Erfahrungen mit dem Interface-Builder

ixbuild

bei der Konstruktion von Motifoberflächen für Supercomputeranwendungen

IAO-Forum Informationstechnik
Stuttgart, 5. November 1992

Joachim Backes
Universität Kaiserslautern
Regionales Hochschulrechenzentrum

Paul-Ehrlich-Straße, Geb. 34
D-6750 Kaiserslautern

Inhaltsverzeichnis

1 Einführung

Die Akzeptanz eines Supercomputer- (und auch Mainframe-) Betriebssystems durch den Benutzer hängt nicht nur von den Funktionen ab, die ein solches System bietet. Diese sind mehr oder weniger interner Art.

In verstärktem Maße werden heute Benutzer mit lokaler Intelligenz (Workstations, PCs, X-Terminals) als Frontend konfrontiert. Dort laufen viele Anwendungen mit graphischer und objektorientierter Oberfläche. Ein Benutzer, der diese Art Schnittstellen schätzen gelernt hat, ist kaum noch bereit, für die Inanspruchnahme von Supercomputerleistungen dessen in vielen Fällen sowohl exotische als auch bedienerunfreundliche Kommandosprache zu erlernen. Ein indirekter Supercomputer-Zugang (bei einer Frontend-Backend-Konstruktion) mit unterschiedlichen Kommandoschnittstellen und Betriebssystemmeldungen wird diese Problematik zusätzlich verstärken. Nein, der Anwender möchte von seiner Workstation aus, wenn möglichst grafisch und mausgesteuert, über einprägsame und sicher führende Menüs mit dem Supercomputer arbeiten.

Kommandointerpreter wie ISPF bzw. PFD (MVS, VSP/I,...) oder SDF (BS2000) beispielsweise zeigen zwar Ansätze in die richtige Richtung. Relativ unintelligente Terminals jedoch wie IBM/3270 oder SNI/9750 machen diese Schnittstellen nicht gerade attraktiv.

Will man nun ein solches grafisches Interface entwickeln, so bietet sich aus Gründen der Portabilität als Fenstersystem naturgemäß das *X-Window-System* an (mit allen seinen Ebenen) mit einer Toolkit-Schnittstelle wie z.B. OSF/Motif® an, wobei natürlich ein gewisser Verlust an Performanz nicht zu verleugnen ist (proprietäre Window-Mechanismen können dagegen stets auf das sie verwendende Betriebssystem hin optimiert werden). Aufgewogen wird dieser Verlust durch einen Gewinn an Flexibilität: X-Window erlaubt die transparente Aufteilung einer Anwendung in die Funktion einerseits sowie die Darstellung andererseits (z.B. auf anderen Workstations, X-Terminals, PCs mit X-Software usw.; Client-Server-Architektur).

Einer variablen Netzeinbindung steht damit nichts im Wege.

Im folgenden soll über Verfahren und Probleme berichtet werden, mit denen der Entwickler eines solchen Interfaces konfrontiert ist.

2 Werkzeuge zur Schnittstellen-Generierung

Bekanntlich ist es nicht trivial, X-Window-fähige Programme zu schreiben. Gerade weil die X-Window-Bibliotheken X-Library (Xlib) , Xt-Library (Xtlib, als Toolkit-Basis) und z.B. OSF/Motif-Toolkit (Xmlib) sehr komplex aufgebaut sind, muss der Programmierer sämtliche X- und Toolkit-Aufrufe selbst durchkonstruieren. Insbesondere muss er, will er sich nicht nur auf die Low-Level-X-Aufrufe beschränken (was aus Aufwandsgründen selbstverständlich nicht empfehlenwert ist), sehr intensiv mit Begriffen wie Xt-Intrinsics, Motif-Widget-Set, Klassen-Hierarchien u.ä. auseinandersetzen. Die Einpassung der unterschiedlichen Fenstertypen, Schrifttypen, Füll- oder Randmuster, Pulldown/Popup-Menüs in das Gesamtbild ist beschwerlich.

