Carl von Ossietzky Universität Oldenburg

Abteilung Rechnernetze & Telekommunikation

Das Simulationssystem Lancelot

Dokumentation

Autoren:

Gerd Block

Thorsten Sterzenbach

Oldenburg im Juni 1997

Inhaltsverzeichnis

1 Einleitung

Die zunehmende Verfügbarkeit leistungsfähiger Grafikhardware bedingt die unaufhaltsame Verbreitung von grafischen Benutzungsoberflächen in allen Bereichen der Software-Entwicklung. Diese Benutzungsoberflächen ermöglichen eine adäquate und strukturierte Darstellung von Informationen und entsprechen damit den Forderungen der Software-Ergonomie nach zunehmender Benutzungsfreundlichkeit.

Die vorliegende Dokumentation gibt eine Einführung in den Aufbau und die Bedienung des Simulationssystems **Lancelot**. Das System soll den Aufbau eines Simulationsmodells erheblich vereinfachen. Dazu steht eine grafische Benutzungsoberfläche zur Verfügung, die den Modellaufbau durch grafikorientierte Bearbeitungsfunktionen am Bildschirm ermöglicht. Die eingegebenen Modelldaten werden dann durch den Modellcompiler von Lancelot in den Code eines Simulationsprozesses übersetzt. Dieser wird dabei in der Simulationssprache C-SIM formuliert, die von Herrn Professor Kowalk entwickelt wurde (siehe [Kowalk93]). Der C-SIM-Code kann anschließend durch einen C-Compiler in ausführbaren Maschinencode transformiert werden. Nach der Codegenerierung kann der Simulationsprozeß gestartet werden. Dabei wird dann sein Ablauf in der Benutzungsoberfläche von Lancelot visualisiert.

Vor der Entwickung von Lancelot gab es für C-SIM bereits weitere Ansätze zur Visualisierung und Auswertung eines Simulationsablaufs. Dazu wurden in dem vom Programmierer abgefaßten C-SIM-Code entsprechende Funktionen eingefügt, die dann während der Laufzeit die Ausführung des Prozesses grafisch visualisierten (XSV/XSVS, siehe [Ono-Tesfaye93]). Die Programmierung eines Simulationsprozesses auf dieser Ebene ist allerdings sehr aufwendig und unnötig kompliziert. Es entstand der Wunsch nach einer Programmierumgebung, die eine bequeme Definition eines Simulationsmodells erlaubt und außerdem automatisch für die Visualisierung des Simulationsablaufs sorgt.

Für die Arbeit mit Lancelot sind Kenntnisse der Simulationssprache C-SIM und der Programmiersprache C erforderlich.

2 Aufbau von Lancelot

Das Simulationssystem Lancelot läßt sich in zwei wesentliche Komponenten unterteilen: Grafische Benutzungsoberfläche und Modellcompiler. Die Benutzungsoberfläche wurde im Rahmen einer Diplomarbeit mit der Anforderung entwickelt, weitgehend unabhängig von einer konkreten Simulationssprache zu sein. Dabei wurden allerdings die speziellen Anforderungen und Eigenschaften von C-SIM als Beispiel einer entsprechenden Simulationssprache berücksichtigt. Als Ergebnis entstand die Benutzungsoberfläche **Xartus.** Das Programm besitzt eine Schnittstelle, über die ein Applikationsprozeß an alle eingegebenen Daten gelangen kann. Außerdem können über die Schnittstelle verschiedene grafische Aktionen in der Oberfläche ausgelöst werden. Weitere Informationen zu den Konzepten der Benutzungsoberfläche sind in [Sterzenbach95] enthalten.

Der Modellcompiler liegt in zwei Versionen vor, die ebenfalls im Rahmen von Diplomarbeiten entstanden. In ihren Anforderungen berücksichtigten diese Arbeiten dabei die Eigenschaften und Funktionalität von C-SIM und Xartus. In der ersten Version entstand der Modellcompiler **Xcalibur**, der die Daten, die mit Hilfe der Benutzungsoberfläche eingegeben werden, auf die Menge der C-SIM-Programme abbildet. Näheres dazu kann in [Block95] nachgelesen werden. Die Folgeversion heißt bigMC (Näheres in ...). Die folgende Abbildung zeigt den groben Aufbau des Simulationssystems Lancelot.

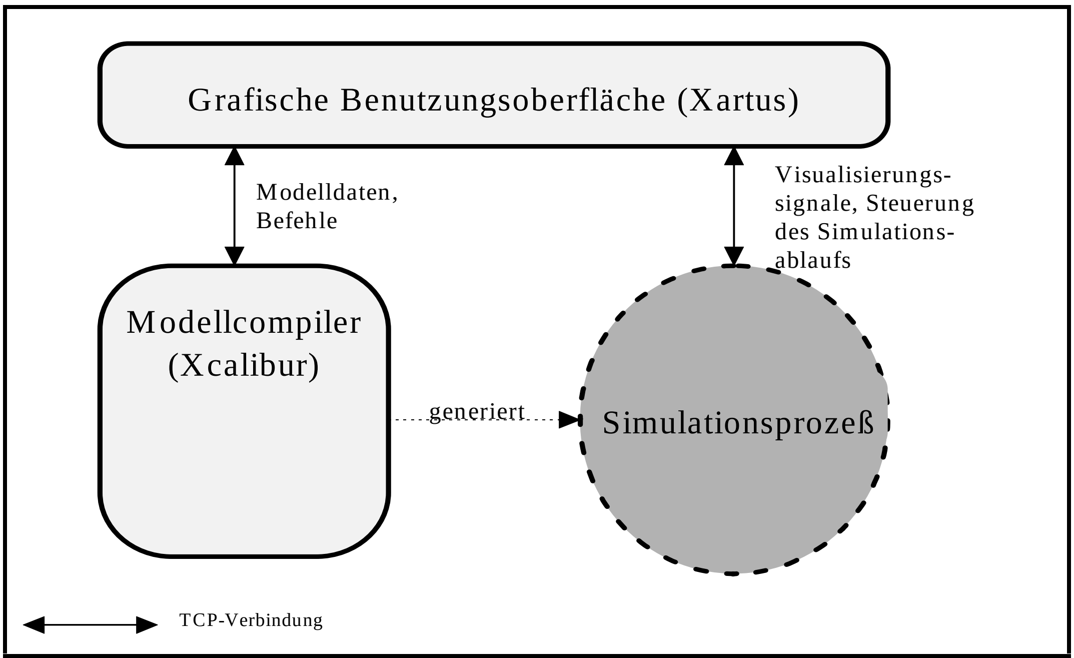


Abbildung 1: Aufbau von Lancelot

Nachdem ein Modell in C-SIM-Code transformiert wurde, wird es durch einen C-Compiler in den Maschinencode eines Simulationsprozesses übersetzt. Dieser Prozeß kann dann über die Benutzungsoberfläche gestartet werden. Während der Ausführung visualisiert dieser Prozeß seinen Ablauf auf dem Bildschirm. Dazu werden verschiedene grafische Aktionen innerhalb der von dem Benutzer aufgebauten, grafisch repräsentierten Objektstrukturen durchgeführt

3 Installation

Das Simulationssystem Lancelot wurde auf einer Linux-Plattform entwickelt und auf die Netzwerkumgebung im ORLAB der Abteilung Rechnernetze und Telekomunikation (Fachbereich Informatik, Universität Oldenburg) übertragen. Die Rechner in dieser Umgebung laufen entweder unter System V Release 4.0 oder HP-UX. In der derzeitigen Version ist die Installation speziell an diese drei Umgebungen (Linux, Sun, HP) angepaßt. Darüber hinaus ist die Installation auf anderen Rechnern durch die bei der Entwicklung verwendeten frei verfügbaren Entwicklungswerkzeuge gesichert, zur Zeit aber noch nicht realisiert. Die minimalen Systemvoraussetzungen (Software) für die Übersetzung des Programmpaketes werden im Abschnitt „Installation einer Vollversion“ dieses Kapitels beschrieben.

Es gibt verschiedene Möglichkeiten zur Installation von Lancelot:

- Einfache Installation einer Benutzerversion zur Arbeit in der Netzwerkumgebung des ORLAB’s
- Installation einer übersetzten Vollversion ohne Sourcen für einen bestimmten Maschinentyp (Sun, HP, Linux)
- Installation einer Vollversion, die anschließend übersetzt werden kann

3.1 Einfache Installation einer Benutzerversion (ORLAB)

Dies ist die einfachste Möglichkeit, um mit Lancelot in der Umgebung des ORLAB’s zu arbeiten und dabei auf eigene Simulationsmodelle und Icons zurückzugreifen. Bei dieser Installation werden alle Voraussetzungen geschaffen, damit der Benutzer in seiner eigenen Umgebung und mit seinen erstellten Modelldaten arbeiten kann. Die Programmdateien von Lancelot sind aber global in den Systemverzeichnissen installiert und werden beim Aufruf von dort aus gestartet. Beim Programmstart liest Lancelot dann die benutzereigenen Daten aus den bei der Installation angelegten Benutzerverzeichnissen. Durch den globalen Aufruf von Lancelot im System ist sichergestellt, daß der Benutzer immer mit der gerade aktuellsten Version von Lancelot arbeitet.

Für die Installation wird die gepackte Benutzerversion `in_user.tar.gz` benötigt. In einem vom Benutzer ausgewählten Verzeichnis (Zielverzeichnis) wird dann das Paket durch

```
gunzip in_user.tar.gz
```

und

```
tar -xvf in_user.tar
```

entpackt. Dabei wird im Zielverzeichnis das Verzeichnis *Lancelot* angelegt, das folgende Verzeichnisstruktur enthält:

- Lancelot
  - bin
  - CSIMPROG
  - Icons
  - Models
  - Xartus
  - Xcalibur

Auf die Bedeutung der einzelnen Verzeichnisse wird im Anschluß an die Erläuterungen zu den verschiedenen Installationsweisen eingegangen.

Vor dem ersten Start von Lancelot müssen nun noch einige Pfadangaben in der Datei `XRes` im Verzeichnis *Lancelot* geändert werden. Im Abschnitt „#1#“ dieser Datei befindet sich folgender Abschnitt zur Festlegung der Arbeitspfade:

```
!Benutzerpfade


Xartus*workpath: ..

Xartus*programpath: ..

Xartus*modelpath: ../Models

Xartus*fileServer*initialDirectory: ../Models/


Xartus*applicationpath1: ..
```

Die abgebildeten Belegungen sind für die weiter unten beschriebenen Vollversionen von Lancelot vorgesehen. Für die in diesem Abschnitt besprochene Installation muß die Belegung der Pfade wie folgt verändert werden:

> Die Ressource `Xartus*workpath` muß mit dem vollständigen Verzeichnispfad zum Zielverzeichnis (im Benutzer-Verzeichnisbaum) belegt werden, der um das Verzeichnis *Lancelot* erweitert wird. Wurde Lancelot also z.B. im Home-Verzeichnis des Benutzers *artus* installiert, und lautet der Pfad zu diesem Benutzer-Verzeichnis */home/student/artus*, so lautet die korrekte Belegung */home/student/artus/Lancelot*.
>
> Der Pfad `Xartus*programpath` lokalisert die benötigten Programmdateien von Lancelot in den Systemverzeichnissen. Er lautet in der Umgebung des ORLAB’s auf allen Maschinen */usr/orlab/lancelot/Lancelot*.
>
> Für `Xartus*modelpath` muß der Pfad zu den Benutzermodellen eingesetzt werden. Er lautet *Models*. Das gleiche gilt für `Xartus*fileServer*initialDirectory`.
>
> `Xartus*applicationpath1` wird schließlich mit dem gleichen Pfad wie `Xartus*programpath` initialisiert.

Damit ist die Installation abgeschlossen. Durch den Aufruf von `lancelot` im Verzeichnis *Lancelot* wird das Simulationssystem gestartet. `lancelot` ist dabei ein Shellskript das im voreingestellten Suchpfad des Benutzers liegt.

Nach den beschriebenen Änderungen in der Datei `XRes` kann Lancelot auf jeder Maschine des ORLAB’s gestartet werden. Die erstellten Modelldaten sind von dem jeweiligen Maschinentyp unabhängig, nur die übersetzten Simulationsprozesse müssen bei einem Wechsel auf eine andere Maschine neu übersetzt werden.

3.2 Installation einer übersetzten Vollversion ohne Sourcen

Diese Installation enthält das übersetzte Programmpaket für einen der oben genannten Maschinentypen. Zur Installation wird die gepackte Datei `in_bin.tar.gz` für den gewählten Maschinentyp benötigt. Diese Datei wird nun in das Verzeichnis kopiert, in dem Lancelot installiert werden soll (Zielverzeichnis). Das Entpacken der Datei geschieht mit

```
gunzip in_bin.tar.gz
```

und

```
tar -xvf in_bin.tar
```

Dabei wird im Zielverzeichnis das Verzeichnis *Lancelot* angelegt, das folgende Verzeichnisstruktur enthält:

- Lancelot
  - bin
  - CSIMPROG
  - Icons
  - Models
  - Xartus
  - Xcalibur

Nun kann nach dem Wechsel in das Verzeichnis *Lancelot/Xartus* das Simulationssystem durch den Aufruf von `lancelot` gestartet werden.

Für die Übersetzung von Simulationsprogrammen durch Lancelot ist es unerläßlich, daß eine Version des C-Compilers von GNU™ (Version 2.5.8 oder höher) auf dem System installiert ist und ohne Probleme aus jedem Benutzerverzeichnis aufgerufen werden kann!

3.3 Installation einer Vollversion

Diese Installation umfaßt sämtliche Sourcen von Lancelot und dient zur Generierung der beiden zuvor beschriebenen Installationsversionen und zur Weiterentwicklung des Systems. Nach dem Entpacken muß das System übersetzt werden.

Zur Installation wird zunächst die Datei `all.tar.gz` benötigt. Diese wird dann in einem ausgewählten Benutzerverzeichnis (Zielverzeichnis) durch

```
gunzip all.tar.gz
```

und

```
tar -xvf all.tar
```

entpackt. Dabei wird im Zielverzeichnis das Verzeichnis *Lancelot* angelegt, das folgende Verzeichnisstruktur enthält:

Lancelot

- bin
- CSIMPROG
- IConfig
- Icons
- Models
- Tools
- User
- Widgets
- Xartus
- Xcalibur

Nach dem Verzeichniswechsel in *Lancelot* kann das Simulationssystem nun durch den Aufruf von `make all` (GNU™-make) übersetzt werden. Die Dauer der Übersetzung ist maschinenabhängig und liegt zwischen ca. 10 Minuten (Intel™ Pentium 90 unter Linux) und 30 Minuten (Sun im ORLAB). Nach der erfolgreichen Übersetzung kann das Programm nach dem Wechsel in das Verzeichnis *Xartus* durch den Aufruf von `lancelot` gestartet werden.

Wie bereits erwähnt, wurde Xartus unter Linux entwickelt und auf die oben genannten Maschinentypen übertragen. Zur Übersetzung der Vollversion sind folgende Entwicklungswerkzeuge erforderlich:

C++-Compiler von GNU™ (Version 2.5.8 oder höher). Dieser Compiler wird auch während des Ablaufs von Lancelot zur Übersetzung von Simulationsprogrammen benötigt!

X11R6-Development-Kit einschließlich der Athena-Widgets (am besten vom MIT). Die Widgets sind normalerweise bei jeder gängigen X11-Version standardmäßig verfügbar. Optional kann das Erscheinungsbild der Benutzungsoberfläche erheblich verbessert werden, wenn statt der einfachen Athena-Widgets die 3D-Athena-Widgets verwendet werden (Library `Xaw3d`).

C-Funktionen zum Aufbau einer TCP-Verbindung mit Sockets.

3.4 Bedeutung der Verzeichnisse

Bei jeder der beschriebenen Installationen wurde beim Auspacken der entsprechenden gepackten Datei eine Verzeichnisstruktur angelegt. In diesem Abschnitt soll nun die Bedeutung der einzelnen Verzeichnisse erläutert werden.

bin: In diesem Verzeichnis werden die übersetzten und ablauffähigen Simulationsprozesse abgelegt. Aus diesem Verzeichnis heraus werden die Prozesse nach der Übersetzung gestartet.

CSIMPROG: In diesem Verzeichnis wird während der Übersetzung eines Simulationsprogramms, der von Lancelot erzeugte C-SIM-Code in der Datei `CSIMPROG.c` abgelegt. Für den fortgeschrittenen C-SIM-Programmierer kann diese Datei hilfreichen Aufschluß über evtl. aufgetretene Fehlersituationen während der Übersetzung des Simulationsprogramms geben.

IConfig (nur Vollversion): Dieses Verzeichnis enthält den Menü-Manager (genauere Informationen finden sich in [Sterzenbach95]).

Icons: In diesem Verzeichnis werden die vom Benutzer erzeugten Icons abgelegt. Es besteht prinzipiell die Möglichkeit, Icons auch außerhalb von Lancelot zu erzeugen. Diese müssen dann im **Bitmap**-Format in diesem Verzeichnis abgelegt werden.

Models: In diesem Verzeichnis werden standardmäßig die vom Benutzer innerhalb von Lancelot erzeugten Modelldaten abgelegt. Diese Standardeinstellung kann durch Veränderung der Ressourcen `Xartus*modelpath` und `Xartus*fileServer*initialDirectory` in der Datei `XRes` verändert werden.

Tools: In diesem Verzeichnis sind die Funktionen zur TCP-Kommunikation und zur String-Verarbeitung von Xartus abgelegt.

User: Dieses Verzeichnis enthält die Xs-Funktionen (siehe Kapitel „Erweiterte Ausgabemöglichkeiten“) zur Kommunikation mit Xartus.

Widgets (nur Vollversion): In diesem Verzeichnis werden einige zusätzliche Widgets definiert, die von Xartus verwendet werden.

Xartus: Dieses Verzeichnis enthält alle zur Funktion der Benutzungsoberfläche relevanten Programmdateien. Bei der Vollversion sind hier auch die Sourcen der Benutzungsoberfläche abgelegt.

Xcalibur: Dieses Verzeichns enthält alle für den Modellcompiler von Lancelot wichtigen Programmdateien. Bei der Vollversion sind hier auch die Sourcen abgelegt.

4 Statischer Modellaufbau

In diesem Kapitel werden nun die Möglichkeiten von Lancelot zum Aufbau eines Simulationsmodells beschrieben. Der Aufbau läßt sich dabei in zwei grundlegende Schritte, den statischen und den dynamischen Modellaufbau, zerlegen. Zunächst werden wir den statischen Modellaufbau erläutern. Dazu werden verschiedene Objektklassen vorgestellt, die als Einzelbestandteile für die Modellierung eines Gesamtmodells verwendet werden können.

4.1 Objektstrukturen

Alle zur Modellierung mit Lancelot geeigneten Simulationsmodelle bestehen aus einer Anzahl von Grundelementen, die zueinander in Beziehung gesetzt werden können. Als Beispiel betrachten wir hier das Modell eines Rechnernetzes, das aus einer beliebigen Anzahl von Rechnern besteht. Diese sind durch Kommunikationsverbindungen miteinander verbunden, so daß ein Datenaustausch zwischen den Rechnern möglich ist. Die Grundelemente des Modells sind die einzelnen Rechner und die Beziehungen zwischen den Elementen sind durch die Verbindungen gegeben. Auf der Ebene von Lancelot werden die Grundelemente durch ***Basisobjekte*** und die Beziehungen durch ***Verbindungsobjekte*** abgebildet. Jedes Basisobjekt besitzt eine symbolische Repräsentation innerhalb der Benutzungsoberfläche. Diese kann durch Linienzüge mit anderen symbolischen Repräsentationen von Basisobjekten verbunden werden. Ein Linienzug ist dabei die grafische Repräsentation eines Verbindungsobjekts, das jeweils zwei Basisobjekte zueinander in Beziehung setzt.

Nun besitzt ein Rechnernetz außer seinem statischen Aufbau auch noch ein dynamisches Verhalten: Den Austausch von Informationen über die Kommunikationsverbindungen. Um dieses Verhalten zu modellieren, werden bei Lancelot ***Iconobjekte*** verwendet, die ebenfalls eine symbolische Repräsentation besitzen. Diese kann über den Linienzug eines Verbindungsobjekts zwischen zwei Basisobjekten bewegt werden und auf diese Weise das dynamische Verhalten eines Modells veranschaulichen.

Prinzipiell würden die bisher vorgestellten Objektklassen bereits zur Modellierung der betrachteten Menge von Simulationsmodellen ausreichen. Um darüber hinaus einen übersichtlichen und strukturierten Aufbau von größeren Modellen zu unterstützen, stellt Lancelot die Klasse der ***komplexen Objekte*** zur Verfügung. Komplexe Objekte können Objektstrukturen enthalten, die aus Basisobjekten, Verbindungsobjekten und anderen komplexen Objekten bestehen, und diese mit Hilfe einer symbolischen Repräsentation verbergen.

Eine fünfte Klasse von Objekten sind die ***Typobjekte***, die zur Definition häufig verwendeter Datenstrukturen (Typen) innerhalb anderer Objekte benutzt werden. Eine ausführliche Beschreibung der Typobjekte erfolgt bei der Erläuterung des dynamischen Modellaufbaus.

Bei der Entwicklung eines Modells mit Lancelot muß das Typkonzept eingehalten werden, d.h. die zum Modellaufbau verwendeten Objekte sind Instanzen von Objekttypen, die vorher entsprechend definiert werden müssen. Instanzen werden bei Lancelot als vollständige Kopien des Typs erzeugt. Dementsprechend existiert nach dem Erzeugen einer Instanz keinerlei Verbindung mehr zum zugrunde liegenden Typ. Spätere Änderungen des Typs wirken sich dementsprechend nicht auf die bis dahin erzeugten Instanzen aus.

