Testing Widget Geometry Management

Daniel Dardailler

Abstract

The complexity of the Xt geometry management and the variety of OSF/Motif layout policies combine to make the understanding of the layout activities in any Xt/Motif based application a very difficult task. This paper presents a set of tools intended to help programmers follow the complex flow of geometry requests and check the compliance to the Xt specifications of a program or a widget set library. A first technique using a specialized widget is described, and then a library-based system is presented. The advantages and drawbacks of both approaches are then discussed in a separate section.

Introduction

Debugging large Motif applications is not a simple task and the first thing OSF usually asks from the programmers reporting bugs is to extract from their large application a minimal program that shows the same anomaly. While this technique usually works fine for most defect issues, layout problems still remain very complex to track down, the problem residing most of the time in the toolkit itself, in the form of untested combinations of nested Composite and Primitive widgets.

Even the simpler of those geometry-related problems almost always involve stepping through the widget code, and if a C interpreter (like Saber) is not available, the developer usually needs to re-build the library `-g` before actually starting to debug. In addition, with or without re-compilation needed, once the test is ready to run in the symbolic environment, the engineer may have forgotten the "magic" of the Xt

Daniel Dardailler (daniel@osf.org) joined the Open Software Foundation in 1990 as a Senior Software Engineer in the User Environment Components group. His responsibility in the OSF/Motif project includes the geometry management of most widgets. He holds a Ph.D. in Computer Science from the University of Nice/Sophia Antipolis.

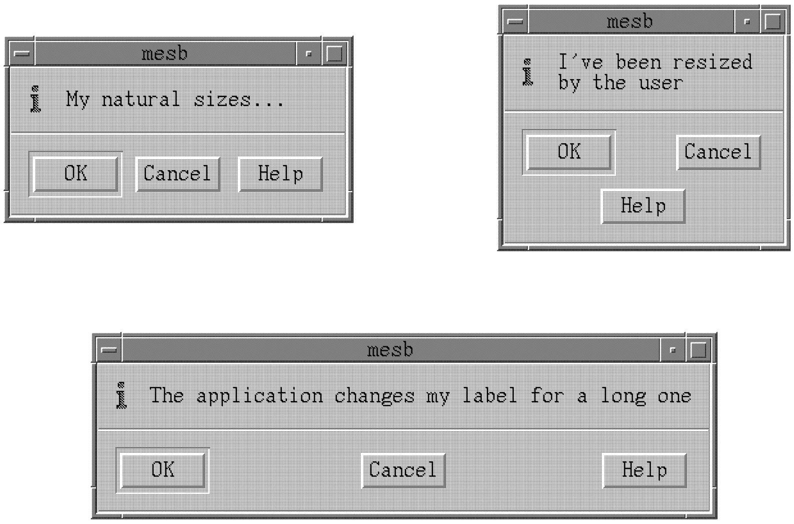

Figure 1: Dynamic re-layout in Motif with Xt.

geometry management, and he may spend valuable time deciding where to put his break-points. As the resources in disk space (for the debug libraries) and engineering become scarce, this type of overhead becomes unacceptable.

The idea of this work is to provide a very flexible framework that allows the application's developer and/or the toolkit's developer to track most geometry management problems without even having to recompile the original code.

By just relinking with a special library, the developer is now able to point to a particular area in a widget tree instance (using pseudo-resource descriptions in a file or an interactive tool like `editres`, the X resource editor), and he will get a report of the geometry flow for this specific zone of the application.

The next chapter briefly presents some aspects of the Xt geometry management. The knowledge of the layered X architecture with the Intrinsics providing the widget abstraction is required.

Xt geometry management

Why is the Xt geometry management so difficult to understand?
Mainly because it is powerful.

If you look at a MacIntosh or a Windows 3.0 screen, you'll see that there is almost no dynamic layout. Either the applications include a scrollable viewport, and everything is clipped when the top level window gets smaller, or there is no scrolled window and user's resize handles aren't present. There is in both toolkits a way for the application to be called back when the top level window is resized, but since no generic mechanism is provided to handle the re-layout, few applications take advantage of these resize notifications.

