgets that represent letters of the alphabet. The capitalization of labels may be performed consistently, without modifying the application code.

  ```
  @ dictionary.caps.SmallLetters ->
    '?*part_A.Label: a' & '?*part_B.Label: b' & ... ;
  @ dictionary.caps.CapitalLetters ->
    '?*part_A.Label: A' & '?*part_B.Label: B' & ... ;
  ```

When a value specification is compiled into an X resource specification, the three coupling mechanisms are tried from the most specific one (value coupling) to the most general one (object coupling).

3.4 Planned extensions

We have several worthwhile extensions in mind, but these have not yet been completely specified.

3.4.1 Cooperating applications

One may wish that resources of different applications be edited together, mapping abstract objects to widgets of different applications. For example *Custom* lets the user specify resources for both the application and the window manager. We think that *RED* should do more in that direction. *RED*

should be able to encapsulate a complex multi-application environment within a single (modular) specification.

For example, our main project is the development of a programming environment generator [6] which is composed of many cooperating programs including a user interface component, a processor for abstract syntax trees (including a pretty printer), and a graph formatting and displaying engine. A single *RED* specification would hide the complexity of the system and increase the consistency of the environment, allowing, for example, the same color resource to be used to display an identifier in a pretty printed program and the corresponding node in the associated dependency graph.

We plan to extend the mapping syntax to allow for the specification of distributed environments within a single *RED* specification.

3.4.2 Composite resource types

Among the different resources that compose a *RED* class, one may desire that certain intimately bound resources be associated as a single, more abstract resource type. For example, the foreground and background colors of an object, although different resources, are often specified jointly for aesthetic reasons. These resources could be associated into typed records. Thus, specialized resource editors could be specified for composite resource types, enabling one, for example, to edit jointly a set of matching colors.

3.4.3 Functions on patterns

We would like to be able to assign patterns to *RED* variables, and offer pattern composition functions. This would greatly ease the process of writing *RED* programs.

3.4.4 Support for different pattern levels

Patterns used as examples in this paper always start with a `?` character. Actual X patterns may start with the application class, the application name (that may be changed by the user), a `?`, or a `*` character. This feature, although specific to the X resource manager, is very useful as it lets the user decide whether a customization is applicable to all instances of an application or only some of them. The *RED* mapping syntax should provide a way to access this type of parameterization.

4 *RED* tools

A *RED* specification is not only a high level specification of application customizations, but also a specification of a dedicated resource editing tool. Consequently, two different tools make use of *RED* specifications:

- the *RED* batch compiler, that processes a specification and generates the corresponding X resource specification;
- the *RED* user interface and specification generator, which is able to build a graphical user interface (specialized forms to edit *RED* resource values) from the *RED* program, and to generate a list of *RED* value specifications from the interaction of the user with the dedicated graphical interface. These value specifications, coupled with the original *RED* program, are processed by the *RED* compiler and produce an X resource specification file.

4.1 Batch compilation

A *RED* compiler is a tool that parses a *RED* program, type checks the specification , and then generates an X resource file from specified value settings. The type checker verifies that all used resources, classes, and objects are declared, computes the list of resources available for each object, and checks that discrete resource values are valid. The compiler can process either the default values specified in the *RED* program or a separate file that contains user customizations.

Examples:

A possible but intricate binding for the *calculator* object is:

```
@ calculator -> '?' & '?.ti*' & '?.hp*' ;
```

Given this binding,

```
calculator.background = Aquamarine3 ;
```

is compiled into:

```
?.background : Aquamarine3
?.ti*background : Aquamarine3
?.hp*background : Aquamarine3
```