Nach dem Programmstart von Lancelot erscheinen zwei Formulare (innerhalb von Windows) auf dem Bildschirm. Das größere davon dient zum Aufbau des eigentlichen Modells und wird ***Root-Formular*** genannt. Das andere Formular gehört zum ***Typ-Editor*** und ermöglicht die Konstruktion beliebiger Typen für die verschiedenen Objektklassen. Die folgende Abbildung zeigt das Formular des Typ-Editors.

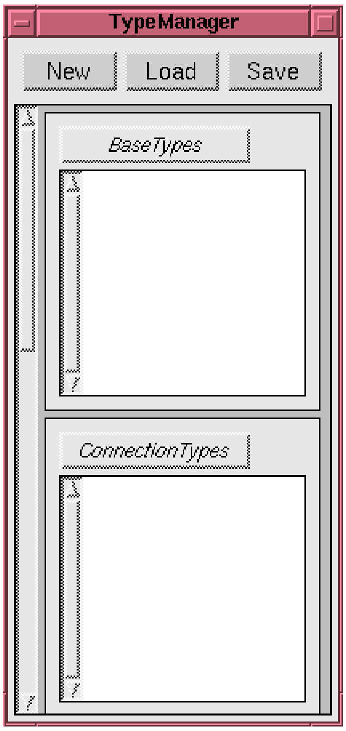


Abbildung 2: Formular des Typ-Editors

Das Formular enthält fünf verschiedene Bereiche (Typ-Bereiche) zur Definition von Typen der einzelnen Klassen. Anfänglich werden im Bildschirmausschnitt des Managers aus Platzgründen davon jeweils nur zwei Bereiche dargestellt. Die andere Bereiche können mit

Hilfe des Scrollbars in diesen Ausschnitt verschoben werden. Durch Vergrößern des zum Manager gehörenden Windows können aber auch alle Bereiche gleichzeitig angezeigt werden.

Am oberen Rand des Typ-Editors befindet sich eine Menüleiste mit den Buttons *New*, *Load*, und *Save*. Nach der Auswahl eines dieser Menübuttons ergibt sich die Auswahl zwischen *Base*, *Connection*, *Complex*, *Icon* und *Type*. Über *New* können neue Typen der einzelnen Klassen erzeugt werden. *Load* und *Save* ermöglichen das Laden und Speichern der Typen. Dies geschieht dabei getrennt nach den verschiedenen Klassen.

Im folgenden werden wir nun die Definition der einzelnen Objekttypen erläutern.

4.2 Definition von Typen für Basisobjekte

Ein neuer Basisobjekt-Typ wird durch Auswahl des Buttons *New* im Menübereich des Typ-Editors und der anschließenden Auswahl von *Base* erzeugt.

Nach dieser Auswahl erzeugt Lancelot das Eingabeformular der Basisobjekte auf dem Bildschirm (Abbildung 3). Dieses enthält einen Eingabebereich für den Namen und einen weiteren zur Definition des dynamischen Verhaltens der Instanzen des Basisobjekt-Typs. Die Definition des dynamischen Verhaltens durch C-SIM-Programmcode wird später noch ausführlich behandelt.

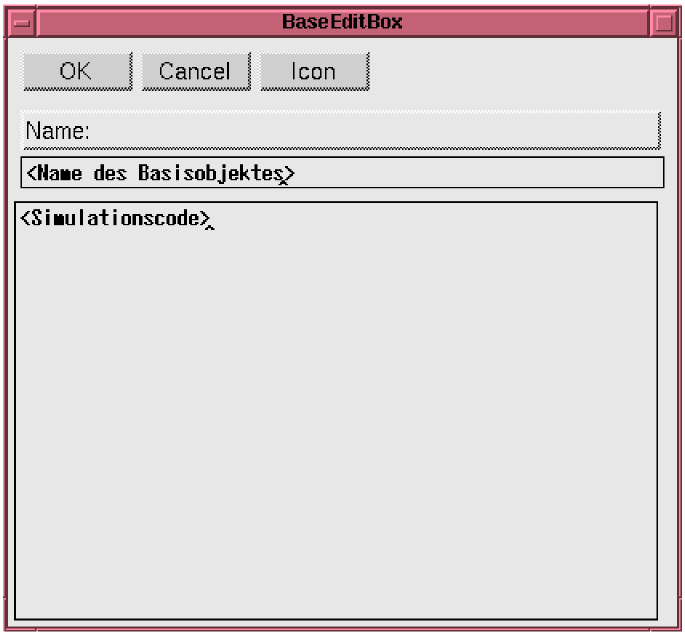


Abbildung 3: Eingabeformular der Basisobjekte

Außerdem enthält das Eingabeformular noch die Button *OK*, *Cancel* und *Icon*. Dabei bewirkt die Auswahl von *Cancel* den Abbruch und *OK* den ordnungsgemäßen Abschluß der Definition des Typs. Die Auswahl von *Icon* erzeugt das Formular der Iconauswahl auf dem Bildschirm (Abbildung 4). Innerhalb dieses Formulars kann der symbolischen Repräsentation des Typs und seiner Instanzen ein Icon zugeordnet werden. Das Erzeugen von Icons wird später zusammen mit der Funktionalität des Root-Formulars erläutert. Für die Darstellung des ausgewählten Icons kann eine Vorder- und Hintergrundfarbe festgelegt werden. Dies geschieht durch Auswahl einer beliebigen Farbe innerhalb der Farbskala und das anschließende Betätigen von *Foreground* bzw. *Background*.

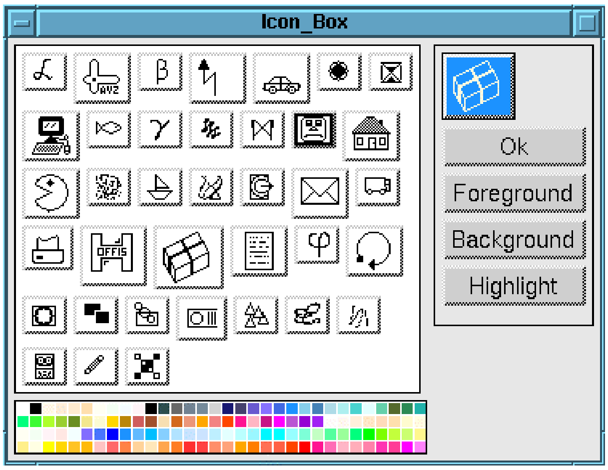


Abbildung 4: Formular der Iconauswahl

Das Formular enthält in der rechten oberen Ecke ein Anzeigenfeld, in dem die aktuelle Darstellung des gewählten Icons (Icon, Vordergrund, Hintergrund) dargestellt wird. Zusätzlich kann der Darstellung über *Highlight* eine Highlightfarbe zugeordnet werden, die im Simulationsablauf einen besonderen Zustand des Objekts kennzeichnet. Die gewählte Highlightfarbe kann überprüft werden, indem der Mauszeiger über das Anzeigenfeld der aktuellen Darstellung bewegt wird. Mit *OK* kann das Formular wieder verlassen werden.

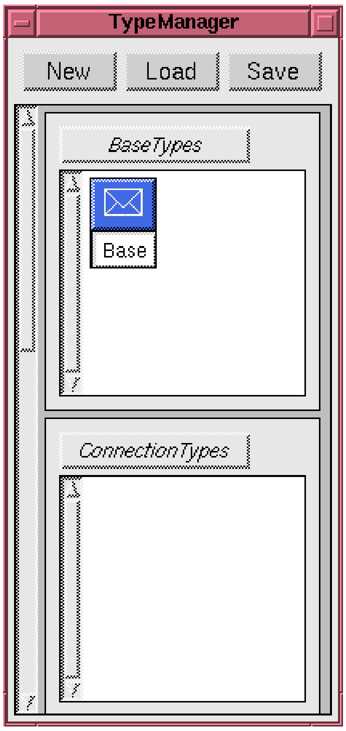


Abbildung 5: Symbolische Darstellung eines Basisobjekts

Nach dem ordnungsgemäßen Abschluß der Definition des Typs mit *OK* im Eingabeformular wird die symbolische Repräsentation des erzeugten Typs im Typ-Bereich der Basisobjekte dargestellt (Abbildung 5). Damit ist die Definition eines Basisobjekt-Typs abgeschlossen.

4.3 Definition von Typen für Verbindungsobjekte

Durch die Auswahl von *New* und anschließend *Connection* im Menübereich des Typ-Editors wird ein neuer Verbindungsobjekt-Typ erzeugt. Dabei wird das Eingabeformular der Verbindungsobjekte auf dem Bildschirm abgebildet (Abbildung 6).

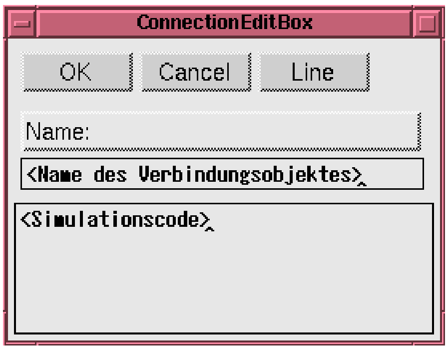


Abbildung 6: Eingabeformular der Verbindungsobjekte

Das Formular enthält zwei Eingabebereiche für den Namen und den Code des Typs und außerdem die Button *OK*, *Cancel* und *Line*. *OK* bestätigt den ordnungsgemäßen Abschluß der Definition und bewirkt das Schließen des Eingabeformulars und die grafische Darstellung des Typs im Typ-Bereich der Verbindungsobjekte. *Cancel* verwirft alle gemachten Eingaben.

Über *Line* gelangt man in das Line-Formular (Abbildung 7), das eine Veränderung der Linienattribute des zu den Instanzen des Typs gehörenden Linienzugs ermöglicht. Das Festlegen einer Farbe geschieht durch eine entsprechende Auswahl in der Farbskala und die gewünschte Linienstärke kann in das Eingabefeld LineWidth (Wertebereich: 1-10) eingetragen werden.

Außerdem ist die Definition beliebiger Strichmuster möglich. Dazu kann eine Anzahl von nicht-negativen Zahlen, jeweils durch Kommata getrennt, in das Eingabefeld NumberOfDashpoints eingeben werden. Jedes Element der eingegebenen Liste ist dabei die Länge eines Mustersegments (in Pixel). Gestrichelte Linien werden als wechselnde Segmente gezeichnet, d.h. die Farbgebung von einem Segment zu einem folgenden wechselt zwischen Vorder- und Hintergrundfarbe. Die Eingabe der Zeichenkette „4,1,2,1" bedeutet z.B., daß das erste Segment in der Vordergrundfarbe eine Länge von 4 Pixeln hat. Ihm folgt ein Segment mit der Länge 1 in der Hintergrundfarbe. Als nächstes folgt ein Segment der Länge 2 in der Vordergrundfarbe und dann wieder ein Segment der Länge 1 in der Hintergrundfarbe.

Wählt man Schwarz als Vordergrund- und Weiß als Hintergrundfarbe, dann besitzt das Muster folgendes Aussehen:

Eine mit diesem Muster gezeichnete Linie stellt sich wie folgt dar:

Eine Linie besteht aus Wiederholungen des angegebenen Musters. Die Gesamtlänge eines Musters ist die Summe aller Elemente der eingegebenen Liste.

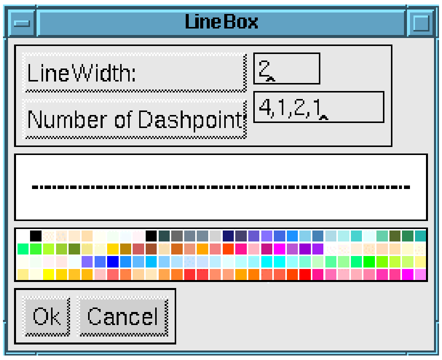


Abbildung 7: Line-Formular

Durch die Festlegung von Linienstärke, Farbe und Strichmuster besteht die Möglichkeit, eine Vielzahl von Verbindungsobjekt-Typen zu erstellen, die durch ihre unterschiedlichen Darstellungen gut voneinander unterscheidbar bleiben. Falls bezüglich der symbolischen Darstellung eines Verbindungsobjekt-Typs keine spezielle Festlegung vorgenommen wird, werden alle Linienzüge von Instanzen dieses Typs mit der Stärke 1 in der Vordergrundfarbe Schwarz und ohne Strichmuster gezeichnet (Standardeinstellung). Die Hintergrundfarbe der Linienzüge ist immer Weiß.

Nachdem der Linienzug das gewünscht Aussehen hat, kann das Line-Formular mit *OK* wieder geschlossen werden. Mit *Cancel* werden alle vorgenommenen Änderungen übergangen.

4.4 Definition von Typen für komplexe Objekte

Die komplexen Objekte sollen den strukturierten und übersichtlichen Modellaufbau unterstützen. Mit ihrer Hilfe können aus Basisobjekten, Verbindungsobjekten und komplexen Objekten bestehende Objektstrukturen zu einer Einheit zusammengefaßt und verborgen werden.

Ein neuer komplexer Objekt-Typ entsteht durch den Aufruf von *New* und Auswahl von *Complex* im Menübereich des Typ-Editors. Anschließend bildet Lancelot das Eingabeformular der komplexen Objekte auf dem Bildschirm ab (Abbildung 8). Es enthält einen Eingabebereich für den Namen des Typs, einen rechteckigen Grafikbereich (Drawing-Area) und zwei Button. Nach Auswahl von *OK* wird das Eingabeformular geschlossen und der Typ symbolisch in dem Typ-Bereich der komplexen Objekte dargestellt. Der Aufruf von *Icon* führt in das bereits beschriebene Formular der Iconauswahl, in dem der symbolischen Repräsentation des Typs ein Icon zugeordnet werden kann.

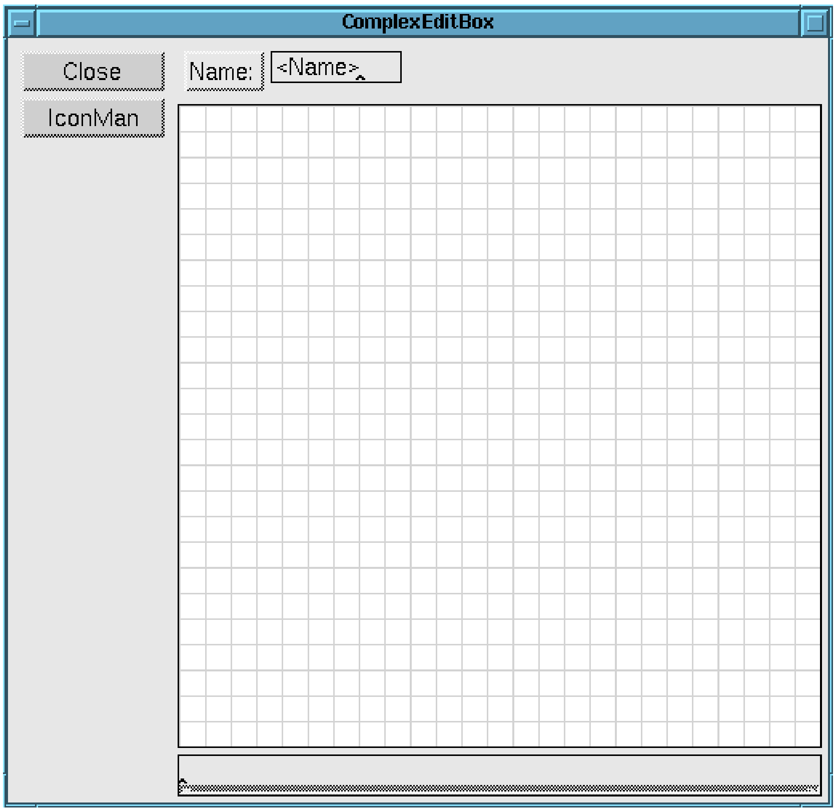


Abbildung 8: Eingabeformular der komplexen Objekte

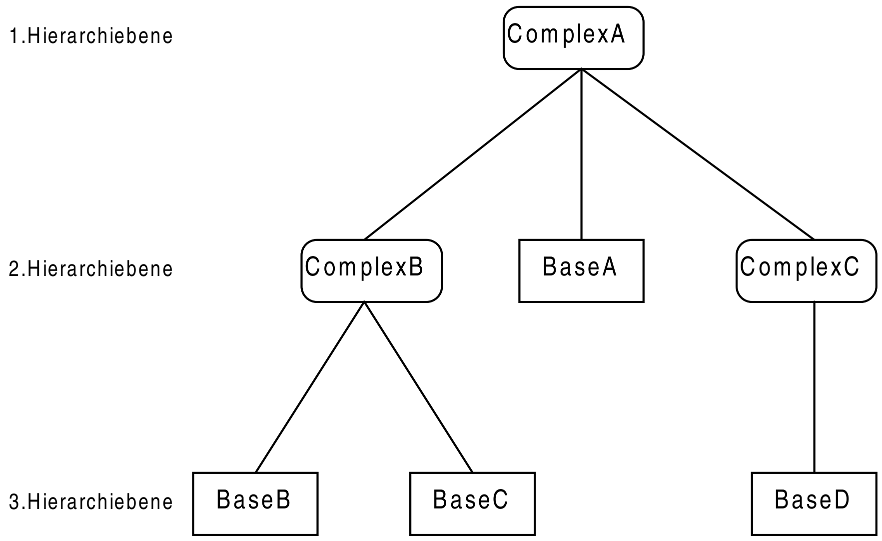
4.4.1 Einfügen von Objekten

In den Grafikbereich des komplexen Objekts können Instanzen von Basisobjekt-Typen und komplexen Objekt-Typen eingefügt werden. Dazu wird zunächst der gewünschte Typ im Typ-Bereich der Basisobjekte bzw. komplexen Objekte ausgewählt (linke Maustaste). Die Auswahl eines Typs wird durch eine grafische Hervorhebung gekennzeichnet. Anschließend wird durch Drücken der linken Maustaste an einer freien Position im Grafikbereich ein Popup-Menü geöffnet, das die Auswahl der entsprechenden Objekt-Klasse ermöglicht. Durch die Wahl von *Base* wird eine Instanz des ausgewählten Basisobjekt-Typs erzeugt und an dieser freien Position symbolisch dargestellt. Durch die Wahl von *Complex* wird entsprechend ein neues komplexes Objekt erzeugt und eingefügt. Als freie Position wird dabei eine Stelle im Grafikbereich bezeichnet, die nicht von der symbolischen Repräsentation eines anderen eingefügten Objekts überlagert wird.

An dieser Stelle sei erwähnt, daß auch Instanzen von komplexen Typen vollständige Kopien ihrer Vorlage sind, d.h. es wird die gesamte Objektstruktur einschließlich aller enthaltenen Objekte vollständig kopiert.

4.4.2 Definition von Verbindungen

Durch das Einfügen von komplexen Objekten in komplexe Objekte entstehen Objekthierarchien. Die folgende Abbildung spiegelt beispielhaft den Aufbau einer solchen Hierarchie wieder:



Dabei befinden sich die einzelnen Objekte auf unterschiedlichen Hierarchieebenen. Das Objekt COMPLEXA liegt auf der 1. Hierarchieebene und enthält die Objekte BASEA, COMPLEXB und COMPLEXC, die auf der 2. Hierarchieebene liegen. COMPLEXB und COMPLEXC enthalten wiederum BASEB, BASEC und BASED auf der 3. Hierarchieebene.

Zwischen jeweils zwei eingefügten Basisobjekten kann durch ein Verbindungsobjekt eine Verbindung hergestellt werden. Dabei können sich die Basisobjekte auf unterschiedlichen Ebenen einer Objekthierarchie, innerhalb verschiedener komplexer Objekte befinden. Zur Definition einer Verbindung muß zunächst ein gemeinsamer Vorfahre (komplexes Objekt) der zu verbindenden Basisobjekte ermittelt werden. Ein gemeinsamer Vorfahre enthält beide Basisobjekte in dem von ihm aufgespannten Teilbaum. Im Grafikbereich dieses Vorfahrens wird nun über das Popup-Menü (linke Maustaste an einer freien Position) eine Instanz des vorher ausgewählten Verbindungsobjekt-Typs durch Auswahl von *Connection* erzeugt. Jetzt kann das Startobjekt (Basisobjekt) ausgewählt werden. Dies geschieht durch Anklicken der entsprechenden symbolischen Repräsentation mit der Maus. Falls das gewünschte Objekt auf einer tieferen Hierarchiestufe liegt, wird zunächst die symbolische Repräsentation des komplexen Objekts angeklickt, dessen Teilbaum das Objekt enthält. Dadurch öffnet sich das Eingabeformular dieses Objekts. Der Vorgang wird nun solange wiederholt, bis die symbolische Repräsentation des gewünschten Basisobjekts sichtbar wird. Nachdem das

Startobjekt ausgewählt wurde, kann ein Linienzug zum Zielobjekt definiert werden. Dabei können in den Grafikbereichen der komplexen Objekte, die im Hierarchiebaum auf dem Weg vom Start- zum Zielobjekt liegen, beliebig viele Koordinaten festgelegt werden, die den Linienzug definieren. Das Festlegen der Koordinaten innerhalb eines Grafikbereiches geschieht, indem an freien Positionen die linke Maustaste gedrückt wird. Nachdem der Linienzug innerhalb eines Grafikbereiches definiert wurde, kann er auf einer höheren Hierarchieebene durch das Schließen des entsprechenden Eingabeformulars (*OK*) fortgesetzt werden. Um auf einer tieferen Ebene fortzufahren, muß die symbolische Repräsentation des betreffenden komplexen Objekts ausgewählt werden, so daß sich das entsprechende Eingabeformular öffnet. Zum Abschluß der Definition wird das Zielobjekt ausgewählt. Dann werden alle Eingabeformulare der komplexen Objekte geschlossen (*OK*), die in der Objekthierarchie unter dem gemeinsamen Vorfahren der verbundenen Basisobjekte liegen. Dabei wird der Linienzug um zusätzliche Verbindungskoordinaten erweitert, die den Verlauf des Linienzugs innerhalb der Grafikbereiche bis zum Rand vervollständigen, wobei der Ein- bzw. Austrittswinkel des Linienzugs auf der nächst höheren Ebene in die zugehörige grafische Repräsentation berücksichtigt wird. Damit ist die Verbindung zwischen den beiden Basisobjekten vollständig definiert und wird in die Objekthierarchie des gemeinsamen Vorfahren (komplexes Objekt) aufgenommen.