In Xt, on the other hand, the toolkit provides all the mechanisms for doing dynamic re-layout. The key point is that the preferred geometry of a composite is computed from its children's preferred geometry (which themselves may be composites).

Figure 1 illustrates the different kind of "traversal" that Xt geometry management can perform on a given application widget instance tree.

The first message box layout (top-left, "natural sizes") results from a initial bottom-to-top flow that is implemented as a postorder top-down traversal. The push buttons and the message label are first asked their preferred sizes, the message box then arranges all its children and computes its own geometry, which is eventually used to size the top level shell window.

In the second layout (top-right, "resized by the user"), a top-to-bottom chain of reconfiguration orders is shown. When the top-level shell window is resized, it resizes its unique message box child. In turn, the message box accommodates its own layout to its new boundaries: the preferred geometry of the label is asked and given the room left in the manager, an alternate button-placement algorithm is used.

The last window is an example of bottom-to-top request-flow. When the application changes the message label, Xt manages to have the parent (the message box itself) informed. As a result of the grow request made by the label, the message box computes a new push buttons layout and asks the top level shell window to become larger. The shell widget itself needs to communicate with the window manager (as it is a different process in the X architecture) and if the size changes are accepted as is, the re-configuration eventually occurs.

Xt, like the X protocol, provides generic mechanisms, not policies. The widgets have to implement their own layout algorithms within the Xt framework. The generic way for a widget to resize another widget (usually a child) is thru a call to XtResizeWidget, but most widgets have private layout policy resources (orientation, packing, visibility, etc) that control *their own geometry* inside the new boundaries. The level of complexity and the bugs found in the manipulation of these resources are orthogonal to the Xt geometry management mechanisms, but one should understand both dimensions in order to maintain the widget set code.

The Application Programmer Interface provides three basic mechanisms:

- a way for a parent widget to reconfigure the size and position of a child widget: XtConfigureWidget (with its variants XtResizeWidget and XtMoveWidget). It's a top-to-bottom order.
- a way for a child widget to request a change in geometry: XtMakeGeometryRequest (and its variant XtMakeResizeRequest). It's a bottom-to-top flow.
- a way for a parent widget to ask its preferred geometry to a child: XtQueryGeometry.

Composite widgets need in turn to implement a GeometryManager method that receives and possibly makes calls to XtMakeGeometryRequest, and a ChangeManaged method that handles the changes in the list of children. The actual space allocation is implemented in the Resize method.

Not to help the overall understanding, the structures used in the API (XtWidgetGeometry, XtGeometryResult) are shared by the request and the query geometry functions but with different semantics...

If I add that all this stuff happens simultaneously and often requires inter-process communications (client/X server/window manager), you'd be convinced that a new tool was needed!

The GeoTattler widget

This chapter describes a new widget, the GeoTattler, whose purpose, once inserted in a widget hierarchy, is to check the compliance to the Xt geometry management specifications of its child and its parent. It does that by reporting in a human-readable form the flow of requests and the inconsistencies occurring during the up and down process of the Xt layout mechanism.

A special section is dedicated to a complementary library, libGeo, which includes the GeoTattler code and an XtCreateWidget wrapper that uses Xt subresources to determine where to insert a GeoTattler widget in the application widget hierarchy at startup time.

Description

The basic idea is to create a new composite widget, the GeoTattler, an instance of which is inserted between a parent and a child widget, for the purpose of tracing and reporting their geometry management negotiations, now routed through the GeoTattler class methods.

This idea of inserting an active component into the flow of a given process for debugging purpose has already been proven useful, at the X protocol level, by the success of the `xscope` or the `xmon` program.

GeoTattler is going to "fake" the child from the parent's point of view, and fake the parent from the child's point of view. It can easily detect, for instance, actions such as the parent resizing the child, since GeoTattler will be resized first and will have to resize the child itself; at this time, it can report the geometry request coming from the parent before passing it to the child.