Dazu kommt ein aufwendiges Austesten des ***Look and Feel***: auch kleine Änderungen bedingen stets ein Neuübersetzen und Binden, oft mit zweifelhaftem Erfolg. Mittlerweile werden von zahlreichen Firmen Software-Produkte angeboten, die dem GUI-Software-Entwickler diese formalen Arbeiten abnehmen, sogenannte Interface-Builder:

- DIATOOLS
- DIALOG Builder
- iXBUILD
- ISA DIALOG Manager
- ...

Solche Systeme beinhalten im wesentlichen

- einen WYSIWYG-Editor, mit dem der Entwickler den statischen Aufbau seiner Fenster und Fensterhierarchien definieren und beschreiben kann sowie
- einen Mechanismus zum Erstellen der Routinen, welche bei gewissen Aktionen aufgerufen werden, die sogenannten *Callbacks*.

Letztere wiederum steuern das eigentliche, dynamische Programmverhalten und sind daher vom Entwickler selbst zu schreiben.

Meistens besitzen diese Systeme auch einen Testmodus, mit dem sich das dyamische Look and Feel der Oberfläche überprüfen lässt.

Unabhängig davon bleibt es jedoch dem Entwickler nicht erspart, sich mit der Funktionalität des verwendeten Toolkit-Sets intensiv auseinanderzusetzen, d.h. insbesondere mit dem dynamischen Verhalten der einzelnen Widgets.

Interface-Builder erzeugen im wesentlichen einen (mehr oder weniger) portablen C-Code, der in das eigentliche Verarbeitungsprogramm für die Vorlaufphase einzubinden ist, bzw. in den das Verarbeitungsprogramm einzubinden ist, sowie optional dazu X-Resource-Dateien.

Der C-Code enthält die Toolkit-spezifischen Aufrufe zur Kreation der Fensterhierarchie. Alternativ dazu kann in manchen Fällen ein UIL-Interface (User Interface Language) erzeugt werden. Dieser UIL-Code ist aber nur zusammen mit einem sogenannten NIDL- Compiler verwendungsfähig.

Manche dieser Interface-Builder haben zwar den Vorteil, einen "kleinen" Code zu erzeugen, sind aber auf Laufzeitbibliotheken angewiesen, die spätestens beim Ablauf des Schnittstellenprogrammes aufgerufen werden. Solche Anwendungen sind daher kaum oder nicht portabel. Andere Interface Builder wiederum unterstützen nur proprietäre und damit ebenfalls nicht portable Fenstersysteme.

Interface-Builder existieren mittlerweile für alle gängigen Hardware-Plattformen wie SUN, HP, IBM/RS6000 ,... Da die meisten Interface-Builder selbst X-Window-basiert sind, reicht oft die Installation eines Interface-Builder auf *einer* Maschine aus.

Am Regionalen Hochschulrechenzentrum Kaiserslautem (RHRK) kommt das Produkt iXBUILD (Fa. IXOS) zum Einsatz.

Im folgenden sollen die automatisierten Verfahren beschrieben werden, mit denen unter Zuhilfenahme von ixbuild das Schnittstellenprogramm *xvp* (X-window access to vector processors) generiert wird (in Kaiserslautem für einen VP100 der Fa. Siemens Nixdorf AG mit Betriebssystem VSP/I).

3 Systemumgebung

In der Supercomputer-Abteilung des RHRK werden vorwiegend Workstations vom Typ HP/Apollo sowie Silicon Graphics/IRIS-Indigo eingesetzt. Dieses Spektrum ist dennoch recht heterogen durch

- unterschiedliche Betriebssystem-Releases innerhalb einer Systemfamilie,
- unterschiedliche X-Window-Releases (X11/R3 oder X11/R4),
- unterschiedliche OSF/Motif-Releases (1.0 oder 1.1).

Da es für keine der beiden Plattformen zum Beschaffungszeitpunkt (Anfang 1991) ixbuild verfügbar war, wurde eine SPARC-SUN (IPC, Monochrom-Schirm) beschafft, auf der problemlos ixbuild-V1.0 installiert werden konnte. Mittlerweile ist die ixbuild-Version V2.0 im Einsatz. Die Vemetzung aller dieser Maschinen erfolgt über ein Ethernet-LAN mit den Protokollen TCP/IP, FTP, TELNET, NFS und X-Window.