4.4.3 Kopieren, Löschen und Ändern

Innerhalb des Grafikbereichs eines komplexen Objekts können von allen eingefügten Basisobjekten und komplexen Objekten beliebig viele Kopien erzeugt und eingefügt werden. Beim Kopieren werden die Daten der Vorlage vollständig in die Kopie übernommen.

Zu jedem in einem Grafikbereich symbolisch dargestellten Basisobjekt und komplexen Objekt kann ein spezielles Popup-Menü aufgerufen werden. Dazu wird die linke Maustaste über dem Icon der symbolischen Repräsentation gedrückt. Das Menü zeigt eine Auswahl aller für das Objekt gerade möglichen Funktionen. Durch die Auswahl von *Copy* wird ein Objekt kopiert. Die symbolische Repräsentation der Kopie wird anschließend unmittelbar neben der Kopievorlage dargestellt.

Auch eingefügte Verbindungsobjekte können kopiert werden, allerdings müssen die Kopien anschließend in einen Verbindungskontext, wie im vorigen Abschnitt beschrieben, eingefügt werden. D.h. die Informationen über die Verbindung zweier Basisobjekte werden nicht in die Kopie übernommen. Zu Verbindungsobjekten wird das entsprechende Popup-Menü durch Anklicken der zum Linienzug gehörenden ***Grips*** abgebildet. Grips, sind kleine ausgefüllte rechteckige Kästchen, die an den Positionen der Verbindungskoordinaten dargestellt werden.

Alle eingefügten Objekte können jederzeit verändert werden. Dazu dient *Edit* im Popup-Menü des betreffenden Objekts (Doppelklick auf die Beschriftung der jeweiligen symbolischen Darstellung hat dieselbe Wirkung). Anschließend öffnet sich das zugehörige Eingabeformular, in dem alle für die einzelnen Objektklassen beschriebenen Daten geändert werden können. Im Eingabeformular von komplexen Objekten können dabei zu den enthaltenen Objekten ebenfalls die Popup-Menüs aufgerufen werden, so daß auch diese Objekte verändert werden können. Dadurch sind auch die auf tieferen Hierarchieebenen liegenden Objekte erreichbar und können jederzeit modifiziert werden. Beim Aufbau eines Simulationssystems besteht häufig der Wunsch, die Eingabeformulare einiger Objekte dauerhaft auf dem Bildschirm abgebildet zu lassen, ohne sie vor jedem neuen Compilierungsvorgang schließen zu müssen. Aus diesem Grund werden die in den Eingabeformularen gemachten Änderungen bereits übernommen, wenn der Mauszeiger das entsprechende Formular verläßt, also auch ohne *OK* aufzurufen. Wird ein Eingabeformular zu einem späteren Zeitpunkt durch *Cancel* geschlossen, gehen alle seit dem Öffnen gemachten Änderungen verloren, auch wenn man zwischenzeitlich das Formular „verlassen“ hat.

Mit *Delete* im Popup-Menü eines Objekts kann dieses Objekt aus einer Objektstruktur gelöscht werden. Das Löschen ist allerdings nur dann möglich, wenn das entsprechende Objekt nicht gerade geöffnet ist, d.h. sein Eingabeformular nicht auf dem Bildschirm abgebildet ist.

4.4.4 Umwandeln von Instanzen in Typen

Eingefügte Instanzen von Basisobjekten, komplexen Objekten und Verbindungsobjekten können in einen Typ umgewandelt und der jeweiligen Typauswahl hinzugefügt werden. Dies geschieht über den Punkt *CopyToType* im Popup-Menü der Objekte. Diese Funktion erweist sich als sehr nützlich, falls der Benutzer an einer Instanz eines Typs aufwendige Änderungen vorgenommen hat und diese Instanz nun als Vorlage zur Erzeugung weiterer Instanzen in anderen komplexen Objekten oder späteren Modellen benutzen möchte.

4.4.5 Erzeugen von Child-Objekten

Für Basisobjekte wird in den Popup-Menüs eine besondere Funktion angeboten: Das Erzeugen von ***Childs***. Childs sind eigentlich nichts anderes als Referenzen auf die Basisobjekte (Vater-Objekte), über deren Popup-Menüs sie erzeugt wurden. Lancelot ordnet den Childs den Namen ihres Vater-Objekts, konkateniert mit einem fortlaufenden Index ([]), zu. Da es sich um Referenzen handelt, ist es nicht sinnvoll ihre Daten zu verändern. Veränderungen können nur im zugehörigen Vater-Objekt vorgenommen werden und wirken sich dann auf alle davon erzeugten Childs aus. Wie später erläutert wird, kann im Code des Vater-Objekts, das dynamische Verhalten einzelner Childs anhand der zugeordneten Indizes getrennt festgelegt werden. Aus diesem Grund läßt sich bei Childs nur die grafische Repräsentation (Icon) verändern, um damit einzelne Childs besonders hervorheben zu können.

Von den Childs eines Vater-Objekts kann immer nur das Child mit dem höchsten Index gelöscht werden. Beim Löschen eines Vater-Objekts werden auch alle zugehörigen Childs gelöscht.

4.4.6 Zusammenfassen von Objektstrukturen

Aus den in komplexen Objekten oder im Root-Formular dargestellten Objektstrukturen können Teilstrukturen zu neuen komplexen Objekttypen zusammengefaßt werden. Dazu wird die entsprechende Teilstruktur mit Hilfe eines Clip-Rechtecks innerhalb der dargestellten Objektstruktur markiert. Der Anfangspunkt des Rechtecks wird gesetzt, indem die mittlere Maustaste an einer freien Position in der Drawing-Area gedrückt wird. Nun kann der Mauszeiger bei gedrückter Maustaste zum Zielpunkt des Rechtecks verschoben werden. Der Zielpunkt wird nach dem Loslassen der Maustaste bestimmt. Alle innerhalb des Rechtecks liegenden Objekte werden dabei in einen neuen komplexen Objekttyp kopiert. Der neue Typ wird anschließend im entsprechenden Typbereich des Typ-Editors dargestellt. Dabei erhält er den gleichen Namen, wie die Vorlage der Objektstrukturen.

4.4.7 Verschieben von eingefügten Objekten

Die symbolischen Repräsentationen eingefügter Basisobjekte und komplexer Objekte können innerhalb eines Grafikbereiches an eine andere Bildschirmposition verschoben werden. Dazu wird die rechte Maustaste im Bereich des Namens der jeweiligen Repräsentation gedrückt gehalten und der Mauszeiger dann an die neue Position verschoben. Auf ähnliche Weise kann auch der Verlauf eines Linienzuges nachträglich verändert werden. Dies geschieht wie gerade beschrieben, nur das in diesem Fall die rechte Maustaste innerhalb eines Grips des Linienzugs gedrückt gehalten wird.

4.4.8 Der Code-Bereich eines komplexen Objekts

Für eine mögliche Erweiterung von Lancelot (z.B. bigMC) wurden die komplexen Objekte um einen Code-Bereich erweitert. Dieser Bereich kann beispielsweise zur Definition von lokalen Variablen benutzt werden, die von einem neuen Modellcompiler in ein erstelltes Simulationsprogramm übernommen werden. Das Öffnen des Formulars für den lokalen Code-

Bereich geschieht über den Button *Code*. Da der bisherige Modellcompiler Xcalibur diese zusätzliche Information in den Modelldaten nicht interpretieren kann, gibt es das Flag *Xartus*extended* in den Ressourcen von Xartus, das ein Umschalten zwischen der Version mit und ohne lokalen Code-Bereich ermöglicht. Es gilt die folgende Belegung:

*Xartus*extended: 0:* Es wird kein lokaler Code-Bereich zu den komplexen Objekten angezeigt, auch die Modelldaten werden ohne lokalen Code übertragen.

*Xartus*extended: 1:* Jedes komplexe Objekt besitzt einen lokalen Code-Bereich und den zugehörigen Button *Code*, auch die Modelldaten werden nun um den lokalen Code erweitert.

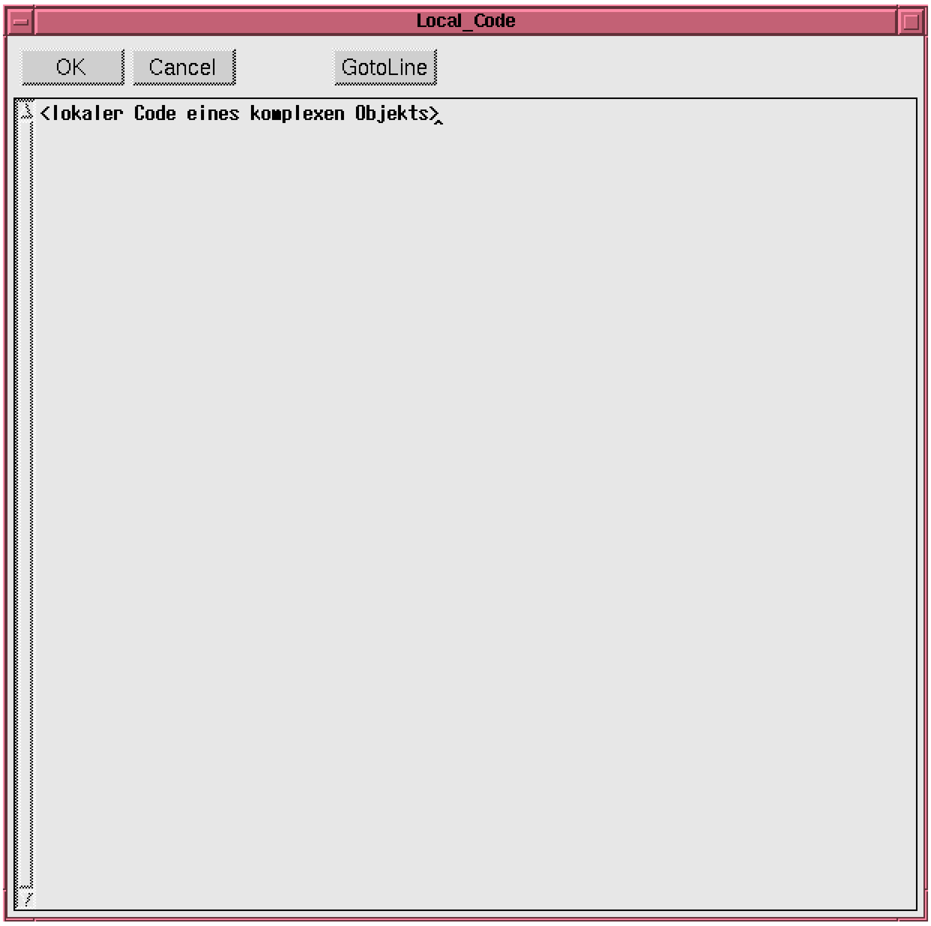


Abbildung 9: Lokaler Code-Bereich eines komplexen Objekts

4.5 Definition von Typen für Iconobjekte

Wie bereits erwähnt wurde, dienen die Iconobjekte zur Visualisierung des dynamischen Verhaltens eines Simulationsmodells. Zu diesem Zweck kann die grafische Repräsentation dieser Objekte über den Linienzug eines Verbindungsobjekts zwischen zwei Basisobjekten bewegt werden.

Ein neuer Typ für Iconobjekte wird mit *New* und anschließend *Icon* im Menübereich des Typ-Editors erzeugt. Das dadurch erzeugte Eingabeformular besitzt das in Abbildung 10 gezeigte Aussehen.

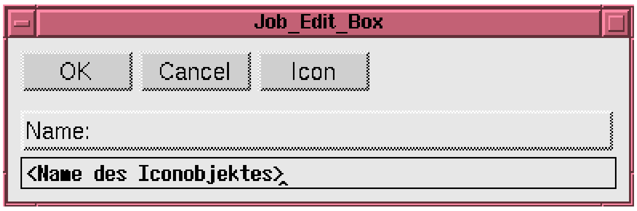


Abbildung 10: Eingabeformular der Iconobjekte

Zur Definition des Typs muß ein Name in den entsprechenden Eingabebereich eingesetzt und eine geeignete grafische Repräsentation über *Icon* ausgewählt werden. *Icon* führt dabei in das Formular des Icon-Managers. *OK* und *Cancel* im Eingabeformular besitzen wieder die gewohnte Funktionalität. Nach der Definition wird der neue Typ im entsprechenden Auswahlbereich des Typ-Editors dargestellt.

Anders als bei den bisher beschriebenen Objektklassen, können Instanzen dieser Typen nur im Formular des Root-Objekts erzeugt und eingefügt werden. Nähere Informationen dazu folgen im Abschnitt „Aufbau des Root-Objekts“.

4.6 Definition von Typen für Typobjekte

Die Typobjekte können bei Lancelot zur Definition von größeren Datenstrukturen verwendet werden. In C-SIM gehören dazu beispielsweise Prozeßtypen, der globale Jobtyp und der globale Queuetyp. Zu diesem Zweck werden Instanzen von definierten Typen dieser Objektklasse dem Root-Objekt zugeordnet (siehe Abschnitt „Aufbau des Root-Objekts“). Dort dienen sie dann entweder als globale Typvorlagen oder können im Code anderer Objekte referenziert werden, um unnötige Neudeklarationen dieser Datenstrukturen zu vermeiden.

Durch *New* und *Type* im Menübereich des Typ-Editors wird ein neuer Typ dieser Objektklasse erzeugt und das zugehörige Eingabeformular auf dem Bildschirm abgebildet (Abbildung 11).

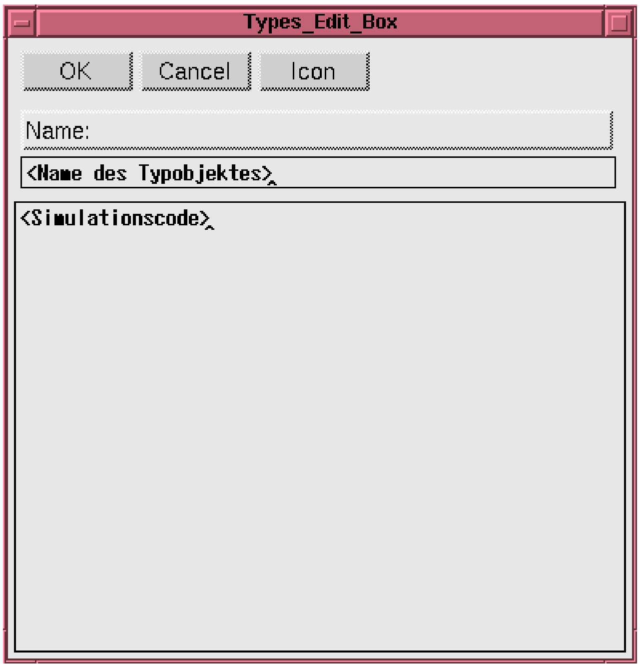


Abbildung 11: Eingabeformular der Typobjekte

Das Formular besitzt einen Eingabebereich für den Namen des Typobjekts und einen für die Definition der Datenstrukturen. Über *Icon* kann dem Typ in gewohnter Weise eine grafische Repräsentation zugeordnet werden. *OK* beendet die Definition und übernimmt den neuen Typ in den Typ-Editor.

4.7 Ändern, Löschen, Speichern und Laden der Typen

Der Typ-Editor ermöglicht das getrennte Laden und Speichern aller Typen der einzelnen Objekt-Klassen über die Button *Load* und *Save* im Menübereich des Typ-Editors. Zum Speichern wird dabei ein Name für die Datei verlangt, in der die Typen der gewählten Klasse abgelegt werden. Für die einzelnen Objektklassen wird dieser Name um eine bestimmte Endung erweitert:

<Dateiname>.bta	für Basisobjekte
<Dateiname>.cta	für Verbindungsobjekte
<Dateiname>.cpa	für komplexe Objekte
<Dateiname>.job	für Iconobjekte
<Dateiname>.joc	für Typobjekte

Beim Laden von Objekttypen (*Load*) werden dann alle Dateien mit der entsprechenden Endung zur Auswahl angezeigt. Es ist auch möglich mehrere Dateien mit gespeicherten Typen für eine Objektklasse zu laden. Die Typen einer neu geladenen Datei werden dabei den bereits existierenden Typen der Klasse hinzugefügt und im entsprechenden Auswahlbereich dargestellt.

Ähnlich, wie eingefügte Instanzen von Typen innerhalb von komplexen Objekten, können auch Typen selbst nachträglich verändert werden. Dazu kann für jeden dargestellten Typ ein Popup-Menü durch Betätigung des linken Mausknopfs über dem Iconbereich der symbolischen Darstellung erzeugt werden. Durch Auswahl des Menüpunkts *Edit* wird das zugehörige Eingabeformular erzeugt. Innerhalb dieses Formulars können dann alle zu dem gewählten Typ gehörenden Daten, einschließlich der grafischen Repräsentation, verändert werden. Veränderungen eines Typs wirken sich nur auf die nachfolgend erzeugten Instanzen des Typs aus. Alle vorher erzeugten Instanzen bleiben von den Veränderungen unberührt.

Die innerhalb eines Typ-Bereichs dargestellten Typen einer Objektklasse können einzeln gelöscht werden. Dabei wird die symbolische Repräsentation des Typs aus dem jeweiligen Bereich entfernt. Alle vorher erzeugten Instanzen bleiben weiterhin bestehen und von dem Löschen ihrer Vorlage unbeeinflußt. Das Löschen eines Typs geschieht durch Aufruf des Popup-Menüs (linke Maustaste über dem Iconbereich) und die Auswahl von *Delete*.

Damit sind nun alle wesentlichen Grundlagen bekannt, um Typen der einzelnen Objektklassen definieren zu können. Deshalb wird nun der eigentliche Modellaufbau beschrieben.

4.8 Aufbau des Root-Objekts

Die Definition geeigneter Typen ist eine notwendige Voraussetzung zum Aufbau eines Simulationsmodells. Das Modell wird dann innerhalb des **Root-Objekts** aufgebaut, das eine erweiterte Funktionalität der komplexen Objekte besitzt. Dementsprechend können aus Basisobjekten, komplexen Objekten und Verbindungsobjekten bestehende Objektstrukturen in das Root-Objekt eingefügt werden. Darüber hinaus besitzt das Root-Objekt allerdings eine Funktionalität, die weit über die der komplexen Objekte hinausgeht.

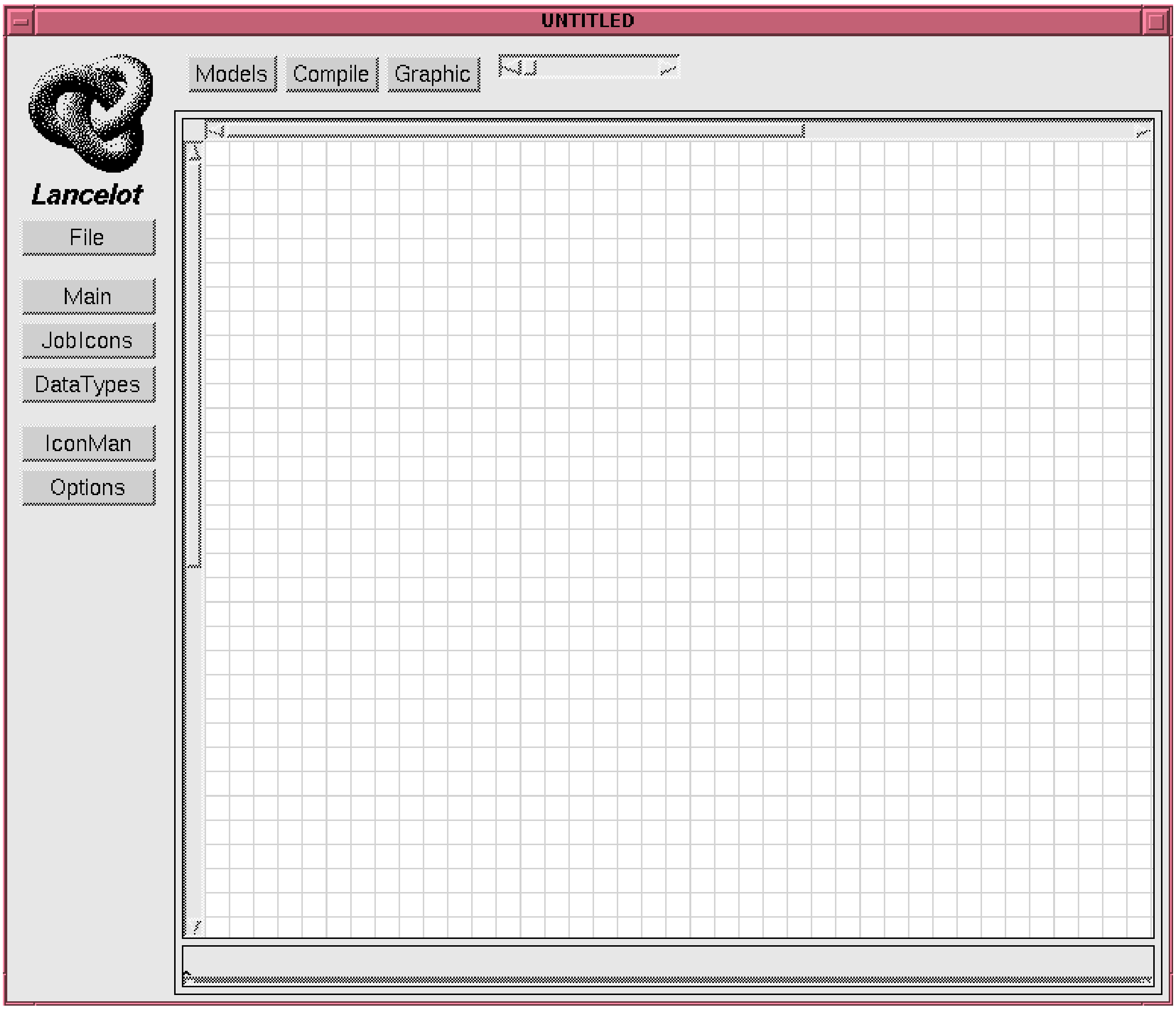


Abbildung 12: Root-Formular

Nach dem Start von Lancelot werden zwei Windows auf dem Bildschirm erzeugt. Das erste Window gehört zum bereits beschriebenen Typ-Editor und das Zweite zum Root-Formular, dem Eingabeformular des Root-Objekts (siehe Abbildung 12). Das Root-Formular besteht aus vier Eingabebereichen. Die nun der Reihe nach beschrieben werden.