The GeoTattler has of course all the methods needed to handle geometry management properly: Initialize, Resize, GeometryManager, ChangeManaged, and QueryGeometry.

In addition, since the current design forces the GeoTattler to have only one child, it has an InsertChild method which controls that number. We believe it's better to use a higher level set of facilities to test the behavior of several children in a same parent (like the use of subresources described in the next section).

Look at the Xt specification (R4/R5), chapter 6: *Geometry Management*, and you'll see that it is full of *shoulds* and *musts*, and that there is in fact a very formal protocol hidden behind these 12 pages of esoteric English text :-) Given this complexity, it would also be interesting, to not only trace, but also to test whether the parent and the child react properly to geometry management actions.

The technique of testing used in the GeoTattler is very simple: it has a certain number of flags stored in the instance widget, and the geometry related class methods set and check them at the right time.

Let's start by the current list of predicates the GeoTattler is able to verify and then we will describe the implementation of the first one in detail.

- The parent has a compliant XtGeometryDone or XtGeometryYes policy (it calls the resize proc of the child on acceptance of a geometry request or not).
- The child is not requesting its parent from its resize proc (might result in infinite loop).
- The parent is not considering an unmanaged child.
- The child deals properly with the XtGeometryAlmost proposal.
- The parent always accepts its own last XtGeometryAlmost proposal .
- An XtGeometryNo or an XtGeometryAlmost reply from the parent didn't change the current geometry.
- The XtCWQueryOnly bit set doesn't change the current geometry either.
- A widget is not making a geometry request from its SetValues method (thus allowing a subclass to change the behavior).
- A child QueryGeometry is conformant to the specification for XtGeometryNo, XtGeometryYes and XtGeometryAlmost.

Because of the Xt specifications being frozen too early in the life time of the Intrinsics, the XtMakeGeometryRequest function is not able to report if the parent has effectively resized the requestor child. The managers' GeometryManager method can return either XtGeometryDone or XtGeometryYes, but the Intrinsics will change any Done to Yes before returning from XtMakeGeometryRequest.

From the child's point of view, after a request has been accepted, this means that either it trusts its widget set policy (if there is one) and always, or never, call its resize proc itself, or it has to track the calls to its own resize method to see if it needs to re-layout (because its parent didn't do it yet).

That's exactly what the GeoTattler simulates. In its resize method, it sets an instance boolean True and its Geometry Manager sets it False before making (passing really) a request to its parent. If the boolean value is True after the return of XtMakeGeometryRequest, the parent is a XtGeometryDone, otherwise, its a XtGeometryYes. Note that this doesn't mean that the parent returned XtGeometryDone or XtGeometryYes to Xt; it can be wrong in its interpretation of its own reply.

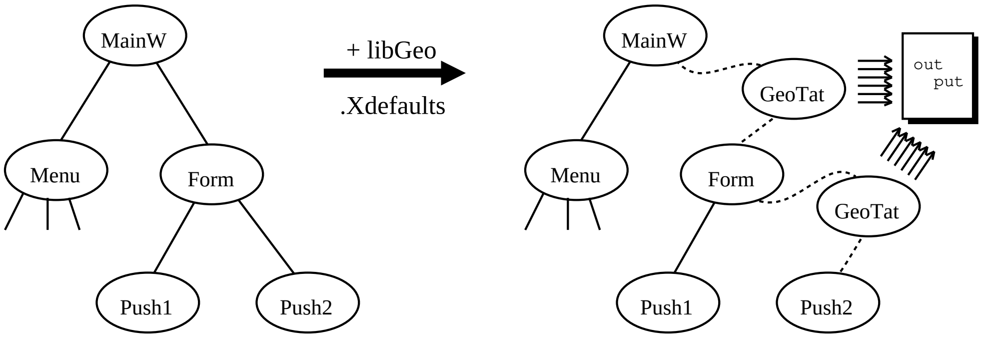

Figure 2: Widget tree modification.

The Motif widget set has moved to a coherent XtGeometryYes policy for all its manager in the 1.2 release, so this is particularly important to check before incorporating any new manager in the toolkit.