Die Schnittstellenprogramm-Generierung wird auf je einer Apollo- (als Master) sowie Indigo-Maschine durchgeführt.

Ixbuild läuft mittels xterm auf der als Server eingesetzten IPC-Maschine und benutzt über NFS die Platte der Apollo-Maschine, einmal für die ixbuild-spezifische Beschreibung der Fensterhierarchien (ixbuild-Daten), zum anderen für die Ablage des generierten C-Layout-Codes. Dadurch erspart man sich einmal Transporte der Layout-Codes über FTP zur Apollo-Maschine, zusätzlich reduzieren sich die Backup-Probleme.

Die Indigo-Maschine greift ebenfalls über NFS auf die Platte der Apollo-Maschine zu, und zwar auf alle Source-Dateien von xvp, d.h. sowohl auf den von ixbuild unabhängigen Teil als auch auf den von ixbuild erzeugten C-Layout-Code. Da dadurch nur *eine* Source-Variante von xvp existiert, ergibt sich die oben angesprochene Portabilität von xvp zwangsläufig. Dabei musste natürlich darauf geachtet werden, dass nur solch Systemschnittstellen vewendet wurden, die BSD und SYS5 gemeinsam sind, auch vom Aufruf her.

Die Erzeugung der xvp-Varianten erfolgt auf der Apollo- und Indigo-Maschine, aber nicht IPC-seitig, da dort die OSF/Motif-Entwicklungsbibliothek nicht zur Verfügung steht.

Auf der folgenden Seite ist die Einbindung der Entwicklungskonfiguration in das LAN vereinfacht dargestellt.