4.8.1 Der Grafikbereich des Root-Formulars

Dieser Eingabebereich besitzt eine fast identische Funktionalität, wie der Grafikbereich des Eingabeformulars der komplexen Objekte. Es können Instanzen von Basisobjekten, Verbindungsobjekten und komplexen Objekte auf die gleiche Weise erzeugt und eingefügt werden. Die erzeugten Objektstrukturen im Root-Formular sind Teil des eigentlichen Simulationsmodells, daß später die Eingabe des Modellcompilers darstellt. Dies ist ein

Unterschied zu den bisher beschriebenen Typen im Typ-Editor, die für den Compiler nicht sichtbar sind.

Der Grafikbereich des Root-Formulars wird von zwei Scrollbars umrahmt. Diese ermöglichen die Ansicht von erzeugten Objektstrukturen, die außerhalb des begrenzten Bildschirmausschnitts liegen.

4.8.2 Der Modellbereich des Root-Formulars

Der Modellbereich befindet sich links neben dem Grafikbereich und enthält diverse Button zum Modellaufbau.

4.8.2.1 Modelle speichern laden und löschen, Lancelot beenden

Nach Auswahl des Button *File* erscheint ein Menü auf dem Bildschirm, das verschiedene Möglichkeiten zum Laden, Abspeichern und Löschen eines Modells anbietet, die im folgenden kurz beschrieben werden:

Save: Speichert ein geladenes Modell im Root-Objekt unter dem im Formular angezeigten Namen auf einem permanenten Datenträger.

Save as: Durch diese Auswahl kann das zur Zeit im Root-Objekt gespeicherte Modell auf einem Datenträger permanent gespeichert werden. Dazu wird der Benutzer um die Eingabe eines Namens gebeten, der dann sowohl den Namen des Modells festlegt als auch zur Ablage in einer Datei mit diesem Namen, erweitert um die Endung ***.rot***, benutzt wird.

Load: Durch diese Auswahl wird ein abgespeichertes Modell in das Root-Objekt geladen. Ein möglicherweise bestehendes Modell wird dabei gelöscht. Vorher wird um die Auswahl der entsprechenden Datei mit den zu ladenden Modelldaten gebeten. Diese kann entweder aus einer Liste von Dateien mit der Endung ***.rot*** ausgewählt oder per Hand eingegeben werden.

Save all: Durch Auswahl von *Save all* werden sowohl die Modelldaten als auch alle definierten Typen im Typ-Editor abgespeichert. Dabei werden getrennte Dateien für die Modelldaten und die Typen der einzelnen Objektklassen angelegt. Auf diese Weise können die abgelegten Daten zu einem späteren Zeitpunkt wieder getrennt geladen werden (*Load* im Root-Formular / *Load* in den Typ-Bereichen). Der eingegebene Dateiname wird dementsprechend um die zugehörenden Endungen erweitert.

Load all: Diese Auswahl ist das Gegenstück zu *Save all*, d.h. es werden die zu einem Modellnamen gehörenden Modelldaten und Typen geladen. Die geladenen Typen werden dabei zu den bereits bestehenden hinzugefügt.

Clear all: Durch diese Auswahl werden alle im Root-Objekt enthaltenen Daten gelöscht. Anschließend kann mit dem Aufbau eines neuen Modells begonnen werden.

Außerdem enthält das Menü den Button *Quit*. Mit *Quit* wird der Ablauf von Lancelot beendet. Nach einer Warnmeldung gehen dabei alle nicht gesicherten Daten verloren.

4.8.2.2 Festlegen des Main-Code

Durch die Auswahl von *Code* im Modellbereich wird ein Eingabeformular (Abbildung 13) geöffnet, in dem die Definition globaler Variablen und die Festlegung eines Hauptprogramms (main-Code) des Simulationsmodells möglich ist. Mit *OK* können diese Eingaben bestätigt und das Formular geschlossen werden. *Cancel* führt zum Abbruch der Eingabe, alle bis zu diesem Zeitpunkt in das Formular eingegebenen Daten gehen dabei verloren.

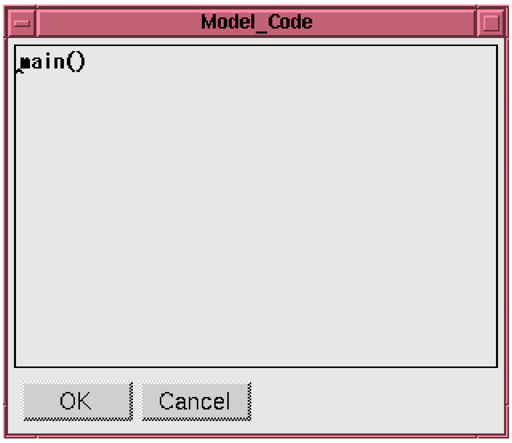


Abbildung 13: Eingabeformular für den Main-Code

4.8.2.3 Zuordnen von Types

Wie bereits beschrieben wurde, ermöglicht Lancelot die Definition größerer Datenstrukturen mit Hilfe von Typobjekten. Damit diese Objekte innerhalb des Modells (Root-Objekts) verfügbar sind und referenziert werden können, müssen Instanzen vordefinierter Typen dieser Objektklasse in die Modelldaten eingefügt werden. Dies geschieht innerhalb eines Formulars (Abbildung 14), das durch Auswahl von *Types* im Modellbereich auf dem Bildschirm geöffnet wird.

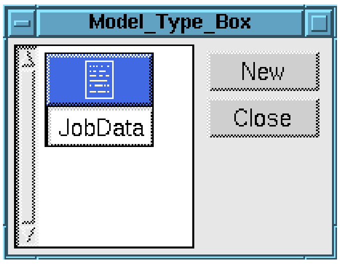


Abbildung 14: Formular für die Zuordnung
von Typobjekten zum Root-Objekt

Innerhalb dieses Formulars kann mit *New* eine Instanz eines im Typ-Editor ausgewählten Objekts erzeugt und in das Root-Objekt eingefügt werden. Diese Instanz wird anschließend im Darstellungsbereich der eingefügten Typen symbolisch angezeigt. Über das Popup-Menü und die Auswahl von *Edit* bzw. *Delete* kann die Instanz in gewohnter Weise verändert bzw. gelöscht werden.

4.8.2.4 Zuordnen von Iconobjekten

Zur Darstellung eines Simulationsablaufs werden bei Lancelot Animationen von Iconobjekten eingesetzt. Dazu müssen Instanzen vordefinierter Typen von Iconobjekten erzeugt und in das Root-Objekt eingefügt werden. Dies geschieht innerhalb eines Formulars (Abbildung 15), das durch Auswahl von *Icons* im Modellbereich auf dem Bildschirm abgebildet wird. Das Formular besitzt den gleichen Aufbau und die gleiche Funktionalität, wie das für die Typobjekte vorgestellte Formular.

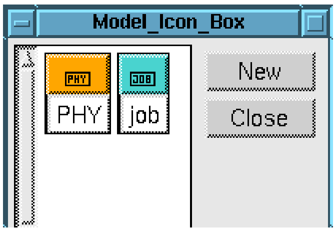


Abbildung 15: Formular für die Zuordnung
von Iconobjekten

4.8.2.5 Der Icon-Manager

Alle Objekte besitzen innerhalb von Lancelot eine symbolische Repräsentation. Bei den meisten Objekten besteht diese Repräsentation aus dem Objektnamen und einem zugeordneten Icon. Der Icon-Manager, der über *IconMan* aufgerufen wird, ermöglicht die

Erzeugung neuer und die Verwaltung der existierenden Icons. Das Formular des Icon-Managers (Abbildung 16) enthält einen Darstellungsbereich, in dem alle momentan verfügbaren Icons angezeigt werden, und mehrere Button zur Verwaltung.

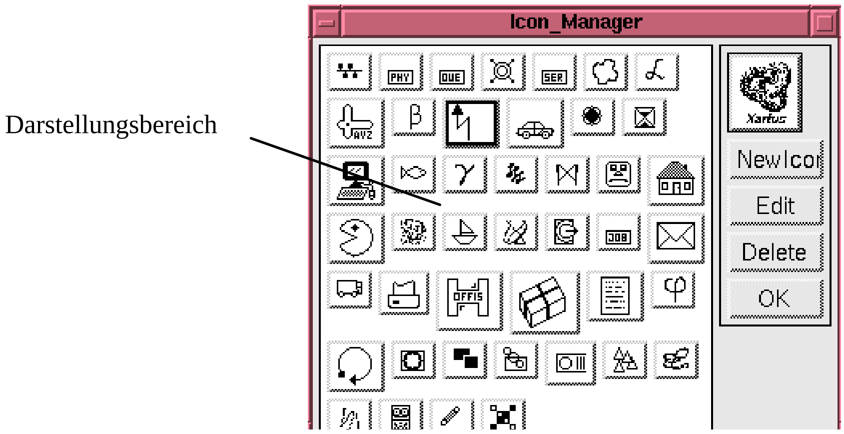


Abbildung 16: Formular des Iconmanagers

Mit *New* wird ein Icon-Editor aufgerufen, der die Konstruktion neuer Icons ermöglicht. Als Editor wird standardmäßig **bitmap** aufgerufen, der normalerweise auf jedem X11-System vorhanden ist, und auf jeden Fall im Basissuchpfad des Benutzers liegen muß. Die erzeugten Icons werden beim Abspeichern in dem Verzeichnis **Lancelot/Icons** abgelegt. Hier werden sie von Lancelot gelesen und stehen damit zur weiteren Verwendung zur Verfügung. Damit Lancelot die Icons erzeugen kann, müssen die entsprechenden Dateien im X11-Bitmap-Format gespeichert werden (wird von **bitmap** standardmäßig benutzt). Prinzipiell läßt sich über die Ressourcen von Lancelot (siehe Anhang) auch ein anderer Icon-Editor festlegen. Es muß dann aber darauf geachtet werden, daß die erzeugten Icons im entsprechenden Verzeichnis und im richtigen Format abgelegt werden. Mit *Delete* lassen sich dargestellte Icons aus der Auswahl löschen, falls sie nicht gerade innerhalb von Lancelot von irgendwelchen Objekten verwendet werden. *Edit* ermöglicht das Verändern erzeugter Icons

4.8.2.6 Einstellen von Optionen

Lancelot ermöglicht über *Options* im Modellbereich des Root-Formulars die Einstellung verschiedener Optionen.

4.8.2.6.1 Auswahl von Faben für die Applikation

Eine Applikation hat die Möglichkeit, die Farbe, mit der die symbolische Repräsentation eines Objekts auf dem Bildschirm dargestellt wird, zu verändern. Dadurch kann dann z.B. ein veränderter Zustand angezeigt werden. Xcalibur verwendet diese Möglichkeit unter anderem dazu, passivierte und aktivierte Objekte auf dem Bildschirm zu kennzeichnen. Diese Farben können in einem Auswahlformular festgelgt werden, das über *Options/Colors* geöffnet wird. Zur Definition einer Farbe muß zunächst eine Farbe in der Palette (rechte Seite) ausgewählt werden. Diese wird dann durch die Auswahl eines der auf der linken Seite dargestellten Felder als Applikationsfarbe festgelegt. Die beiden oberen Felder sind dabei ausschließlich für die Farben Weiß und Schwarz definiert und können nicht neu belegt werden.

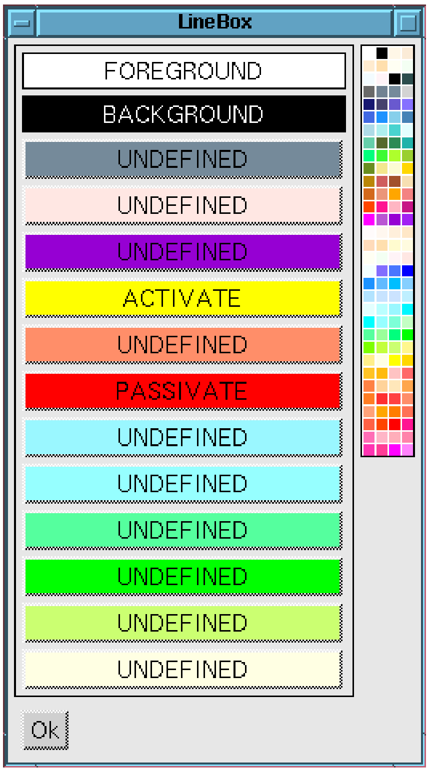


Abbildung 17: Festlegen von applikationsspezifischen Farben

Die Applikation kann die Farben entweder über die Funktion `Xs_ColorObject` (siehe Abschnitt „Anbindung einer Applikation“) oder über die Canvas-Funktionen (siehe Abschnitt „Erweiterte Ausgabemöglichkeiten“) einsetzen. Beim Aufruf der entsprechenden Funktionen wird die Angabe eines Farbindex gefordert. Dafür kann ein Wert zwischen 1 und 14 eingesetzt werden. Diese Werte entsprechen der Reihe nach den Farbfeldern in der oben gezeigten Auswahlbox.

Außerdem wird eine Applikation zur Anzeige eines bestimmten Objektzustands immer die gleiche Farbe verwenden. Um eine leichtere Zuordnung der Farben zu ermöglichen können die oben dargestellten Farbfelder über die Ressourcen (siehe Anhang) entsprechend beschriftet werden. So verwendet Xcalibur zur Darstellung eines passivierten Objekts immer den Farbwert 8. Das achte Feld in der obigen Darstellung enthält daher die Beschriftung PASSIVATE. Die Festlegung der Beschriftung des Feldes mit der Nummer i geschieht über die Ressource `Xartus*appcolori`. Die von einer Applikation verwendeten Farbwerte lassen sich somit zur Laufzeit dynamisch verändern. Beim Verlassen des Programms werden die ausgewählten Farben gespeichert.

4.8.2.6.2 Einstellen der Gitternetzdichte

Der Grafikbereich der komplexen Objekte und des Root-Objekts enthalten ein Gitternetz, um eine geordnete Plazierung der enthaltenen Objekte zu ermöglichen. Die Dichte dieses Gitternetzes kann über *Options/Grid* eingestellt werden. Dazu ist lediglich das Eintragen eines entsprechenden positiven Wertes (≤ 40) in die unten dargestellte Dialogbox erforderlich.

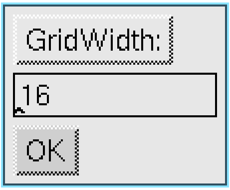


Abbildung 18: Einstellen der Gitternetzdichte

4.8.2.6.3 Einstellen der Animationsgeschwindigkeit

Wie im folgenden Abschnitt „Steuerbereich des Root-Formulars" noch beschrieben wird, kann die Animationsgeschwindigkeit über einen entsprechenden Schieberegler innerhalb des Root-Formulars eingestellt werden. Dennoch kann es passieren, daß der einstellbare Bereich nicht den Anforderungen des Benutzers entspricht. So wird die Animation auf unterschiedlich schnellen Rechnern, bei gleicher Stellung des Reglers, auch unterschiedlich schnell ablaufen. Um diese Geschwindigkeit dennoch anpassen zu können, gibt es über *Options/Speed* die Möglichkeit, einen Wert für eine Warteschleife anzugeben.

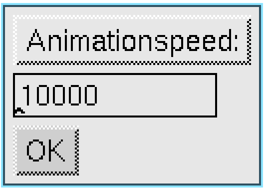


Abbildung 19: Einstellen der Animationsgeschwindigkeit

Ein in der oben dargestellten Dialogbox eingetragener Wert z bewirkt, daß zwischen jedem Animationsschritt eine Schleife von 0 bis z ausgeführt wird. Durch einen höheren z-Wert wird die Animation also langsamer ablaufen.

4.8.2.6.4 Laden eines Hintergrundbildes

Im Root-Formular kann ein Hintergrundbild dargestellt werden. Das Hintergrundbild kann dabei entweder über die Ressource `*Xartus*backpixmap` (XRes) oder über eine Dialogbox, die über *`Options/BackPixmap`* geöffnet wird, festgelegt werden. In beiden Fällen ist der komplette Pfad zu der entsprecheneden X11-Bitmap-Datei anzugeben. Um <u>kein</u> Hintergrundbild anzuzeigen müssen die entsprechenden Einträge gelöscht werden.

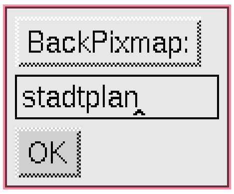


Abbildung 20: Festlegen eines Hintergrundbildes
mit dem Namen "stadtplan"

4.8.3 Der Menübereich des Root-Formulars

An der oberen Kante des Root-Formulars befindet sich der Menübereich. Innerhalb der angebotenen Menüs können verschiedene Funktionen aufgerufen und Einstellungen vorgenommen werden. Die möglichen Funktionen und Einstellungen beziehen sich dabei auf den Compiler und den Simulationsablauf und werden nun kurz erläutert:

Model:

RunSimulation: Startet den Simulationsprozeß mit dem Namen des geladenen Modells (wird in der Titelleiste des Root-Formulars angezeigt). Falls das Modell noch nicht compilert wurde, erfolgt eine entsprechende Fehlermeldung.

StopSimulation: Beendet einen laufenden Simulationsprozeß während seiner Ausführung. Das dargestellte Modell auf dem Bildschirm wird zurückgesetzt; d.h. alle Objekte werden in ihren Ausgangszustand versetzt (Markierungen werden zurückgenommen).

CutSimulation: Ein laufender Simulationsprozeß wird von Lancelot „abgekoppelt“. Anschließend besteht keinerlei Verbindung mehr zwischen Lancelot und dem Simulationsprozeß. Folglich bleiben Grafikaufrufe des Simulationsprozesses und Steuersignale von Lancelot ohne Wirkung. Falls der Simulationsprozeß nicht von sich aus terminiert muß er auf Betriebssystemebene deaktiviert werden (z.B. mit *kill* ...). Durch die Trennung von der Benutzungsoberfläche kann die Ablaufgeschwindigkeit des Simulationsprozesses erheblich gesteigert werden.

Compiler:

Compile: Startet die Übersetzung des Simulationsmodells in ablauffähigen Maschinencode. Die Übersetzung geschieht dabei in zwei Phasen: Zunächst wird das Modell in C-SIM-Code transformiert. Anschließend erfolgt dann die Übersetzung durch einen C-Compiler. Der Abschluß der Übersetzung wird durch eine entsprechende Meldung angezeigt. Der ausführbare Maschinencode wird nach der Übersetzung unter dem Modellnamen (ohne Endung) in dem Verzeichnis *Lancelot/bin/* abgelegt.

Options: Die Aktivierung bzw. Deaktivierung der nachfolgenden Menüfunktionen erfordert eine erneute Compilation des Simulationsprogramms, d.h. die Änderung einer der Optionen wird erst wirksam, nachdem der Compiler erneut aufgerufen worden ist. Eine Deaktivierung der nachfolgenden Menüpunkte bewirkt, daß der Compiler die Generierung bestimmter Funktionsaufrufe im Simulationsprogramm unterdrückt. Der Vorteil liegt dabei in einer erheblichen Beschleunigung des Simulationsprozesses.

Graphic:

Animation: Durch Deaktivierung dieser Funktionen werden vom Compiler keine Animationsfunktionen in den Simulationscode eingefügt.

Mark Object: Der momentane Zustand (Status) eines Prozesses wird durch Farbveränderung seines Iconobjekts grafisch dargestellt. Die Deaktivierung dieses Menüpunktes bewirkt, daß sämtliche Markierungsfunktionen vom Compiler unterdrückt werden.

Mark Act. Proc. (mark active process): Ist diese Funktion eingeschaltet, wird der momentan aktivierte Prozeß, d.h. der Prozeß, dessen Phase derzeit ausgeführt wird, markiert. Das Deaktivieren dieser Funktion veranlaßt den Compiler, die Markierungsfunktionen nicht im Simulationscode einzufügen.

Systemtime: Eine Deaktivierung bewirkt, daß die Funktion zur Ausgabe der Simulationszeit im Root-Formular nicht in den Simulationscode aufgenommen wird.