The GeoTattler is a subclass of the Motif XmManager, and it is really specific to the Motif toolkit due to its constraint resources set (see the Discussions chapter for details). As a Motif manager subclass, it also inherits other features of XmManager including Gadgets handling and Motif keyboard traversal management. Adapting the GeoTattler for a different Toolkit, though, would not be difficult.

How to use the GeoTattler: libGeo

The GeoTattler widget is fairly transparent in the sense that it never mentions its own name in the output trace. It's always in the form: "`parent` is doing something to `child`", even if the truth is that `parent` is doing something to the GeoTattler, which in turn is doing something to `child`.

The goal of the libGeo library is to extent this transparency to the application itself, so that the GeoTattler be usable without recompiling any program.

Let's describe how it works from the user point of view (the user being a developer who wants to check the geometry behavior of an application) and then we'll describe the implementation.

Let's consider a very simple hierarchy of widgets: a main window containing a menubar and a form with 2 buttons in it (as illustrate in the left part of figure 2).

The initial production line for this test is:

```
cc -o sample sample.o -lXm -lXt -lX11
```

where sample.c is the application creating the widgets, Xm is the motif library, Xt the X toolkit Intrinsics and X11 the X library.

The first thing to do is to re-link the program with the Geometry tester library inserted before Xt:

```
cc -o sample sample.o -lXm -lGeo -lXt -lX11
```

By doing just that, the behavior shouldn't have changed.

Then, if these lines are added to the resource environment (i.e the .Xdefaults file):

```
*XmForm*geoTattler: ON
sample*Push1.geoTattler: OFF
```

the widget tree becomes as illustrated in the right part of figure 2 and the program produces this specific output:

(after an XtSetValues (..XtNwidth..) on Push2 by the application)

```
"Push2" is doing a geometry request:
       - width wants to move from 226 to 284
     to its parent "Form".
"Form" is resizing its child "Push1" to (200,32)
"Form" replies Yes to the request.
```

(after a resize of the toplevel shell by the user)

```
"sample" is resizing its child "Form" to (300,200)
"Form" is asking "Push1" for its preferred geometry
"Form" is asking "Push2" for its preferred geometry
"Form" is resizing its child "Push1" to ..
      etc, etc...
```

The XtNgeoTattler resource (really a pseudo-resource), with the Boolean values ON and OFF (default) is the only new external interface the user needs to learn. Since the implementation uses a standard function to retrieve the value, the matching rules (bindings, class or instance name, conversion) and the specification niceties of the X resource manager apply.

The semantic of this resource is that if a widget has the attribute XtNgeoTattler set, it is reparented to a GeoTattler widget.

The libGeo.a archive is made of only two modules: the GeoTattler widget itself, GeoTattler.o, and a module GeoCreate.o that simply replaces the XtCreateWidget function. Instead of just calling the internal _XtCreateWidget routine, it now calls XtGetSubResources with the class, name and parent of the widget being created and if a geoTattler resource has been set, a GeoTattler is created as a child of the given parent and the widget itself is created as the child of the GeoTattler.

Two additional hooks are provided for the realization and the destruction of the tree at creation time. If the parent is already realized, the GeoTattler needs also to be realized, since its child can be realized at any moment in the future and an X protocol error would be generated if the GeoTattler was not realized. A destroyCallback is also added to the child, so that it will destroy the GeoTattler if it is being killed before its original parent.

```
#define XtNgeoTattler "geoTattler"
#define XtCGeoTattler "GeoTattler"

typedef struct { Boolean geo_tattler ;} GeoDataRec ;
static XtResource geo_resources[] = {
    { XtNgeoTattler, XtCGeoTattler, XtRBoolean, sizeof(Boolean),
      XtOffsetOf(GeoDataRec, geo_tattler),
      XtRImmediate, (caddr_t) False }
} ;

Boolean _GeoIsTattled (Widget widget)
{
    GeoDataRec geo_data ;
    Boolean is_geotattled ;

        /* First check for a matching widget in the cache */
    if (_is_geo_cached (widget, &is_geotattled)) return is_geotattled ;

        /* no widget found in the cache, look in the database */
    XtGetSubresources(widget, (XtPointer)&geo_data,
                      XtName(widget), XtClassName(widget),
                      geo_resources, XtNumber(geo_resources), NULL, 0);

        /* now add this guy in the cache */
    _geo_cache(widget, geo_data.geo_tattler) ;

    return geo_data.geo_tattler;
}
```

Figure 3: _GeoIsTattled implementation.