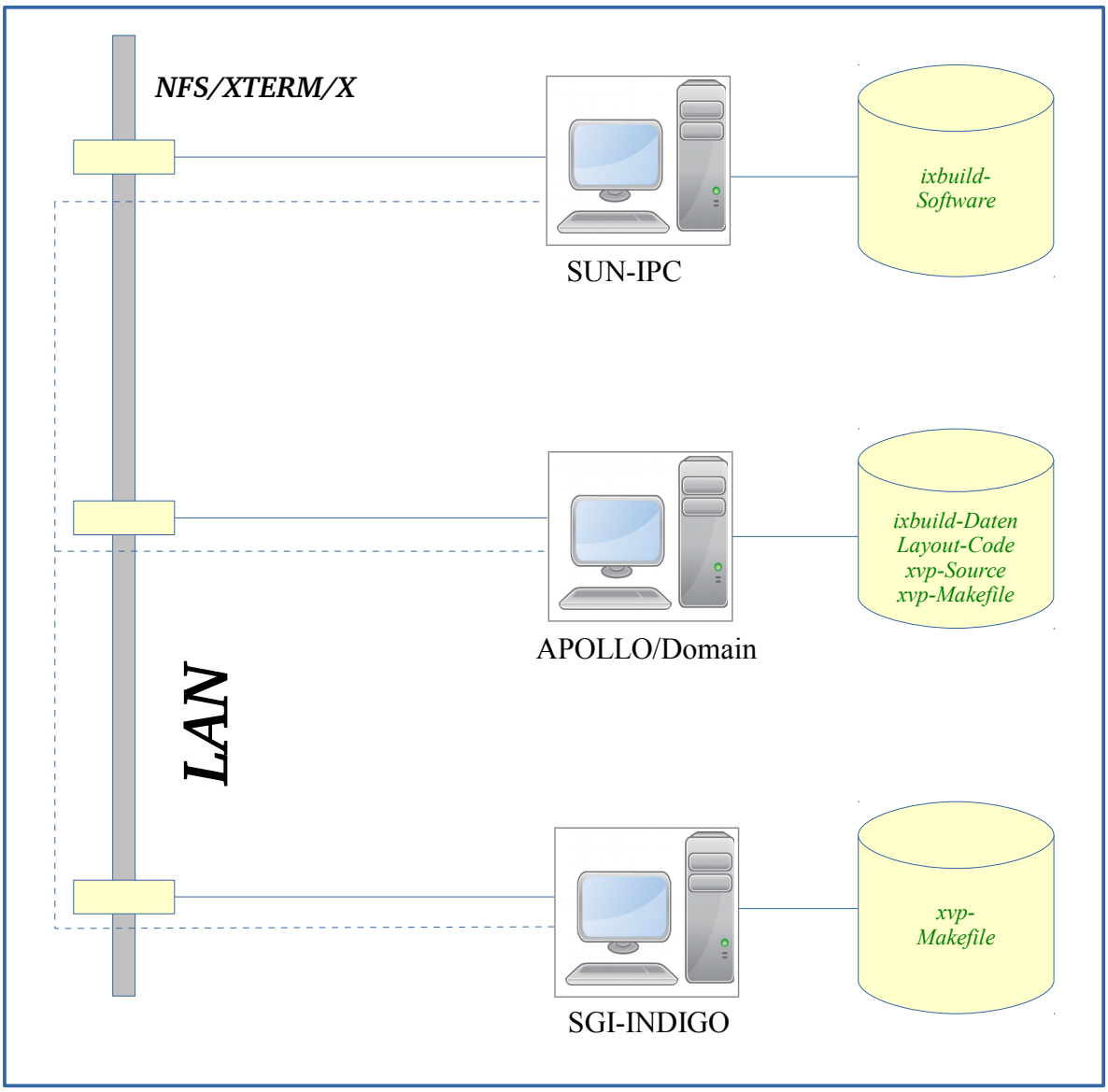

Abbildung 1. Entwicklungs-Konfiguration

4 Verwaltung der ixbuild-Anwenderdaten und Codeerzeugung

Da das Schnittstellenprogramm ***xvp*** einen Zugriff auf die verschiedenen Supercomputer-Funktionen ermöglichen soll, liegt es nahe, auch von den ixbuild-Daten her gesehen eine solche Aufteilung in die verschiedenen Funktionsgruppen vorzunehmen. Es ist dagegen nicht sinnvoll, die gesamte xvp-Fensterhierarchie in nur *einem* ixbuild-Datenfile abzulegen, denn

- aufgrund der Heterogenität der Supercomputer-Funktionen (z.B. Compilieren, Binden, Testen, Dateidienste, Informationsdienste, ..., insgesamt 36!) wäre eine Zusammenfassung in einer Datei sehr schwer zu handhaben;
- im ixbuild-WYSIWYG-Editor sind alle funktionsbezogenen Fenster zusammen nicht auf einmal verfünftig darstellbar;
- zur Ablaufzeit des Schnittstellenprogrammes würden alle funktionsbezogenen Fenster auf einmal erstellt werden, was aber sowohl von der Funktion her unerwünscht ist, aber auch von der Performance her (X-Server-Belastung!).

Die dadurch vorgegebene funktionsbezogene Aufteilung der ixbuild-Daten ist ohne Probleme mit ixbuild möglich, desgleichen das Umsetzen der Semantik der einzelnen Funktionen in die entsprechenden Pulldown- und Popup-Menüs, Push- und Toggle-Buttons, Labels usf., sowie die Zuordnung der Callbacks. Deren dynamisches Verhalten kann im ixbuild-Testmodus nicht ausgetestet werden.

Aufgrund der Vorgaben durch die xvp-Struktur kann man die generierten Layout-Codes nicht unmittelbar für die Make-Läufe zu übernehmen, denn

- es gibt keine Möglichkeit ixbuild mitzuteilen, in welches Fenster bzw. welche (Pop-up,...-) Shell eine Funktions-Hierarchie einzuhängen ist, d.h. ixbuild geht davon aus, dass eine Codeerzeugung den Code für ein monolithisches Hauptprogramm definiert;
- der Layout-Code ist nicht dazu geeignet, als Unterprogramm eingebunden zu wer den: fehlende oder (im Context) zu viele Header-Files, auf die erzeugten Widgets kann nicht ohne weiteres von außen zugriffen werden[1];
- die Bezüge auf eigene und X11-spezifische Bitmapfiles sind u.U. nicht brauchbar, beispielsweise wenn ihr Lagerort für ixbuild ein anderer ist als später zur Laufzeit des Schnittstellenprogrammes;
- in manchen Fällen lassen sich über ixbuild nicht alle Widgetresourcen steuern, die man aber bei der Kreation gerne mitangeben möchte;
- ...