Parameter: Die Aktivierung eines Parameters, löst die Generierung der Preprozessoranweisung `#define <Paramtername>` im Simulationscode aus. Die Bedeutung der verschiedenen C-SIM Parameter ist in [Block95] beschrieben. Das automatische Anlegen von Stationen durch den Compiler wird bei ausgeschaltetem Menüpunkt *Station* unterdrückt. Dies geschieht, um Inkonsistenzen auszuschließen.

autom. Station: Ist diese Funktion ausgeschaltet, werden die vom System angelegten Stationen bei Verwendung von `Insert`-, `Seize`-, `Enter`- und `Await`-Aufrufen im Phasencode nicht automatisch aktualisiert.

Warnings: Werden im Phasencode Prozesse referenziert, zwischen denen keine Verbindung im Root-Formular existiert, gibt der Compiler eine Warnungsmeldung aus. Bleibt dieser Menüpunkt ausgeschaltet, werden die Warnungsmeldungen unterdrückt.

Graphic:

Animation: Während eines Simulationsablaufs können mit dieser Funktion die ausgelösten Animationen unterdrückt werden. Dadurch wird der Ablauf des Prozesses erheblich beschleunigt.

Highlight: Mit dieser Funktion können die während eines Simulationsablaufs aufgerufenen Markierungen der Objekte abgeschaltet werden.

SimDisplay: Mit Hilfe dieser Funktion können alle Ausgaben eines laufenden Simulationsprozesses in das Simulations-Display (s.u.) unterdrückt werden.

4.8.4 Der Steuerbereich des Root-Objekts

Rechts neben dem Menübereich im Root-Formular befindet sich der Steuerbereich. Er besteht z.Zt. nur aus einem Scrollbar (Speedbar), mit dem die Geschwindigkeit des

Simulationsablaufs gesteuert werden kann. Dabei erhöht sich die Ablaufgeschwindigkeit, je weiter der Scrollbar nach rechts verschoben wird.

4.8.5 Das Simulations-Display

Rechts neben dem Steuerbereich befindet sich das Simulations-Display. Es unterteilt sich in zwei Felder, in die vom Simulationsprozeß bestimmte Meldungen ausgegeben werden können. Das erste Feld enthält in der derzeitigen Version von Lancelot die Zeichenkette „SimulationTime“. Im zweiten Feld wird während einer Simulation die aktuelle Simulationszeit angezeigt. Die entsprechenden Funktionen zur Belegung und Aktualisierung des Displays werden im Rahmen der Dokumention der Schnittstelle für den Applikationsprozeß (*artlib*) in einem späteren Kapitel beschrieben.

Damit sind die Ausführungen zum statischen Modellaufbau abgeschlossen. Im Anschluß wird nun der dynamische Modellaufbau beschrieben, dabei erfolgen zum besseren Verständnis auch einige praktische Beispiele.

5 Dynamischer Modellaufbau

5.1 Basisobjekte als C-SIM-Prozesse und C-SIM Warteschlangen

Basisobjekte werden als grafische Repräsentanten für C-SIM-Prozesse und C-SIM-Warteschlangen eingesetzt. Durch die Angabe des Schlüsselwortes `process` in der ersten Zeile des Eingabeformulars wird das Basisobjekt als C-SIM-Prozeß und durch Eingabe des Schlüsselwortes `queue` als C-SIM Warteschlange definiert.

5.1.1 Basisobjekte als Prozesse

Nach der Eingabe des Schlüsselwortes `process` stehen dem Programmierer eine Vielzahl von Möglichkeiten zur Verfügung, um den C-SIM-Prozeß zu spezifizieren. Hierzu muß die Prozeßspezifikation einer bestimmten Syntax genügen, die durch den Compiler festgelegt wird. Die Prozeßspezifikation innerhalb des Eingabeformulars läßt sich hierbei in verschiedene Sektionen aufteilen.

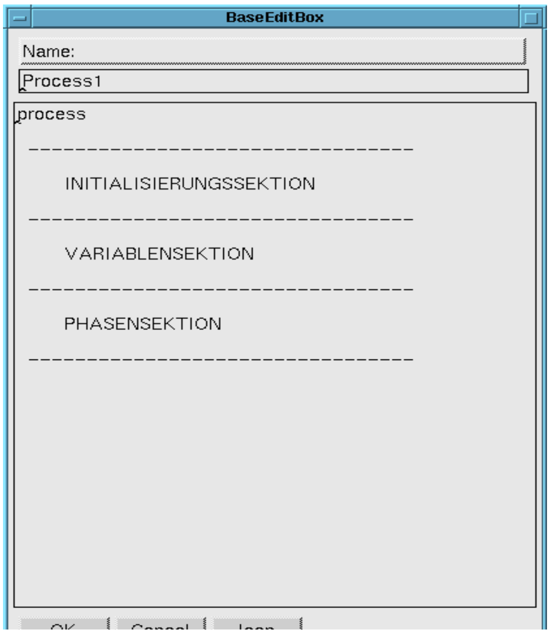


Abbildung 21: Aufteilung des Eingabeformulars für Basisobjekte

5.1.1.1 Initialisierungssektion

Die Initialisierungssektion wird mit dem Schlüsselwort `Init` eingeleitet.

```
process
  Init
    time = 0.7;
    status = PASSIVE;
   ...
```

Innerhalb dieser Sektion können vom Programmierer verschiedene Prozeßvariablen mit Startwerten belegt werden. Die Syntax zur Initialisierung lautet:

`time     = ...;`

Diese Variable legt die Startzeit für den Prozeß fest

`phase    = ...;`

Hier läßt sich diejenige Phase angegeben, die bei der ersten Aktivierung ausgeführt werden soll.

`status  = ...;`

Legt den Zustand des Prozesses für den Simulationsbeginn fest. Die möglichen Inititalisierungswerte entsprechen den in C-SIM erlaubten Statusangaben (`ACTIVE`, `PASSIVE`, ...).

```
function = ...;
```

Legt den Funktionstyp fest.

```
ptn      = ...;
```

Legt die Prozeßtypnummer fest.

Eine Reihenfolge bei der Initialisierung ist nicht vorgeschrieben. Ebenso ist die Anzahl der Initialisierungen optional, d.h. es müssen nicht alle Variablen mit Startwerten belegt werden. Bei fehlenden Eintragungen übernimmt der Compiler die Initialisierungen der nicht aufgeführten Variablen mit Defaultwerten. Verzichtet der Programmierer komplett auf die Initialisierungssektion, werden intern und für den Programmierer unsichtbar ebenfalls Defaultwerte verwendet.

Variable	*Defaultwert*
`time`	`Systemtime`
`phase`	Name der ersten Phase der Phasensektion (siehe 5.1.1.3)
`status`	`ACTIVE`
`function`	`0`
`ptn`	`3`

5.1.1.2 Variablensektion

Die Variablensektion beginnt mit dem Schlüsselwort `Local`. In dieser Sektion kann der Programmierer die lokalen Variablen des Prozesses deklarieren.

Beispiel:

```
Local
 QueueType LocalQueue;
 int counter1, counter2;
 JobType Job1, Job2;
```

Die Deklarationssyntax entspricht dabei der normalen ANSI C-Syntax.

Auch komplexere Deklarationen, etwa das Anlegen von Strukturdefinitionen sind in der Variablensektion möglich.

5.1.1.3 Phasensektion

Die Einleitung der Phasensektion geschieht durch Angabe des Schlüsselwortes `Phase`. In der Phasensektion werden die auszuführenden Phasen eines Prozesses spezifiziert. Dabei werden Anfang und Ende einer Phase durch folgende Syntax festgelegt:

```
PHASE(<Phasenname>)
  <Phasencode>
END
```

Jede Phase ist durch einen Phasennamen gekennzeichnet. Innerhalb der Prozeßspezifikation müssen eindeutige Phasennamen gewählt werden; die Wiederverwendung von Phasennamen in anderen Prozeßspezifikationen ist jedoch erlaubt. Nach der Festlegung des Phasennames kann der Programmierer die Phase programmieren. Die Programmierung einer Phase folgt hierbei der gewohnten C-SIM- bzw. C-Syntax. Zu Beginn einer Phasenspezifikation kann die Deklaration phasenlokaler Variablen vorgenommen werden.

Beispiel:

```
PHASE(Init)
  int i; /*phasenlokale Variable*/
  for(i = 0; i < 10; i++)
    . . .
END
```

Die Verwendung lokaler Variablen (Variablen aus der Variablensektion) innerhalb des Phasencodes erfordern vom Programmierer keine besondere Berücksichtigung. Lokale Variablen werden vom Compiler automatisch erkannt und als solche gekennzeichnet. Eine spezielle Markierung der Variablen (in der herkömmlichen C-SIM-Programmierung üblicherweise mit Hilfe des `THIS`-Makros) wird dem Programmierer vom System abgenommen.

Wird innerhalb des Phasencodes der Name eines anderen Prozesses verwendet, ist es notwendig, den vollständigen Pfadnamen des angesprochenen Prozesses anzugeben. Der Pfadname eines Prozesses setzt sich dabei aus den Namen der umgebenden komplexen Objekte und dem eigentlichen Prozeßnamen (Name des Basisobjekts) zusammen.

Beispiel:

Um im Phasencode die lokale Variable `data` eines Prozesses `Process2` anzusprechen, wird folgende Syntax verwendet:

a) `Complex1Complex2Process2->data = ...;`

`Process2` befindet sich hier im komplexen Objekt `Complex2`, wobei `Complex2` wiederum in `Complex1` geschachtelt ist.

b) `Process2->data = ...;`

`Process2` befindet sich direkt auf der Root-Ebene, die Angabe des Basisobjektnamens ist ausreichend, d.h. befindet sich ein Prozeß direkt auf der Root-Ebene und ist somit nicht Teil eines komplexen Objekts, wird der Prozeßname bzw. Basisobjektname direkt angegeben.

5.1.1.4 Basisobjektenamen als Parameter von C-SIM-Funktionen

Die Angabe des vollständigen Pfadnamens ist nicht notwendig, falls Prozeßnamen als Argumente von C-SIM-Funktionen übergeben werden. In diesem Fall übernimmt der Compiler eine korrekte Ersetzung des Prozeßnamens durch den vollständigen Pfadnamen.

<u>*Beispiel*</u>:

a) `Insert(Job, Queue);`

Auch wenn sich die referenzierte Warteschlange `Queue` in einem komplexen Objekt befindet, reicht die einfache Angabe des Basisobjektnamens aus, um ein korrektes Ergebnis zu erzielen. Der Compiler überprüft, ob eine Verbindung zu einem Basisobjekt mit dem Namen `Queue` existiert. Falls das angesprochene Basisobjekt `Queue` Teil eines komplexen Objekts ist, wird der entsprechende Pfadname vom Compiler eingesetzt.

b) `Insert(Job, Process2->Queue);`

Hier wird eine lokale Warteschlange des Prozesses `Process2` an die Insert-Funktion übergeben. Auch in diesem Fall erfolgt eine Ersetzung des Arguments falls sich der Prozeß `Process2` nicht auf der Root-Ebene befindet.

Der Verzicht auf die automatische Ersetzung des Pfadnamens bei Referenzierungen, die nicht als C-SIM-Funktionsargumente übergeben werden, geschieht aus Effizienzgründen. Die Überprüfung, ob es sich bei einem Bezeichner im Phasencode um den Namen eines Basisobjektes handelt, würde die Compilierungszeit erheblich verlängern würde. Da Prozesse „im Normalfall“ über C-SIM-Funktionen kommunizieren, beschränkt sich Xcalibur daher auf das gezielte Überprüfen der Funktionsargumente.

Wie bereits erläutert worden ist, lassen sich über den Menüpunkt *Child* im Funktionsmenü eines Basisobjekts beliebig viele Kinderprozesse (Children) zu einem Prozeß erzeugen. Aus programmiertechnischer Sicht handelt es sich bei den Children um Instanzen eines bestimmten C-SIM Prozesses. Children erhalten dabei den Namen des Vaterprozesses ergänzt um einen Index. Über diesen Index lassen sich die einzelnen Children im Phasencode gezielt ansprechen. Children übernehmen die Prozeßspezifikation von ihrem Vaterobjekt, d.h. die lokale Variablen und Phasen des Vaterobjekts gelten ebenfalls für die Kinderobjekte. Childern verhalten sich im Programmverlauf wie ihr Vaterobjekt, da sie die im Vaterobjekt spezifizierten Phasen durchlaufen. Hierbei ist zu beachten, daß Kinderprozesse die lokalen Variablen und Phasen des Vaters benutzen, d.h. es werden beim Erzeugen für das Child keine neuen bzw. eigenen lokalen Variablen und Phasen angelegt.

Beispiel:

a)
```
Send(Job, Client[3]);
```

Durch diesen Funktionsaufruf wird das Kind des Prozesses `Client` mit dem Index 3 angesprochen.

b)
```
for(i = 0; i <= 5; i++){
          ...
     Send(Job, Client[i]);
}
```

In dieser `for`-Schleife wird die `Send`-Funktion sowohl für den Vaterprozeß, als auch für die fünf Children dieses Prozesses ausgeführt. Der Vaterprozeß wird dabei im ersten Schleifendurchlauf angesprochen (Variable i gleich 0).

c)
```
Insert(Job, Client[0]->Queue);
```

Der Aufruf der `Send`-Funktion in Beispiel b) unterschiedet sich vom Aufruf in Beispiel a) dadurch, daß der referenzierte Prozeß zur Übersetzungszeit nicht bekannt ist, sondern in Abhängigkeit vom Wert von i zur Laufzeit bestimmt werden muß. Soll ein Vaterprozeß direkt angesprochen werden, d.h. ohne eine Laufzeitvariable wie in b), so muß dies wie in Beispiel c) mit Indexangabe `[0]` geschehen.

Um dem Programmierer zu ermöglichen, daß sich ein Child in der Phasenausführung vom Vaterobjekt oder den übrigen Kinderprozessen unterscheidet, kann im Programmcode die spezielle Systemvariable `index` verwendet werden.

Beispiel:

```
if(index == 5)
      RepeatAfter(0.2);
else
      RepeatAfter(0.7);
```

Hier wird für das Child mit dem Index 5 ein eigener Zeitpunkt der erneuten Phasenausführung angegeben, die übrigen Kinder und der Vaterprozeß verhalten sich nach der Anweisung im `else`-Zweig. Die Variable `index` wird automatisch vom System angelegt und braucht deshalb nicht vom Programmierer deklariert zu werden.

Beim Löschen eines Childs ist darauf zu achten, daß das Basisobjekt mit dem jeweils höchsten Index von der Oberfläche entfernt wird. Das Löschen eines Childs mit einem kleineren als dem höchsten Index bewirkt dagegen nur das Entfernen der grafische Repräsentation, d.h. die C-SIM-Prozeßinstanz selbst wird nicht gelöscht und arbeitet für den Betrachter unsichtbar im Hintergrund.

Oft ist die Anzahl der in einer Simulation geforderten C-SIM-Prozeßinstanzen so groß, daß es für den Programmierer unzumutbar wird, die Erzeugung sämtlicher Children über das Menü vorzunehmen. Wird nach dem Schlüsselwort `process` die Anweisung

```
children = <Anzahl der Prozessinstanzen>;
```

angegeben, läßt sich die gewünschte Anzahl der Children auf einfachere Art festlegen. Eine grafische Repräsentation der auf diese Weise angelegten Prozeßinstanzen entfällt jedoch. Werden für einen Vaterprozeß Kinderprozesse über das Menü erzeugt und darüberhinaus weitere Kinder mit der obigen Anweisung gefordert, ist demnach nur das Simulationsverhalten der über das Menü generierten Kinderprozesse sichtbar.

5.1.2 Basisobjekte als Warteschlangen

Mit dem Schlüsselwort `queue` in der ersten Zeile des Eingabeformulars wird das Basisobjekt als Warteschlange festgelegt.

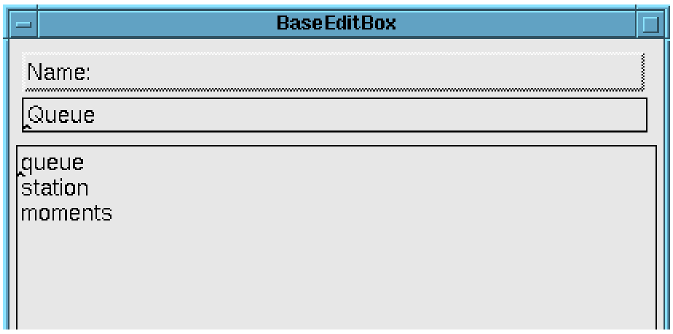


Abbildung 22: Eingabeformulareintragungen bei Warteschlangen

5.1.2.1 Stationen und Warteschlangen

Folgt die Angabe des Schlüsselwortes `station`, wird für die Warteschlange eine Station vom Typ `StationType` angelegt. Mit Hilfe einer Station kann eine Durchsatzmessung für eine Warteschlange vorgenommen werden. Eine vom System automatisch erzeugte Station kann auf folgenden Weise im Pogrammcode angesprochen bzw.verwendet werden.

```
<Pfadname der Warteschlange>->StationRef;
```

`StationRef` ist hier eine lokale Zeigervariable auf die vom System angelegte Station.

Die Abfrage der in der Station gespeicherten Daten erfolgt mit Hilfe der in C-SIM zur Verfügung gestellten Auswertungsfunktionen. Als Parameter für eine Auswertungsfunktion wird in C-SIM die auszuwertende Stationsstruktur vom Typ `StationType` übergeben. In Lancelot kann als Argument der Name einer Warteschlange direkt übergeben werden. Der Compiler nimmt dann die Ersetzung des Warteschlangennames durch den zugehörigen Stationsnamen automatisch vor.

Beispiel:

```
printf(„Füllung der Warteschlange Queue: %d“, Residual(Queue));
```

Xcalibur ersetzt in diesem Fall das übergebene Warteschlangenargument durch den korrekten Namen der zugehörigen Station. Diese Ersetzung erfolgt allerdings nur für Warteschlangen, die durch ein Basisobjekt in der Benutzungsoberfläche dargestellt sind. Für Warteschlangen die global oder lokal definiert sind (und für die vom System ebenfalls automatisch Stationen angelegt werden), muß der Programmierer die Referenz auf die vom System erzeugte Station explizit angeben.

<u>*Beispiel:*</u>

```
printf(„Füllung GlobalQueue: %d“,Residual(GlobalQueue->StationRef);
```

Bei eingeschaltetem Menüpunkt „autom. Station on“ wird zu jeder durchgeführten Insert- bzw. Seize-Funktion auf die Warteschlange eine StationIn- bzw. StationOut-Funktion für die zugehörige Station aufgerufen, d.h. die Station wird automatisch aktualisiert. Die Aktualisierung bezieht sich sowohl auf die Anzahl der Jobobjekte in der Warteschlange, als auch auf die Verweilzeit der Jobobjekte in der Warteschlange.

5.1.2.2 lokale und globale Warteschlangen

Lokale Warteschlangen sind Warteschlangen die für einen Prozeß lokal spezifiziert werden. Die Deklaration solcher Warteschlangen erfolgt in der Variablensektion.

<u>*Beispiel*</u>:

```
Local
   QueueType LocalQueue1, LocalQueue2;
```

Auch für lokale Warteschlangen werden Stationen automatisch anlegt und bei entsprechender Anwendung einer `Insert`- oder `Seize`-Funktion auf die zugehörige Warteschlange aktualisiert.

<u>*Beispiel:*</u>

<u>Prozeßspezifikation `process1`</u> <u>Prozeßspezifikation `process2`</u>

```
Local
  QueueType Queue;
Phase
  PHASE(init)
    Queue = NewQueue(0);
  END
```

```
Local
   JobType Job;
Phase
  PHASE(action)
     int i;
     for(i = 1; i <= 10; i++){
       Job = NewJob(icon2);
       Insert(Job,process1->Queue);
       }
     printf(„%d\n,Residual(process1->
                   Queue->StationRef));
END
```

Für die lokale Warteschlange des Prozeß `process1` wird vom System eine lokale Station angelegt. Das Ansprechen der Warteschlange durch den `Insert`-Aufruf von `process2` bewirkt eine automatische Aktualisierung der Station. Wird die Phase `init` vor der Phase `action` bzw. `process1` vor `process2` ausgeführt, so wird durch die Print-Anweisung in Phase `action` die Zahl zehn ausgegeben. Da es sich hier um die Auswertung einer lokalen Warteschlange handelt, die auf der Oberfläche nicht durch ein eigenes Basisobjekt repräsentiert wird, muß die Endung `->StationRef` vom Programmierer angefügt werden.

Globale Warteschlangen sind Warteschlangen, die im Code-Formular des Root-Objekts deklariert werden. Wie die lokalen gehören auch die globalen Warteschlangen zu keinem Basisobjekt im Root-Formular. Jedoch wird auch für diesen Warteschlangentyp automatisch eine Station angelegt, auf die dann mit `<Warteschlangenname>->StationRef` zugegriffen werden kann.

5.1.2.3 Momente und Warteschlangen

Bei Angabe des Schlüsselwortes `moments` im Eingabeformular einer Warteschlange wird für die Warteschlange eine lokale Datenstruktur angelegt. Dieser neu angelegte Datensatz dient wie der Stationstyp zur Auswertung von Warteschlangeninformationen.

5.2 Iconobjekte

Der Informationsaustausch zwischen den einzelnen Prozessen einer Simulation erfolgt in C-SIM mit Hilfe von Aufträgen (Jobs). Jeder Auftrag, der während eines Simulationslaufs verschickt wird, ist dabei eine C-Variable vom Typ `JobType`. Bevor eine Auftragsvariable eingesetzt werden kann, ist es demnach notwendig, sie im Programmtext zu deklarieren und ihr den entsprechenden Speicherplatz zuzuweisen.