The libXtGeo library

While developing the XtCreateWidget wrapper for the GeoTattler, it appeared that modifying the Xt library was an easy and very powerful way to see what really happens at the Intrinsics level (obvious, isn't it?).

Description

Following this idea, the Xt modules (R5 MIT implementation) dealing with geometry management were isolated:

- Geometry.c, of course, for the basic mechanisms.
- Manage.c, for the ChangeManaged calls.
- Intrinsic.c, for XtRealizedWidget which calls ChangeManaged.

```
void XtConfigureWidget(w, x, y, width, height, borderWidth)
    Widget w;
    Position x, y;
    Dimension width, height, borderWidth;
{
    XWindowChanges changes, old;
    Cardinal mask = 0;

    _GeoPrintTrace(w,"%s is being configured by its parent %s\n",
                   XtName(w), XtName(XtParent(w)));
    _GeoTabTrace();

    if ((old.x = w->core.x) != x) {
        _GeoPrintTrace(w,"x moves from %d to %d\n",w->core.x, x);
        changes.x = w->core.x = x;
        mask |= CWX;
    }
    ..... same tests for w->core.y, width, height and border_width ...

    if (mask != 0) {
        if (XtIsRealized(w)) {
            if (XtIsWidget(w)) {
                _GeoPrintTrace(w,"XConfigure %s's window.\n", XtName(w));
                XConfigureWindow(XtDisplay(w),XtWindow(w),mask,&changes);
            }
            else {
                _GeoPrintTrace(w,"ClearRectObj called on %s.\n", XtName(w));
                ClearRectObjAreas((RectObj)w, &old);
            }
        } else {
                _GeoPrintTrace(w,"%s not Realized.\n",XtName(w));
        }

        _GeoUnTabTrace();

        if ((mask & (CWWidth | CWHeight)) &&
              XtClass(w)->core_class.resize != (XtWidgetProc) NULL) {

            _GeoPrintTrace(w,"Resize proc is called.\n");


            (*(w->core.widget_class->core_class.resize),w);
        } else {
            _GeoPrintTrace(w,"Resize proc is not called.\n");
        }

    } else {
            _GeoPrintTrace(w,"No change in configuration.\n");
            _GeoUnTabTrace();
    }
} /* XtConfigureWidget */
```

Figure 4: Modified version of XtConfigureWidget.

- SetValues.c, which makes geometry requests at the end of the chain.
- Shell.c, for the Shell RootGeometryManager.

The new library, libXtGeo.a is made of a modified version of these five modules.

Note that the Create and VarCreate modules are not needed anymore, since the XtGetSubResources work is now done dynamically. A function _GeoIsTattled that takes a widget is provided for this purpose. It first looks in a cache to see if the triple [widget, widget_name, class_name] matches something, and if not, it calls XtGetSubResources and caches the result (implementation shown in figure 3) .

In addition, the libXtGeo library includes a module which provides 3 public functions:

- _GeoPrintTrace, which takes a widget and a printf specification. This function uses _GeoIsTattled(widget) to determine if the trace is wanted.
- _GeoTabTrace, which adds a tabbing to the indented output.
- _GeoUnTabTrace, which pops a tabbing.

The modifications made to the Xt modules themselves are fairly straightforward: before any interesting action, the libXtGeo functions are called (figure 4 shows the new XtConfigureWidget routine, from Geometry.c).

The output trace given by libXtGeo is much more complete than with the GeoTattler. Tracking the calls to the shell root geometry manager and made by the shell root geometry manager, for instance, is only feasible at this level.