Die nachstehenden Codefragmente sollen dies verdeutlichen:

1 Dieses Problem ist ab ixbuild V2.0 behoben

```
...
/*****************************************************
*     IXBUILD v2.0 Generated Code File
*     Interface Code for xvp_info
*****************************************************/
#include "xvp_info.layout.h"
#define MAX_ARGS 100
/*** Insert your code after this line ***/
...
{
...
n = 0;

XtSetArg(args[n],XmNlabelPixmap,BitmapFileToPixmap(Lable_log
"/usr/export/home/ipc01/backes/logo.bmp","black","white"));
      n++;
...
}
```

ungünstiger main-Bezug

Dieser Pfadname könnte sich ändern

Wie man sieht, empfiehlt sich eine Anpassung des erzeugten Layout-Code an die eigenen Bedürfnisse.

Um die erforderlichen Änderungen zu automatisieren, wurden mehrere awk-Scripts entwickelt, die nach der Layout-Code-Generierung auf der Apollo-Maschine aufgerufen werden und deren Aufgabe im wesentlichen darin besteht, den Layout-Code in die entsprechenden Stücke in Form von Include-Files aufzuteilen und diese den Bedürfnissen entsprechend anzupassen. Dieses Verfahren ist leider empfindlich gegen Versions-Änderungen von ixbuild, da das Format der generierten Layout-Files weder festgeschrieben noch dokumentiert ist. Die Praxis hat aber gezeigt (so beim Übergang von ixbuild V1 auf ixbuild V2), dass der Aufwand zur Anpassung der Scripts gering ist.

5 Erfahrungen beim Einsatz von ixbuild

Die Installation von ixbuild auf einer SUN-IPC-Maschine war, wie bereits erwähnt, problemlos durchzuführen. Beim Übergang von Version V1 auf Version V2 waren lediglich Feinheiten zu berücksichtigen.

Bei der Entwicklung einer solch komplexen Schnittstelle, wie oben beschrieben, kommt man jedoch nicht umhin, die ixbuild-Resource-Files auf die speziellen Bedürfnisse zuzuschneiden, so z.B.:

- Soll eine Widget-Klasse als Child einer anderen Widget-Klasse erlaubt sein soll oder nicht?
- Soll der WYSlWYG-Editor Geometrie-Änderungen bei bestimmten Klassen erlauben oder nicht?
- Welche Ressourcen sollen im sog. Customize-Editor änderbar sein?

Da das Layout dieser Resourcen ausführlich beschrieben ist, sind auch diese Anpassungen recht einfach durchführbar (z.B. mit awk-Scripts). Dies hat den Vorteil, dass sie permanent sind. Teilweise ist man sogar zu dieser Vorgehensweise gezwungen, da nicht alle diese Anpassungen beim Start von ixbuild oder später über dessen Hauptmenü anwählbar sind. Jedoch auch hier ist Vorsicht angebracht bei Versionsübergängen.

Die Funktionalität des WYSIWYG-Editor sowie der einzelnen Resource-Editoren ist mehr als ausreichend. Darüberhinaus erleichert ab Version V2.0 die Möglichkeit der Mehrfachselektion von Widgets (und gemeinsamen Operationen darauf) das Arbeiten wesentlich.

Lediglich das Umbauen einer Widgethierarchie gestaltet sich nicht so einfach, da (heterogene) *Mengen von Widgets* verschiedener Stufen gemeinsam in einer anderen Stufe nicht neu einzubringen sind (unzureichende COPY-Funktion).

Ebenfalls nicht ganz befriedigend ist die Vorgehensweise von ixbuild beim Editieren der Font-Resourcen von Labels, Push-Buttons usf..

Diese beiden Probleme werden aber mit der ixbuild-Version V2.1 ausgeräumt sein, wie sich bei (den zahlreichen und sehr kooperativen) Gesprächen sowohl mit dem Entwickler als auch mit der Produktbetreuung ergeben hat.