Beispiel:

```
Local
  JobType signal;
Phase
  PHASE(init)
    signal = NewJob();
  ...
```

Die Funktion `NewJob` legt dabei den notwendigen Speicherplatz für die Auftragsvariable an.

Iconobjekte sind in Lancelot die grafischen Repräsentanten für Aufträge, d.h. mit Hilfe von Iconobjekten wird der Signalaustausch zwischen den Prozessen innerhalb des Root-Formulars sichtbar gemacht. Ein Iconobjekt ist demnach ein vom Programmier definiertes Sinnbild, welches das Aussehen eines Auftrags für eine Animation festlegt.

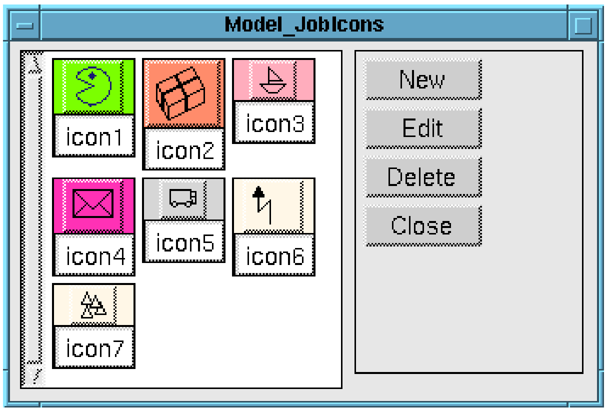


Abbildung 23: Auswahl verschiedener Iconobjekte

Wird ein Auftrag im Programmtext mit einem bestimmten Iconobjekt verknüpft, so erfolgt beim Versenden dieses Jobs eine entsprechende Animation mit dem gewählten Icon im Root-Formular.

Die Verknüpfung zwischen Auftrag und Iconobjekt erfolgt im Programmtext durch die Übergabe eines Parameters an die Funktion `NewJob`.

Beispiel:

```
Local
  JobType signal;
Phase
  PHASE(init)
    signal = NewJob(icon1);
    ...
```

Hier wird der Funktion `NewJob` der Name eines Iconobjekts übergeben, mit dem der Auftrag `signal` während des Simulationslaufs sichtbar gemacht werden soll. Der Aufruf

```
Insert(signal, Queue);
```

löst beispielsweise eine Animation auf der Oberfläche aus, bei der sich das Iconobjekt mit dem Namen `icon1` entlang der Verbindungslinie (Verbindungsobjekt) vom aufrufenden Prozesses zum grafischen Repräsentanten einer Warteschlange `Queue` bewegt.

Eine Verknüpfung zwischen Jobvariable und Iconobjekt läßt sich ebenfalls auf nachfolgend beschriebene Weise herstellen.

Beispiel:

```
. . .
PHASE(init)
  signal = NewJob();
  . . .
  signal->PID = icon1;
  . . .
```

Jede Jobvariable besitzt dazu eine Komponente `PID` in der sich die Identifikationsnummer eines Iconobjekts speichern läßt. Der Compiler wandelt dann den vom Programmierer angegebenen Namen des Iconobjekts in die zugehörige Identifikationsnummer um. Erst nach der Zuweisung eines Iconobjekts an die Komponente `PID` wird das Versenden eines Auftrags mit einer entsprechenden Animation durchgeführt.

Beispiel:

```
signal = NewJob();
if(Bedingung_A)
   signal->PID = icon1
else if(Bedingung_B)
   signal->PID = icon2
Insert(signal, Queue);
```

Ist die Bedingung der ersten `if`-Abfrage erfüllt, wird das Iconobjekt `icon1` zur Animation eingesetzt; ist die zweite Bedingung erfüllt, erfolgt eine Animation mit Iconobjekt `icon2`. Wird in keine der beiden `if`-Abfragen verzweigt, bleibt die Animation der `Insert`-Funktion aus, da kein entsprechendes Iconobjekt für den Job definiert worden ist..

Alternativ läßt sich ein Iconobjekt als Parameter an die Funktion `NewJob` übergeben; das übergebene Iconobjekt wird dann für die Animation des Auftrags verwendet. Es ist jederzeit möglich, der Variablen `PID` anschließend einen neuen Iconnamen zuzuweisen. Die Variable kann aber auch mit `Null` überschrieben werden, um eine Animation zu unterdrücken.

Beispiel:

```
signal = NewJob(icon2);    /*Es wird mit icon2 animiert*/
signal->PID = icon1;       /*Es wird mit icon1 animiert*/
signal->PID = Null;        /*Es erfolgt keine Animation*/
Insert(signal, Queue);
```

5.3 Typobjekte

Statt der expliziten Angabe der Variablensektion (siehe 5.1.1.2) hat der Programmierer die Möglichkeit ein Typobjekt anzulegen, das die gewünschten lokalen Variablen enthält. Ist ein Typobjekt erstellt worden, kann die Variablensektion eines Prozesses, die mit dem Schlüsselwort `Local` eingeleitet wird, durch folgende Syntax ersetzt werden:

```
processtype: <Name des Typobjekts>
```

Sollen in einer Simulation mehrere Prozesse mit gleichen lokalen Variablen eingesetzt werden, so genügt in den jeweiligen Prozeßspezifikationen die Angabe des Typobjekts. Der Programmierer spart sich auf diese Weise das explizite Ausschreiben der Variablendeklaration in den jeweiligen Eingabeformularen der einzelnen Prozesse.

Typobjekte entsprechen den Prozeßtypen in C-SIM. Prozeßtypen werden in der herkömmlichen C-SIM-Programmierung durch folgende Syntax festgelegt:

```
ProcessData(Server1)
   int a, b;
};
```

Hierdurch wird ein Prozeßtyp `Server1Typ` definiert, der die beiden lokalen Variablen a und b vom Typ Integer beinhaltet. In Lancelot erfolgt die Definition eines Prozeßtyps durch die Erzeugung eines Typobjekts. Der Programmierer trägt dazu im Eingabeformular eines Typobjekts die gewünschten lokalen Variablen ein und vergibt einen eindeutigen Namen für das Objekt.

Typobjekte werden automatisch vom Compiler in das Simulationsprogramm aufgenommen, und in die entsprechende C-SIM-Notation übersetzt. Auf diese Weise angelegte Prozeßtypen lassen sich auf einfache Weise speichern und in verschiedenen Simulationen einsetzen.

Typobjekte dienen neben ihrer Verwendung als Prozeßtypen auch zur Definition des globalen QueueTyps und Jobtyps. In der herkömmlichen C-SIM-Programmierung lautet die Notation für diese Typvorlagen:

```
QueueData
<Variablendeklarationen>
};

JobData
<Variablendeklarationen>
};
```

Die globalen Typvorlagen `QueueData` und `JobData` werden in Lancelot ebenfalls durch Typobjekte definiert. Wird ein Typobjekt mit dem Namen `QueueData` erzeugt, lassen sich die lokalen Variablen für den globalen Queuetyp im Eingabeformular dieses Objekts angeben.

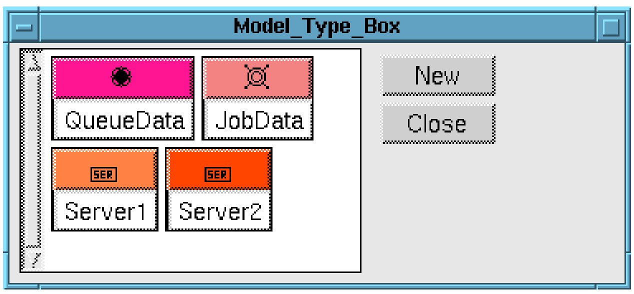


Abbildung 24: Auswahlmaske für Typ-Objekte

Variablendeklarationen im Eingabeformular eines Typobjekts mit Namen `JobData` definieren dann die lokalen Variablen des globalen Jobtyps. Bei fehlenden Typobjekten werden vom Compiler beide Typvorlagen ohne lokale Variablen angelegt.

5.4 Code-Formular

Im Code-Formular des Root-Objekts wird vom System automatisch die Eintragung `main()` vorgenommen. C-SIM ist eine Erweiterung der Programmiersprache C und erwartet deshalb die explizite Angabe der Hauptfunktion. Wie in einem C-Programm kann der Programmierer vor dem main-Aufruf globale Variablen und Funktionsdefinitionen deklarieren und danach beliebige Variablendeklarationen und Funktionsaufrufe folgen lassen. Im Unterschied zum C-Programm darf eine Klammerung der `main`-Funktion nicht erfolgen.

5.4.1 Simulationszeit

5.4.1.1 Simulationsdauer

Um die Dauer der Simulation festzulegen, muß im Code-Formular die Systemvariablen `SimulationTime` angegeben und dieser ein entsprechender Wert zugewiesen werden. Das Einfügen der Variablen kann global oder lokal erfolgen,d.h. vor oder nach der main-Funktion.

<u>*Beispiel:*</u>

```
SimulationTime = 20000;

main()

/* SimulationTime = 1500; */

...
```

Fehlt die Angabe der Simulationsdauer wird Zeitspanne einer Simulation automatisch auf 100 Zeiteinheiten festgelegt.

5.4.1.2 Ausgabe der Simulationszeit

Die Ausgabe der aktuellen Simulationszeit in der oberen rechten Ecke des Root-Fensters kann durch den Aufruf der Funktion `ShowSimTime(double SimTimeStep ,int Nachkommastellen)` erfolgen. Dieser Aufruf muß vor dem Hauptfunktionsaufruf (main-Funktion) plaziert werden. Der erste Parameter legt die Zeitspanne fest, mit der eine

Aktualisierung der angezeigten Simulationszeit geschehen soll. Der zweite Parameter gibt an mit welcher Genauigkeit (in Bezug auf die Nachkommastellen) die Ausgabe erfolgt.

Beispiel:

```
ShowSimTime(0.9, 5);

main()

...
```

Im Beispiel wird die Ausgabe der Simulationszeit nach jeweils 0.9 Zeiteinheiten aktualisiert und mit einer Genauigkeit von fünf Stellen hinter dem Komma ausgegeben.

5.5 Konstruktion eines Simulationsmodells

Die bis hierher beschriebene Funktionalität von Lancelot wird in diesem Abschnitt zur Konstruktion eines konkreten Simulationsmodells angewandt. Das Modell wird dabei schrittweise von der Entwurfsphase bis zur ablauffähigen Simulation entwickelt.

5.5.1 Modellentwurf

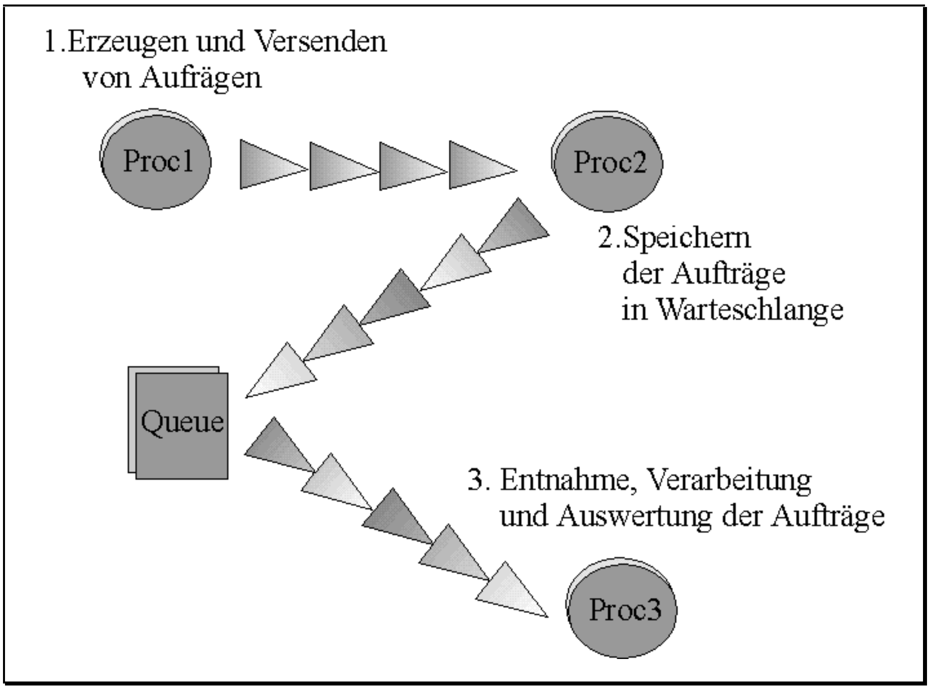


Abbildung 25: Schematische Darstellung des Simulationsmodells

Das Modell soll dabei aus insgesamt drei Prozessen und einer Warteschlange bestehen, die über Aufträge miteinander kommunizieren. Der erste Prozeß erzeugt Aufträge und verschickt diese an den zweiten Prozeß. Die von Prozeß zwei entgegengenommenen Aufträge werden in einer Warteschlange zwischengespeichert und anschließend von Prozeß drei entnommen, verarbeitet und ausgewertet (siehe Abbildung 20).

5.5.2 Anlegen eines Iconobjekts

Die während eines Simulationslaufs animierten Aufträge sollen im Modell durch ein einheitliches Icon repräsentiert werden. Es ist daher ausreichend nur ein Iconobjekt zu erstellen. Als Name für das neu angelegte Icon wählen wir `icon1`.

5.5.3 Erzeugen der Prozesse und der Warteschlange im Typ-Manager

Zur Erzeugung des ersten Prozesses wird im Typ-Mananger ein neuer Basisobjekt-Typ mit dem Namen `Proc1` angelegt. Die Programmierung im Eingabeformular des Basisobjekts beginnt mit der Eintragung `process`, durch die ein Basisobjekt als Prozeß ausgewiesen wird.

```
process
   Local
     Job JobType;
   Phase
     PHASE(INIT)
       Job = NewJob(icon1);
       Send(Job, Proc2);
       RepeatAfter(0.3);
     END
```

Aufgabe des Prozesses ist es, Aufträge zu generieren und diese in regelmäßigen Abständen an einen zweiten Prozeß zu senden. Hierzu wird in der Variablensektion eine Auftragsvariable `Job` deklariert, die zu Beginn der einzigen Phase des Prozesses allokiert wird. Dem Allokationsaufruf für die Variable wird dabei der Name des im vorherigen Entwicklungsschritts erstellten Icons übergeben. Nach der Allokation wird ein `Send`-Befehl, der den Auftrag zu `Proc2` schickt, aufgerufen. Der abschließende Aufruf `RepeatAfter` legt die Zeitdauer für eine erneute Ausführung der Phase fest. `RepeatAfter` definiert somit die zeitlichen Abstände zwischen denen der Prozeß einen Auftrag erzeugt und zum Prozeß `Proc2` abschickt. Hier das Eingabe-Formular für `Proc2`:

```
process
  Local
    JobType Job;
  Phase
    PHASE(DELAY)
       Await(Job);
       NextAfter(SEND, 0.1);
    END
    PHASE(SEND)
       Insert(Job, Queue);
       NextAfter(DELAY, 0.0);
    END
```

Es folgt die Erzeugung des Prozesses `Proc2`, der die Aufträge von `Proc1` entgegennimmt und sie anschließend in eine Warteschlange einfügt. Für diesen Prozeß wird in der Variablensektion wieder eine Auftragsvariable deklariert. Die Phasensektion besteht hier aus den Phasen `DELAY` und `SEND`. In der ersten Phase wird die `Await`-Funktion aufgerufen, die eine weitere Ausführung des Prozesses so lange verhindert, bis ein Auftrag eintrifft. Ist dies der Fall, so wird durch die anschließende `NextAfter`-Funktion die Phase `SEND` aufgerufen. In der `SEND`-Phase wird der entgegengenommene Auftrag durch einen `Insert`-Aufruf in die Warteschlange `Queue` eingefügt.

Als nächstes wird der dritte Prozeß `Proc3` erzeugt. Dieser Prozeß entnimmt in bestimmten Abständen Aufträge aus der Warteschlange.

```
process
  Local
    JobType Job;
  Phase
    PHASE(GET)
       if(Queue->Number != 0){
          Job = Seize(Queue);
          NextAfter(ACTION, 01);
       } else
          RepeatAfter(0.3);
    END
    PHASE(ACTION)
       DISPOSE(Job);
       NextAfter(GET, 0.0);
    END
```

Simuliert wird dies durch eine Phase `GET`, in der die Auftragsentnahme mittels eines `Seize`-Befehls erfolgt. In einer `if`-Abfrage wird geprüft, ob sich ein Auftrag in der Warteschlange befindet; falls ja wird mit `NextAfter` in die nächste Phase verzweigt, die den Auftrag verarbeitet bzw. mit `DISPOSE` löscht.

Als letztes Objekt des Modells wird nun die Warteschlange `Queue` erzeugt. Zur Spezifikation der Warteschlange wird die Syntax aus Abbildung 17 verwendet.

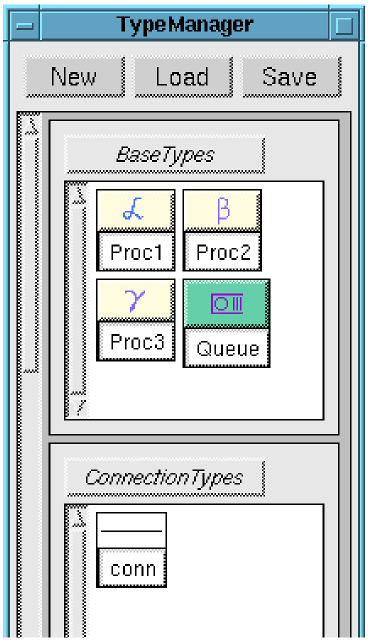


Abbildung 26:Typ-Manager mit den erzeugten Basisobjekt-Typen

Nach der Erzeugung der vier Basisobjekte und der Auswahl geeigneter Icons für die einzelnen Objekte, ergibt sich das obige Bild für den Typ-Manager. Anschließend werden Instanzen dieser Objekt-Typen erzeugt und mit den in der nächsten Abbildung gezeigten Verbindungen im Root-Formular eingefügt.

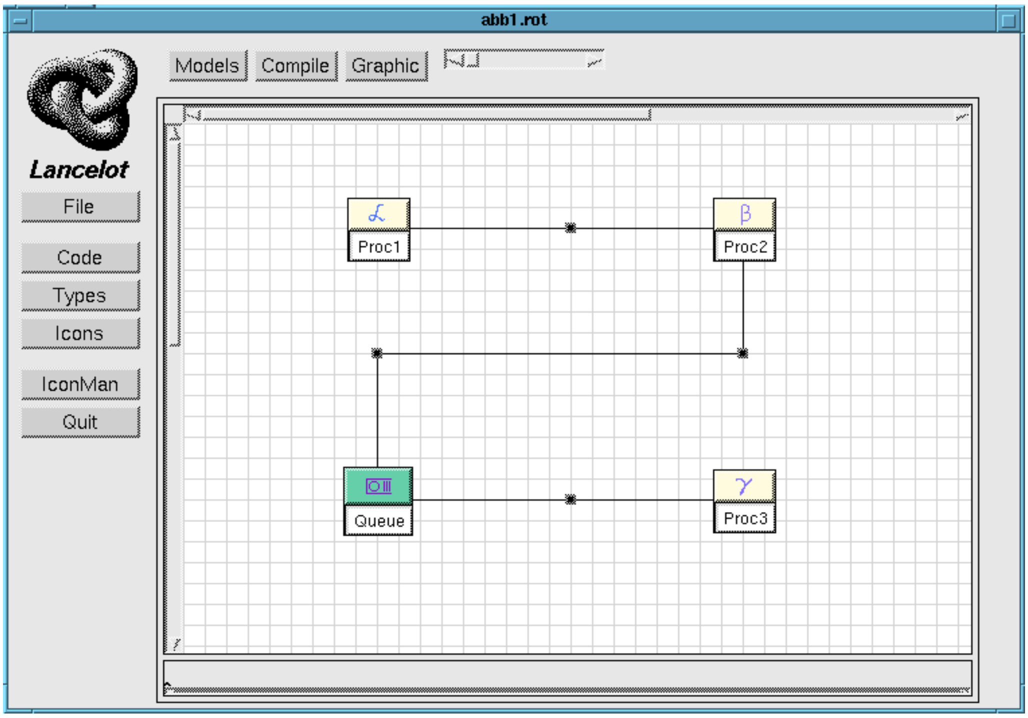


Abbildung 27: Beispielmodell

Da für unser einfaches Simulationsmodell keine globalen Variablen definiert werden müssen, beschränken sich die Eintragungen im Code-Formular des Root-Objekts auf die Angabe der `main`-Funktion und die Festlegung der Simulationsdauer.

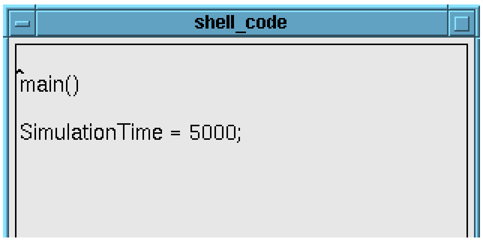
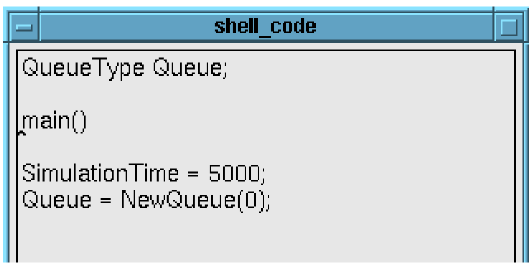


Abbildung 28: Formular des Root-Objekts

Die Modellspezifikation ist mit dem vorherigen Schritt abgeschlossen, d.h. Xcalibur und C-Compiler können über die Benutzungsoberfläche aufgerufen werden, um ein ablauffähiges Simulationsprogramm zu generieren. Anschließend kann die Simulation über den Menüpunkt *RunSimulation* gestartet werden.