In addition, and since the "black box" was opened, the Editres event_handler was added to Shell.c (so that the widget hierarchy be easily viewable) and a hack was added to XtSetValues in order to perform dynamic setting of the geoTattler subresource. The result is the following: providing that your application has been linked with libXtGeo, you can **interactively** turn ON and OFF geometry management report using the `editres` resource editor.

How to use libXtGeo

The libXtGeo library provides the programmer with the same interface (XtNgeoTattler:ON/OFF) than the GeoTattler package. Programs only need to be re-linked with -lXtGeo before -lXt and the resource environment to be modified as well.

Let's consider the report generated by a single push button inside a shell, when the activate callback of the button calls XtSetValues to change the label string to `GoodBye world` (as shown in figure 5).

With the following .Xdefaults file:

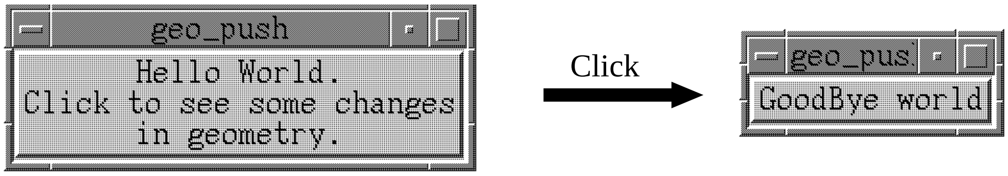

Figure 5: Bottom-to-top resize request.

```
 # original label string
geo_push*push.labelString:Hello World.\n\
                         Click to see some changes\n\
                         in geometry.
 # needed to allow bottom-top requests to succeed
geo_push*allowShellResize:True
 # we want exhaustive report for this demo
geo_push*geoTattler: ON
```

the output is (when the user clicks on the button and the setvalues on the label is performed):

```
XtSetValues on "push" sees some geometry changes.
    "push" is making a geometry request to its parent "geo_push".
         Asking for a change in width: from 235 to 127.
         Asking for a change in height: from 58 to 26.
         Go ask the geometry manager.
            "geo_push" is making a geometry request to its parent Root.
                  Asking for a change in width: from 235 to 127.
                  Asking for a change in height: from 58 to 26.
                  Go ask the RootGeometryManager.
                       Configuring the Shell X window :
                                width = 127
                                height = 26
                       ConfigureNotify succeed, return XtGeometryYes.
            Root returns XtGeometryYes.
            Reconfigure "push"'s window.
     "geo_push" returns XtGeometryYes.
     XtSetValues calls "push"'s resize proc.
XtSetValues calls XClearArea on "push".
```

For regular applications (hundreds of widgets), this kind of report becomes very dense and the geoTattler:OFF value is used to **focus** on very specific area of the widget tree.

Discussions

GeoTattler issues

Since the design principle of the GeoTattler widget is to insert additional widgets in a widget's tree, it also modifies the underlying hierarchy. A side effect is that it breaks the behavior of applications and toolkits that were relying on specific properties of this hierarchy.

Inter-dependence between widget classes is the first point we have to look at. In Motif especially, there are a lot of places where the XmIs*Class* macros are used, and introducing a GeoTattler somewhere in the instance tree is often an issue.

The RowColumn widget, for instance, has an XmNentryCallback resource which is used to revector the activateCallback of all its Buttons children. It does that by checking the class of the children before systematically adding its own callback. If one of the button is reparented to a GeoTattler, which is not a button and has no activateCallback anyway, the entryCallback won't work anymore for the button child of the GeoTattler. Another example is the BulletinBoard widget which checks if its parent is a DialogShell before reporting its dialogTitle attribute up one level.

Note that these widget class inter-dependencies wouldn't be present if we had a general *widget property mechanism* in the Intrinsics (eg. Traits).

Another problem is found in the resource file themselves, when they become very specific about the hierarchy. Things like:

```
myapp*XmFrame.XmPushButton.shadowThickness: 0
```

won't work anymore if a widget is systematically inserted in the hierarchy:

```
myapp*XmPushButton.geoTattler: ON
```

The new description is breaking the initial parent-child relationship between the frame and the push buttons (since all PushButtons are now reparented to GeoTattlers). But since the resource file can easily be updated when the geoTattler resource itself is added, this problem can be considered as minor.

A much more nasty issue appears with constraint resources.

Constraint resources are maintained by the parent for the child, and if the parent class changes, they are not maintained anymore.

Consider a PanedWindow for instance, it has a XmNpaneMax constraint resource which controls the maximum size the user can resize a particular pane (the pane child the resource is set for). If this pane child is reparented to a GeoTattler, the GeoTattler then becomes the child of the PanedWindow and the paneMax constraint originally set on the child is lost. For constraint resources that are related to geometry management (like Form attachments), it is a very serious problem, even in a test environment.

The only solution to this problem we have found so far is to make the GeoTattler itself a constraint widget (which it was already, as a subclass of the Motif XmManager) with a set of constraint resources equal to the union of all the constraint resources present in the widget set tested.

At creation time, when the GeoTattler ConstraintInitialize method is called, we check the current constraint resource values against their default; if they have changed, we report the change to the parent of the GeoTattler (using XtSetValues on the GeoTattler itself). The same code is used at ConstraintSetValues time to track the change between the current and the new constraint attributes.

This is not a generic solution but it works well for a given toolkit at a given time. It is also very localized code, easy to adapt for new Constraint widgets.

Toolkit approach issues

The main concern with the modified toolkit approach is the maintenance of the code, since the original Xt continues to evolve and we are not planning on having MIT adopt our code. But since the libXtGeo is only a subset of the entire Xt and we (the OSF/Motif team) are working in close relationship with the X consortium for Xt specifications, this is not a big problem as long as we maintain libXtGeo ourselves.

A clear advantage of libXtGeo over the GeoTattler is its independence of the widget set tested. libXtGeo is more than a simple DEBUG version of Xt (outline output, geoTattler sub-resource specification, specialization in Geometry Management) but it is still to be used as a replacement of a part of Xt, on top of which the widget sets are built.

Conclusion

I have used only libXtGeo in my tests for Motif and it has already proven to be **very** helpful. Anyone developing a new widget or debugging an application that has layout problems might be interested by this tool (freely available on the server `export.lcs.mit.edu` in the archive `contrib/libXtGeo.tar.Z`).

The GeoTattler work wasn't done in vain though. It gave us a better understanding of the consequences of modifying a genuine widget instance tree (and there are other examples of such a technique, like the self-moving widget).

Both the GeoTattler widget and the libXtGeo package give information about the current flow of requests in the widget set, but using these tools, you'd never really perform an *exhaustive* checking of the geometry managers. If the XtCWQueryOnly flag is never used by any child widget, for instance, there is no way to find out that composite parents in a widget set are not properly dealing with query only requests.

Another test bed needs to be built, with an enhanced version of the GeoTattler as the central component. The final program would resemble a small interface builder, allowing the user to create various hierarchies of widget classes, embedded with GeoTattlers. It would let the operator activate very specific geometry actions, thus really **validating** the geometry management of any component.

Bibliography

Robert W. Scheifler, James Gettys. *The X Window System. C library and Protocol Reference.* Second edition. Digital Press. ISBN 1-55558-050-5.

Paul Asente, Ralph Swick. *The X Window System Toolkit.* Digital Press. ISBN 1-55558-051-3.

Open Software Foundation. *OSF/Motif 1.1 Programmer's Reference.* Prentice Hall. ISBN 0-13-640681-5.

Dan Heller. *Motif Programming Manual.* Volume Six - O'Reilly & Associates, Inc. ISBN 0-937175-70-6

Kee Hinckley and Andrew Schulert *Geometry Management with Xt: Advice for Widget Authors.* Xhibition 91 Conference Proceedings, June 1991.

Hubert Rechsteiner. *The Self Moving Widget.* Xhibition 91 Conference Proceedings, June 1991.