5.5.4 Einfügen „unsichtbarer“ Warteschlangen

Die im Modell von den Prozessen angesprochene Warteschlange `Queue` wird im Root-Objekt durch ein Basisobjekt repräsentiert. Werden zur Laufzeit der Simulation Aufträge in die Warteschlange eingefügt oder aus ihr entnommen, wird dieser Vorgang durch eine Animation visualisiert. Möchte der Anwender Warteschlangen integrieren, die nicht auf der Oberfläche erscheinen und nur „im Hintergrund“ existent sind, muß eine entsprechende Spezifikation im Programmcode vorgenommen werden. Die Spezifikation einer solchen Warteschlange kann im Code-Formular des Root-Objekts erfolgen. Zur Veranschaulichung ersetzen wir im Modell die grafisch repräsentierte Warteschlange `Queue` durch eine unsichtbare globale Warteschlange gleichen Namens. Dazu wird das Basisobjekt von `Queue` auf der Oberfläche gelöscht und die nachfolgend abgebildeten Eintragungen im Code-Formular eingefügt:

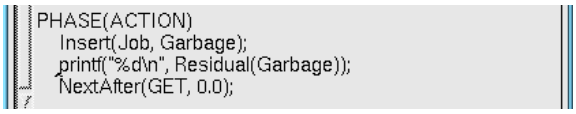


Die Simulation reagiert unverändert, bis auf die Tatsache, daß das Einfügen und Entnehmen der Aufträge bezüglich der Warteschlange `Queue` unsichtbar bleibt.

Im Gegensatz zu den globalen Warteschlangen wird die Spezifikation lokaler Auftragsspeicher im Eingabeformular eines Prozesses durchgeführt. Um beispielsweise eine lokale Warteschlange `Garbage` für `Proc3` zu spezifizieren werden folgende Eintragungen vorgenommen.

```
Local
  JobType Job;
  QueueType Garbage;
Phase
  PHASE(INIT)
     Garbage = NewQueue(0);
     NextAfter(GET, 0.0);
  END
  ...
```

In der Variablensektion wurde eine Variable `Garbage` deklariert und in der Phasensektion entsprechender Speicherplatz reserviert.



```
PHASE(ACTION)
   Insert(Job, Garbage);
   printf("%d\n", Residual(Garbage));
   NextAfter(GET, 0.0);
```

Um die Anwendung lokaler Warteschlangen im Phasencode zu verdeutlichen, wurde in der Phase `Action` von `Proc3` der `DISPOSE`-Befehl durch einen `Insert`-Aufruf ersetzt worden. Die anschließende Print-Anweisung, gibt die aktuelle Füllung für `Garbage` aus. Soll die lokale Warteschlange von einem der übrigen Prozesse (z.B. `Proc2`) angesprochen werden, ist folgende Syntax zu benutzen:

```
Insert(Proc3->Garbage);
Seize(Proc3->Garbage);
```

In diesen Fällen wird jeweils eine Animation durchgeführt, d.h. entlang der Verbindungslinie bewegt sich ein Auftrags-Icon zwischen den beteiligten Prozessen, von denen der eine die lokale Warteschlange verbirgt.

Die Deklaration unsichtbarer Warteschlangen folgt der üblichen C-Syntax. Demnach ist ebenfalls die Angabe mehrerer, durch Kommata getrennter Warteschlangenbezeichner erlaubt:

```
QueueType Queue1, Queue2, Queue3;
```

Darüber hinaus lassen sich auch Felder von Warteschlangen anlegen:

```
QueueType Queue[10];
```

Die Speicherplatzreservierung läßt sich dann in einer `for`-Schleife mit entsprechender Laufvariable vornehmen:

```
for(i = 0; i < 9; i++){
 Queue[i] = NewQueue(0);
}
```

5.5.5 Erzeugen von Children

Im nächsten Schritt soll das Modell um Children für `Proc1` erweitert werden, die zusammen mit dem Vaterprozeß in einem komplexen Objekt `Comp1` gruppiert werden. Wir löschen zunächst den Prozeß `Proc1` aus dem Root-Formular, von dem aber nach wie vor eine Kopie im Typ-Manager existiert. Anschließend wird im Typ-Manager ein neuer komplexer Objekt-Typ angelegt. Der zu Beginn der Modellerstellung erzeugte Prozeß `Proc1` aus dem Typ-Manager wird in den komplexen Objekt-Typ kopiert. Jetzt werden innerhalb des komplexen Objekt-Typs drei Children von `Proc1` angelegt.

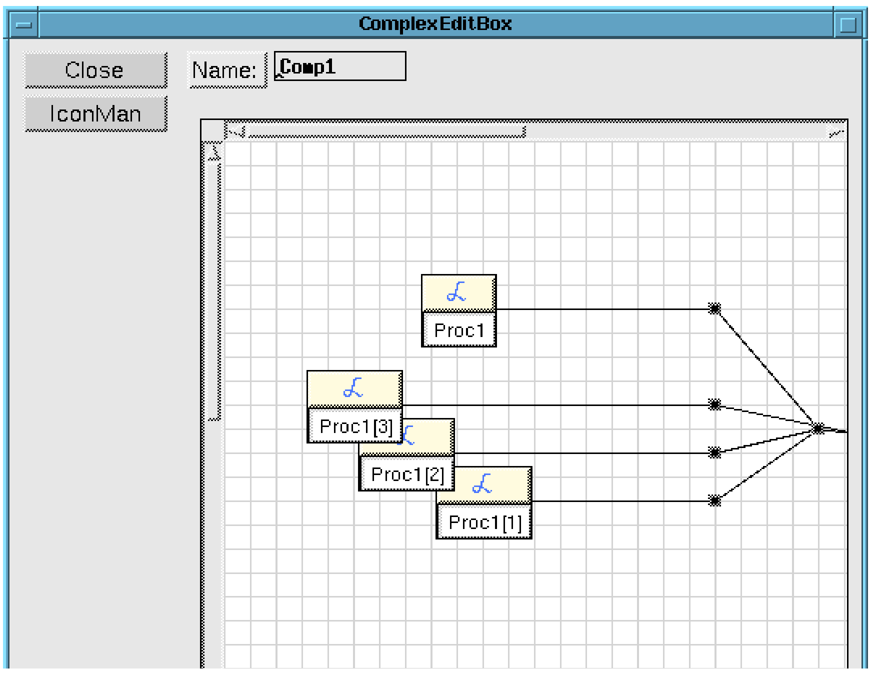


Abbildung 29: Vaterprozeß mit Childprozessen in `Comp1`

Die Instanz `Comp1` des komplexen Objekt-Typs, die nun vier Basisobjekte beinhaltet, wird in das Root-Formular kopiert. Anschließend werden die gruppierten Prozesse des komplexen Objekts einzeln mit `Proc2` verbunden. Das Modell kann nun ohne weitere Änderungen compiliert und gestartet werden.

Wichtig ist, daß die Prozesse vom System automatisch in der Reihenfolge gestartet respektive ausgeführt werden, in der man sie in das Root-Formular kopiert. Demnach werden `Proc2` und `Proc3` vor den gruppierten Prozessen in `Comp1` abgearbeitet, da `Comp1` erst nach den beiden anderen Prozessen im Root-Formular plaziert worden ist.

Die Ausführungsreihenfolge für die gruppierten Objekte ist demnach: `Proc1`, `Proc1`[1], `Proc1`[2], `Proc1`[3]. Children werden immer nach aufsteigendem Index gestartet.

Die Reihenfolge in der Ausführung, die durch das Erzeugen von Prozessen auf der Oberfläche festgelegt wird, kann jedoch im Programmcode manipuliert werden. Dazu kann für jeden Prozeß der Ausführungszeitpunkt in der Initialisierungssektion (siehe Abschnitt 5.1.1.1) explizit angegeben werden. Verantwortlich für den Zeitpunkt der ersten Ausführung eines Prozesses ist die Variable `time.`

Beispiel:

```
Init
   time = 0.1;
```

Eine Modifizierung des Programmcodes um den obigen Eintrag in den Eingabeformularen von `Proc2` und `Proc3` bewirkt, das beide Prozesse nach den gruppierten Prozessen in `Comp1` gestartet werden, da die Startzeit der gruppierten Prozesse vom System mit dem Defaultwert 0. `time = 0.0` initialisiert worden ist.

Ohne weiter Änderungen im Programmcode vornehmen zu müssen, kann auch das komplexe Objekt `Comp1` kopiert werden.

Bis auf einen eindeutigen Namen für die Kopie (z.B. `Comp2`) und zusätzlichen Verbindungen von den Prozessen innerhalb des neuen komplexen Objekts zu `Proc2` brauchen vom Programmierer vor einer Neucompilation keine weiteren Aktionen durchgeführt werden. Insgesamt erzeugen nun acht Prozesse (jeweils vier aus einem komplexen Objekt) einzelne Aufträge und schicken diese an `Proc2`.

Als nächstes wird gezeigt, durch welche Maßnahmen eine Animation ausgelöst werden kann, auch wenn das Ziel der Animationsbewegung (z.B. eine Warteschlange) in einer C-SIM-Funktion (z.B. `Insert`) zur Übersetzungszeit unbekannt ist.

Für die nachfolgenden Betrachtungen gehen wir von dem unten abgebildeten Grundmodell aus. `Proc3` besitzt dabei eine lokale Warteschlange `Garbage`. `Comp1` enthält den Prozeß `Proc1` mit seinen drei Children.

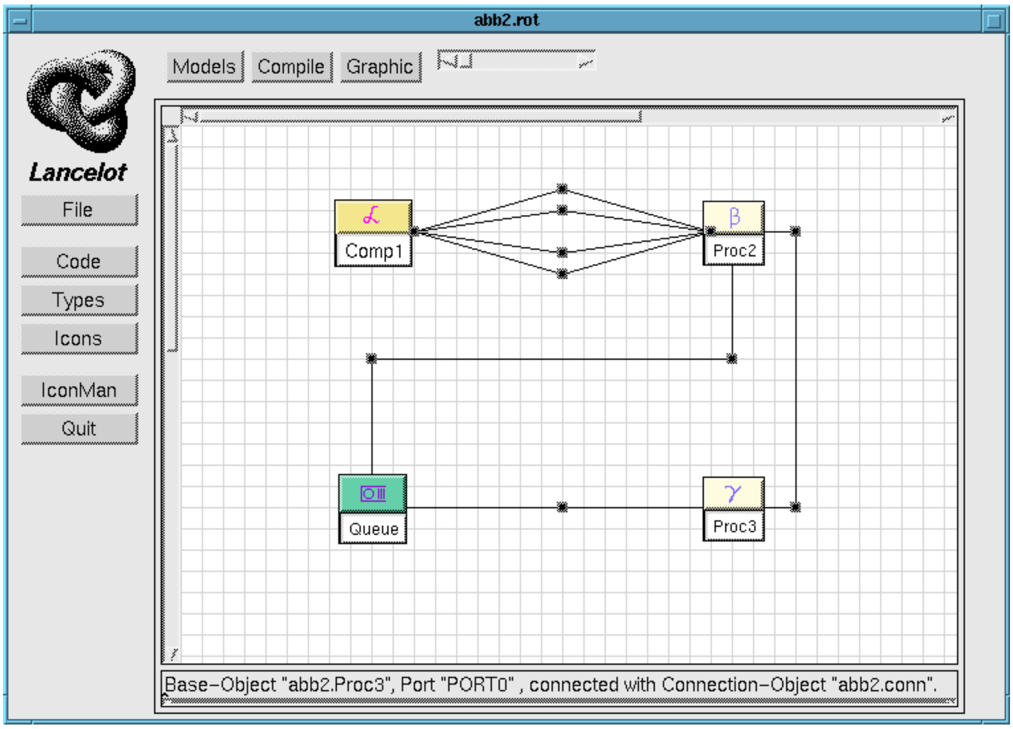
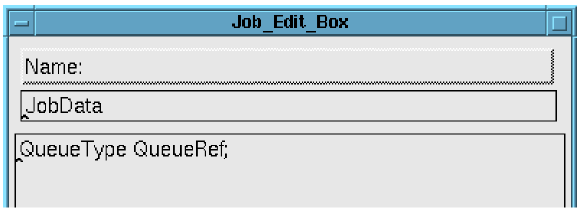


Abbildung 30: Einfügen von `Comp1` in das Modellbeispiel

Wir erweitern das Typobjekt `Jobdata` um eine Zeigervariable `QueueRef`, die eine Warteschlange adressieren kann:



Wir modifizieren den Programmcode von `Proc1` um folgende Zeilen:

```
PHASE(INIT)
 Job = NewJob(Icon1);
 if(index == 2)
     Job->QueueRef = Proc3->Garbage;
 else
     Job->QueueRef = Queue;
 Enter(Job, Proc2);
 RepeatAfter(0.3);
END
```

Wird die Phase von Child `Proc1[2]` ausgeführt, adressiert der neu erzeugte Auftrag `Job` in seiner Variable `QueueRef` die lokale Warteschlange von `Proc3`. Die übrigen Kinderprozesse, sowie der Vaterprozeß adressieren mit `QueueRef` die globale Warteschlange `Queue`.

Der Insert-Aufruf von `Proc2`, mit dem bisher die Aufträge in die Warteschlange `Queue` eingefügt worden sind, wird ersetzt durch:

```
PHASE(SEND)
   Insert(Job, Job->QueueRef);
   NextAfter(DELAY, 0.0);
END
```

Zur Übersetzungszeit ist somit nicht bekannt welche Warteschlange der `Insert`-Funktion als Parameter übergeben wird. Die Entscheidung in welche Warteschlange eingefügt wird, ist hier allein von der Belegung der Variablen `QueueRef` abhängig. Obwohl der aktuelle Wert des Parameters der `Insert`-Funktion erst zur Laufzeit bestimmt werden kann, wird in diesem Fall eine korrekte Animation ausgeführt. Für jeden Auftrag den `Proc2` von `Proc1[2]` erhält, zeigt `QueueRef` auf die lokale Warteschlange von `Proc3`; es erfolgt deshalb eine Auftragsbewegung von `Proc1[2]` nach `Proc2`. In den übrigen Fällen zeigt `QueueRef` auf die Warteschlange `Queue`, das demnach Ziel der Animationsbewegung ist.

6 Erweiterte Ausgabemöglichkeiten

Bisher wurden in den Ausführungen zum statischen und dynamischen Modellaufbau die Standard-Möglichkeiten zur Visualisierung und Ausgabe von Simulationsergebnissen vorgestellt. Lancelot bietet darüber hinaus weitere Ausgabemöglichkeiten, mit denen der Entwickler einer Simulation die programmierten Abläufe und Ergebnisse den individuellen Vorstellungen entsprechend darstellen kann.

6.1 Programmierung von Displays

Häufig möchte der Programmierer während der Simulation die zeitlich abhängige Wertebelegung einzelner Variablen innerhalb modellierter Objekte beobachten. Da die automatische Anzeige aller zu einem Objekt gehörenden Variablenwerte durch Lancelot sicherlich keine geeignete Lösung für diese Anforderung ist, wurden für Basisobjekte, komplexe Objekte und Verbindungsobjekte sogenannte **Buffer** eingeführt. Ein Buffer ist ein Darstellungsbereich eines Objektes, in dem beliebige Textausgaben des Simulationsprozesses ausgegeben werden können. Die Daten und die Art der Darstellung dieser Daten werden dabei durch den Simulationsprozeß und damit letztlich durch den Programmierer spezifiziert. Standardmäßig sind die Buffer der erzeugten Objekte innerhalb von Lancelot also leer. Erst nachdem ein Buffer durch den Simulationsprozeß initialisiert wurde, kann er auf dem Bildschirm dargestellt werden. Dies geschieht durch Anklicken der grafischen Repräsentation des zugehörigen Objekts mit der mittleren Maustaste. Durch erneutes Anklicken der grafischen Repräsentation wird die Darstellung wieder vom Bildschirm entfernt.

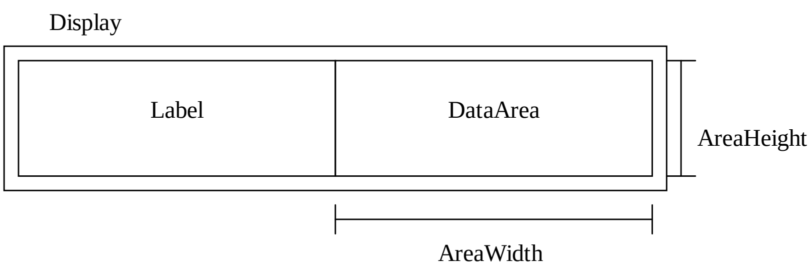


Abbildung 31: Geometrische Struktur eines Displays

Wie bereits erwähnt wurde, legt der Programmierer einer Simulation die Art der Darstellung der Daten innerhalb eines Buffers fest. Zu diesem Zweck können für einen Buffer mehrere

Displays definiert werden. Ein Display ist ein rechteckiger Darstellungsbereich, der sich in zwei nebeneinander liegende Unterbereiche gliedert. Der linke Bereich dient zur Beschriftung der darzustellenden Daten und wird deshalb **Label** genannt. Innerhalb des rechten Bereichs werden die eigentlichen Daten dargestellt; dieser Bereich wird **Data-Area** genannt. Die Abbildung 31 veranschaulicht die geometrische Struktur eines Displays.

Zur Definition eines Displays innerhalb des Simulationscodes wird die vordefinierte Datenstruktur `Xs_Display` verwendet:

```
struct Xs_Display{
        char* label;
        int   width;
        int   height;
        };
```

Um also für einen Buffer ein Display zu definieren, wird zunächst eine Struktur des Typs `Xs_Display` erzeugt. Anschließend wird `label` die Referenz auf eine Zeichenkette mit der Beschriftung des Labels zugewiesen. Die Variablen `width` und `height` geben die Breite und die Höhe der Data-Area an. Da es sich bei der Datenausgabe um Text handelt, geben `width` und `height` die Anzahl der Zeichen in der jeweiligen Dimension an.

Zur Initialisierung des Buffers dient die Funktion

```
int Xs_InitBuffer(char *pid, struct Xs_Display display[], int num);
```

Als erstes Argument erwartet die Funktion den Pfadnamen des Objekts (Zielobjekt), dessen Buffer initialisiert werden soll. Dieser Pfadname ist dem Programmierer normalerweise verborgen. Er kann jedoch über eine Zeichenkette referenziert werden. Die Zeichenkette besteht aus dem Objektnamen (Zielobjekt), der um den vollständigen Pfad in der Objekthierarchie erweitert wurde. Der vollständige Pfad in der Objekthierarchie ist die Konkatenation der einzelnen Objektnamen von komplexen Objekten, die in der Objekthierarchie auf dem Weg vom Root-Objekt zum Zielobjekt liegen. Ist das Zielobjekt mit dem Namen BaseA beispielsweise in einem komplexen Objekt mit dem Objektnamen ComplexB enthalten, das wiederum von einem komplexen Objekt ComplexA enthalten wird und seinerseits im Root-Objekt enthalten ist, so lautet die entsprechende Zeichenkette: **ComplexAComplexBBaseA**. Zur Verwendung beim Aufruf der Funktion `Xs_InitBuffer` wird diese Zeichenkette noch um den Zusatz **->PID** verlängert. Insgesamt wird der Pfadname also durch die Zeichenkette **ComplexAComplexBBaseA->PID** referenziert. Leider ist diese etwas komplizierte Referenzierung des Pfadnamens erforderlich, solange der Modellcompiler die Umsetzung der Buffer-Funktionen nicht automatisch durchführt.

Als zweites Argument erwartet die Funktion eine Referenz auf ein Array von Display-Strukturen. Wie bereits erwähnt wurde, kann ein Buffer mehrere Displays enthalten, die innerhalb des Buffers untereinander angeordnet werden. Die Breite des Buffers hängt also somit von der Breite der einzelnen Display ab. Das letzte Argument der Funktion bezeichnet die Anzahl der Displays, die initialisiert werden sollen. Im folgenden wird ein Programmfragment gezeigt, daß für ein Basisobjekt den Buffer mit drei Displays initialisiert:

```
#define no_of_displays   3
main()
...
struct Xs_Display display_array[no_of_displays];
...
/* Initialisierung der Display-Strukturen */
display_array[0].label = "Zustand:";
display_array[0].width = 20;
display_array[0].height = 1;
display_array[1].label = "Variable a =";
display_array[1].width = 4;
display_array[1].height = 1;
display_array[2].label = "Variable b =";
display_array[2].width = 4;
display_array[2].height = 1;
...
Xs_InitBuffer(ComplexAComplexBBaseA->PID, display_array,
no_of_displays);
...
```

Dieses Programmfragment erzeugt für das Basisobjekt BaseA in der obigen Beispielstruktur einen Buffer mit dem folgenden Aussehen:

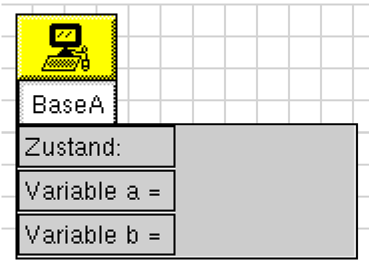


Abbildung 32: Buffer des Beispielprogramms

Die Funktion gibt den Wert 1 zurück, falls der Buffer korrekt initialisiert werden konnte. Der Wert 0 signalisiert einen Fehler, der während der Initialisierung aufgetreten ist. Falls der Rückgabewert -1 ist, wurde der Simulationsprozeß vor dem Aufruf der Funktion von Lancelot getrennt, so daß eine Initialisierung des Buffers nicht mehr möglich ist.

Nach der Initialisierung eines Buffers können nun Zeichenketten in die DataArea's der definierten Displays ausgegeben werden. Dazu dient die Funktion

```
Xs_PrintBuffer(char *pid, char *str[], int num);
```

Das erste Argument hat dabei die gleiche Bedeutung wie bei der Funktion `Xs_InitBuffer`. Als zweites Argument wird nun ein Array von Referenzen auf die darzustellenden Zeichenketten übergeben. Die Array-Größe sollte dabei mit der Anzahl der initialisierten Displays übereinstimmen. Die Zeichenketten werden ihrer Reihenfolge entsprechend den einzelnen Display zugeordnet und können eine beliebige Länge haben. Sie werden automatisch auf die bei der Initialisierung angegebene Größe reduziert. Bei Zeichenketten, die über mehrere Zeilen dargestellt werden sollen, dient die Zeichenfolge '\n' zur Kennzeichnung des Zeilenumbruchs.

Als drittes Argument benötigt die Funktion noch die Anzahl der übergebenen Zeichenketten. Nach der korrekten Ausführung der Funktion (Rückgabewert 1), sollten im Buffer des referenzierten Objekts, die übermittelten Zeichenketten dargestellt werden. Die folgenden Anweisungen beschreiben den oben initialisierten Buffer:

```
char *char_array[3];
char_array[0] = „ENABLED“;
char_array[1] = intchar(4);
```

```
char_array[2] = intchar(1);
Xs_PrintBuffer(ComplexAComplexBBaseA->PID, char_array; 3);
```

Dabei ist `intchar(int)` eine Funktion, die einen Integerwert in eine Zeichenkette umwandelt. Das Ergebnis sieht dann etwa so aus:

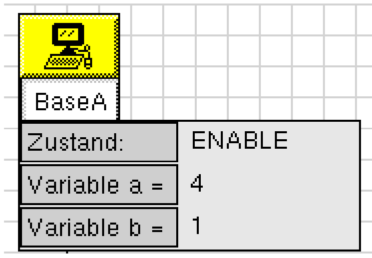


Abbildung 33: Ausgabe in den Buffer

Der Rückgabewert 0 signalisiert einen Fehler bei der Ausführung der Funktion. Der Wert -1 zeigt die Trennung von Simulationsprozeß und Lancelot an.

Häufig werden in einem Buffer ausschließlich Zahlenwerte (z.B. vom Typ `int` oder `double`) dargestellt. Um dem Programmierer den Aufwand zur Umwandlung der Zahlenwerte in Zeichenketten zu ersparen, werden zwei zusätzliche Funktionen zur Datenübergabe angeboten:

```
int Xs_PrintBufferInt(char *pid, int str[], int num);
```

Diese Funktion unterscheidet sich von der Funktion `Xs_PrintBuffer` lediglich dadurch, daß die Funktion ein Array auf Integer-Zahlen als zweites Argument erhält. Natürlich muß dieses Array vorher mit den darzustellenden Werten initialisiert werden.

```
int Xs_PrintBufferDouble(char *pid, double str[], int num, int point);
```

Auch diese Funktion besitzt die gleiche Funktionalität, wie die Funktion `Xs_PrintBuffer`, außer daß als zweites Argument ein Array des Datentyps `double` erwartet wird. Zusätzlich wird über das vierte Argument (`point`) die Anzahl der dargestellten Stellen hinter dem Komma festgelegt.

Der Inhalt eines Buffers bleibt so lange erhalten, bis er durch einen erneuten Initialisierungsaufruf oder eine weitere Print-Anweisung überschrieben wird, bzw. das referenzierte Objekt gelöscht wird. D.h., daß der Inhalt des Buffers auch nach der Terminierung des Simulationsprozesses erhalten bleibt.

Die Funktionen zur Programmierung der Displays werden durch die Funktionenbibliothek *artlib* zur Verfügung gestellt, die automatisch während der Übersetzung des Simulationsprozesses vom Modellcompiler in den Code eingebunden wird. In dieser Funktionenbibliothek befinden sich auch die Funktionen zur individuellen Visualisierung, die im nun folgenden Abschnitt vorgestellt werden.

6.2 Individuelle Visualisierung

Ein Simulationsprozeß kann seinen Ablauf über die grafische Benutzungsoberfläche Xartus auf dem Bildschirm visualisieren. Bei Lancelot fügt der Compiler Xcalibur während der Übersetzung des Simulationsmodells standardmäßig entsprechende Funktionen für die Visualisierung in den C-SIM-Code ein. Die Visualisierung findet dann innerhalb des vom Benutzer aufgebauten Simulationsmodells statt.

Darüber hinaus stehen dem Benutzer von Lancelot weitere Möglichkeiten zur anwendungsgerechten Gestaltung einer Visualisierung zur Verfügung. Diese Art der Visualisierung kann er innerhalb des C-SIM-Codes eines Simulationsmodells selbst programmieren. Dazu werden Funktionen der Bibliothek *artlib* verwendet (s.o.). Diese Funktionen ermöglicht es, aus dem Simulationsprozeß heraus mehrere Zeichenflächen innerhalb der Benutzungsoberfläche von Lancelot zu erzeugen. In einer Zeichenfläche, auch Canvas genannt, können dann Icon und Texte erzeugt und bewegt werden. Außerdem lassen sich auf einfache Art und Weise grafische Primitiven wie Linien, Punkte und Kreise erzeugen. Im folgenden werden nun die entsprechenden Funktionen vorgestellt und erläutert.

Der Rückgabewert -1 signalisiert bei allen Funktionen, daß der Simulationsprozeß von Lancelot getrennt wurde und eine Ausführung der aufgerufenen Funktion aus diesem Grund nicht mehr möglich ist. Der Rückgabewert 0 zeigt einen Fehler an, der bei der Ausführung der Funktion aufgetreten ist. Ein solcher Fehler kann beispielsweise auf ungültige Parameter beim Funktionsaufruf zurückzuführen sein.

```
Xs_Canvas Xs_CreateCanvas(int width, int height);
```

Diese Funktion erzeugt eine neue Zeichenfläche mit der Pixelbreite `width` und der Pixelhöhe `height`. Um die von `Xs_CreateCanvas` erzeugte Zeichenfläche in späteren Funktionsaufrufen referenzieren zu können, gibt die Funktion ein Handle vom Typ `Xs_Canvas` zurück. Das folgende Programmstück veranschaulicht exemplarisch die Erzeugung eines Canvas mit den Pixelausmaßen 200×200:

```
...
Xs_Canvas canvas; /* Definition einer Variable von Typ Xs_Canvas */
...
/* Erzeugen des Canvas und Zuweisen des zurückgegebenen Handles */
canvas = Xs_CreateCanvas(200, 200);
...
```

Die von `Xs_CreateCanvas` erzeugte Zeichenfläche wird noch nicht auf dem Bildschirm dargestellt. Dies geschieht durch die Funktion

```
int Xs_PopupCanvas(Xs_Canvas canvas_id);
```

Dabei muß für `canvas_id` ein gültiges Handle für einen vorher erzeugten Canvas übergeben werden. Im Fall der erfolgreichen Darstellung des Canvas wird der Wert 1 zurückgegeben, im Fehlerfall 0 (beispielsweise, wenn der Canvas vorher nicht erzeugt wurde).

```
int Xs_PopdownCanvas(Xs_Canvas canvas_id);
```

Durch den Aufruf dieser Funktion wird der durch `canvas_id` referenzierte und auf dem Bildschirm abgebildete Canvas, wieder vom Bildschirm entfernt. Die Daten des entsprechenden Canvas bleiben aber erhalten, so daß der Canvas zu einem späteren Zeitpunkt wieder mit `Xs_PopupCanvas` abgebildet werden kann. Der Rückgabewert 1 signalisiert die erfolgreiche Ausführung der Funktion; der Fehlerfall wird durch den Wert 0 angezeigt.

```
Xs_Icon Xs_CreateIcon(Xs_Canvas canvas_id, char* iconname, int foreground,
                      int background);
```

Mit dieser Funktion kann ein Icon erzeugt werden. `canvas_id` bezeichnet dabei den Canvas, dem das Icon zugeordnet werden soll. Mit dem Argument `iconname` wird der Name eines Iconobjekts im Icon-Formular des Root-Objekts spezifiziert. Über `foreground` und `background` können der Vordergrund- und Hintergrundfarbe des Icons festgelegt werden. Dabei stehen 14 verschiedene Farben zur Auswahl. Der Wert 0 steht für die Farbe weiß und 1 für schwarz. Auf die Farbgebung für die Werte 2 bis 14 kann der Benutzer von Lancelot über die Menüoption *Options/Colors* Einfluß nehmen. Falls das Icon erzeugt wurde, wird ein Handle vom Typ `Xs_Icon` zurückgegeben. Der Rückgabewert 0 zeigt einen Fehler an.

Das erzeugte Icon wird erst auf dem Bildschirm sichtbar, nachdem die folgende Funktion aufgerufen wurde

```
int Xs_PopupIcon(Xs_Canvas canvas_id, Xs_Icon icon, int x, int y);
```

Diese Funktion bildet das durch `icon` referenzierte Icon (Handle) mit der linken oberen Ecke an den Pixelkoordinaten `x` und `y` innerhalb des durch `canvas_id` referenzierten Canvas ab. Der Rückgabewert 1 signalisiert dabei die erfolgreiche Durchführung der Funktion.

```
int Xs_PopdownIcon(Xs_Canvas canvas_id, Xs_Icon icon);
```

Mit dieser Funktion kann ein dargestelltes Icon wieder vom Bildschirm entfernt werden. `canvas_id` referenziert dabei den entsprechenden Canvas und `icon` das Icon innerhalb dieses Canvas. Das Icon bleibt dabei erhalten und kann zu einem beliebigen Zeitpunkt erneut innerhalb des Canvas abgebildet werden. Nach der erfolgreichen Ausführung kehrt die Funktion mit dem Rückgabewert 1 zurück.

```
int Xs_RemoveIcon(Xs_Canvas canvas_id, Xs_Icon icon);
```

Durch den Aufruf dieser Funktion wird das durch `icon` referenzierte Icon in einem Canvas (`canvas_id`) gelöscht. Falls es auf dem Bildschirm abgebildet ist, wird zunächst `Xs_PopdownIcon` ausgeführt. Anschließend werden dann alle zu dem Icon gespeicherten Daten entfernt. Ein erneuter Darstellungsaufruf (`Xs_PopupIcon`) des Icon im folgenden erzeugt einen Fehler (Rückgabewert 0). Nachdem das Icon erfolgreich gelöscht wurde, gibt die Funktion den Wert 1 zurück.

```
int Xs_MoveIcon(Xs_Canvas canvas_id, Xs_Icon icon, int x1, int y1,
                int x2, int y2);
```

Der Aufruf dieser Funktion bewirkt eine Animation des Icon (`icon`) im Canvas (`canvas_id`) von der Bildschirmposition (`x1`,`y1`) zur Position (`x2`, `y2`). Die Animationsgeschwindigkeit kann dabei in gewohnter Weise mit dem Speedbar im Root-Formular gesteuert werden. Falls das Icon vor der Animation auf dem Bildschirm abgebildet war, so bleibt es dies auch nach der Animation. Falls es vor der Animation nicht abgebildet war, wird es nur für die Dauer der Animation auf dem Bildschirm erzeugt und anschließend wieder entfernt. Die erfolgreiche Ausführung der Animation wird durch den Rückgabewert 1 signalisiert.

```
Xs_Text Xs_CreateText(Xs_Canvas canvas_id, char* text, int foreground,
                      int background);
```

Diese Funktion kann zur Erzeugung eines Textes innerhalb eines Canvas (`canvas_id`) verwendet werden. `text` ist dabei eine Referenz auf die darzustellende Zeichenkette. Mögliche Zeilenumbrüche im Text werden durch die Zeichenfolge '\n' gekennzeichnet. `foreground` und `background` haben die gleiche Bedeutung wie bei der Funktion `Xs_CreateIcon`. Als Rückgabewert wird im Erfolgsfall ein Handle vom Typ `Xs_Text` zurückgegeben.

Der Text erscheint erst nach dem Aufruf von `Xs_PopupText` auf dem Bildschirm.

```
int Xs_PopupText(Xs_Canvas canvas_id, Xs_Text text_id, int x, int y);
```

Durch diese Funktion wird das durch `text_id` referenzierte Textobjekt (Handle) mit der linken oberen Ecke an den Pixelkoordinaten `x` und `y` innerhalb des durch `canvas_id` referenzierten Canvas dargestellt. Der Rückgabewert 1 signalisiert dabei die erfolgreiche Durchführung der Funktion.

```
int Xs_PopdownText(Xs_Canvas canvas_id, Xs_Text text_id);
```

Mit dieser Funktion kann ein innerhalb eines Canvas dargestelltes Textobjekt wieder vom Bildschirm entfernt werden. `canvas_id` referenziert dabei den entsprechenden Canvas und `text_id` das Textobjekt innerhalb dieses Canvas. Das Objekt bleibt dabei erhalten und kann zu einem beliebigen Zeitpunkt erneut innerhalb des Canvas abgebildet werden. Nach der erfolgreichen Ausführung kehrt die Funktion mit dem Rückgabewert 1 zurück.

```
int Xs_ChangeText(Xs_Canvas canvas_id, Xs_Text text_id, char* text);
```

Es ist möglich den Text eines Textobjekts (`text_id`) in einem Canvas (`canvas_id`) nach der Erzeugung zu ändern. `text` ist dabei eine Referenz auf die neue Zeichenkette. Der Rückgabewert 1 zeigt die erfolgreiche Änderung an.

```
int Xs_MoveText(Xs_Canvas canvas_id, Xs_Text text_id, int x1, int y1,
                int x2, int y2);
```

Auch ein Textobjekt (`text_id`) kann innerhalb des zugehörigen Canvas (`canvas_id`) von einer Bildschirmposition (`x1`,`y1`) zu einer anderen (`x2`,`y2`) bewegt werden. Die Funktionalität ist dabei vergleichbar mit der Funktion `Xs_MoveIcon`. Der Rückgabewert 1 signalisiert die erfolgreiche Ausführung der Animation.

```
int Xs_RemoveText(Xs_Canvas canvas_id, Xs_Text text_id);
```

Diese Funktion wird verwendet, um ein Textobjekt (`text_id`) aus dem zugehörigen Canvas (`canvas_id`) zu löschen. Eine mögliche Abbildung des Objekts auf dem Bildschirm wird vorher entfernt. Ein erneuter Abbildungsversuch des Objekts erzeugt einen Fehler (`Xs_PopupText`, Rückgabewert 0). Nach dem erfolgreichen Löschvorgang kehrt die Funktion mit dem Wert 1 zurück.

```
Xs_LineStyle Xs_CreateLineStyle(Xs_Canvas canvas_id, int width,
                                int colorindex);
```

Zum Zeichnen von Linien, Punkten und Kreisen muß zunächst das Aussehen dieser Grafik-Primitiven festgelegt werden. Dies geschieht mit der Funktion `Xs_CreateLineStyle`. Dabei kann für Linien und Kreise die Linienbreite über `width` gesetzt werden. Mit `colorindex` wird eine Farbe ausgewählt. Wie bereits bei `Xs_CreateIcon` beschrieben wurde, ist ein Farbwert zwischen 0 und 13 möglich. Als Rückgabewert liefert die Funktion einen Handle vom Typ `Xs_LineStyle` zurück. Diese Handle wird von den Funktionen zum Zeichnen der grafischen Primitiven benötigt.

```
int Xs_DrawPoint(Xs_Canvas canvas_id, Xs_LineStyle line_style, int x,
                 int y);
```

Die Funktion ermöglicht das Zeichnen eines einzelnen Pixels in einem Canvas (`canvas_id`) an der Position (`x`, `y`). Die Farbe des Pixels wird durch das Handle (`line_style`) auf ein entsprechendes LineStyle-Objekt festgelegt. Nach der erfolgreichen Ausführung kehrt die Funktion mit dem Wert 1 zurück.

```
int Xs_DrawLine(Xs_Canvas canvas_id, Xs_LineStyle line_style, int x1,
                 int y1, int x2, int y2);
```

Zum Zeichnen einer Linie innerhalb eines Canvas (`canvas_id`) werden dieser Funktion die Koordinaten des Anfangspunkts (`x1`,`y1`) und die Koordinaten des Endpunkts (`x2`,`y2`) der Linie übergeben. Das Zeichnen der Linie geschieht mit den im LineStyle-Objekt (`line_style`) festgelegten Attributen. Nach dem Zeichnen wird der Rückgabewert 1 zurückgegeben.

```
int Xs_DrawCircle(Xs_Canvas canvas_id, Xs_LineStyle line_style, int x1, int
                  y1, int width, int height, int angle1, int angle2);
```

Diese Funktion ermöglicht das Zeichnen von Kreis- und Ellipsenbögen in einem Canvas (`canvas_id`). Die Argumente `x1` und `y1` bilden dabei die Koordinaten der linken oberen Ecke des Rechtecks, das den zugehörigen Kreis oder die zugehörige Ellipse umschließt. `width` bezeichnet die Breite und `height` die Höhe dieses Rechtecks. Mit `angle1` wird der Anfangswinkel (in vierundsechzigstel Grad) des zu zeichnenden Bogens bestimmt. `anlgle2` bezeichnet den Winkel (in vierundsechzigstel Grad), in dem der Bogen gezeichnet werden soll (bei `angle1` begonnen). Wenn man `angle1` mit 0 und `angle2` mit 23040 (64x360) angibt, wird ein ganzer Kreis gezeichnet. Die Linienbreite und Farbe des Bogens wird durch das LineStyle-Objekt (`line_style`) festgelegt. Nach dem Zeichnen wird der Rückgabewert 1 zurückgegeben. Die folgende Abbildung veranschaulicht die Bedeutung der erläuterten Parameter:

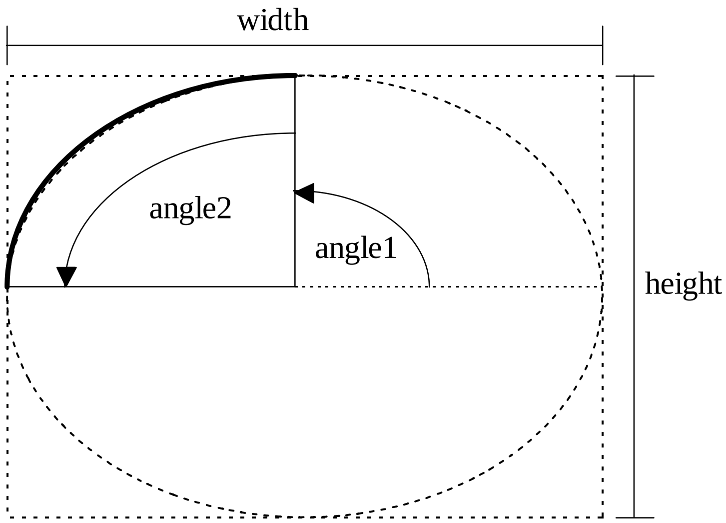


Abbildung 34: Die Definition eines Bogens

```
int Xs_ClearCanvas(Xs_Canvas canvas_id);
```

Gezeichnete Linien, Kreise und Punkte können mit dieser Funktion wieder aus einem Canvas (`canvas_id`) gelöscht werden. Die übrigen Objekte (Icon, Textobjekte) lassen sich mit den bereits vorgestellten Funktionen vom Bildschirm entfernen. Das erfolgreiche Löschen des Canvas-Inhalts wird mit dem Rückgabewert 1 quittiert.

7 Die Statistikkomponente XStat

Innerhalb von Simulationsprogrammen erlaubt Lancelot die Einbindung der Statistikkomponente XStat, die im Lancelot-Programm-Paket enthalten ist. Dabei handelt es sich um ein Programm zur graphischen Darstellung von statistischen Daten in dynamischen Zeitdiagrammen. Der Benutzer kann dabei die Darstellungsart und -form der Diagramme interaktiv über Menüs auswählen. So kann er die Daten als Balken-, Linien-, Punkt- oder Tortendiagramme in Kombination mit einer Reihe anderer Optionen anzeigen lassen. XStat verfügt über eine einfache Schnittstelle zur Datenübergabe in Form von Funktionen einer Funktionenbibliothek (`libxst.a`). Somit läßt sich XStat bequem in ein beliebiges C-Programm einbinden und von dort aus starten und benutzen. Innerhalb von Lancelot können diese Funktionen problemlos im C-SIM-Code des Simulationsmodells benutzt werden.

XStat wurde im Rahmen einer Studienarbeit in der Abteilung Rechnernetze & Telekommunikation im Fachbereich Informatik an der Universität Oldenburg entwickelt, und baut auf dem Programm XSVS (X-SimView), Version 2.0 von Th. Ono-Tesfaye auf. Die Darstellungsmöglichkeiten und -funktionen von XSVS wurden dabei erweitert. Die Anforderungen und Konzepte, die der Entwicklung von XStat zugrunde liegen, können in [Sterzenbach94] nachgelesen werden.

7.1 Begriffserläuterung

Zum besseren Verständnis der nachfolgenden Ausführungen werden nun einige Begriffe erläutert.

Grundlage eines Diagramms sind die dazu gehörenden Datenwerte (z.B. eine Linie im Diagramm entspricht einer Folge von Werten des Datentyps `double`). Diese Folge von Werten, die miteinander im Zusammenhang stehen, wird im folgenden als **Serie**; die Anzahl der Werte einer Serie als **Serienlänge** bezeichnet. Es ist möglich, XStat bis zu 20 verschiedene Serien zur gleichzeitigen Darstellung zu übergeben. Es wird daher auch oft von der **Serienanzahl** gesprochen. Diagramme können sehr unübersichtlich werden, wenn bei einer großen Serienlänge alle Werte gleichzeitig dargestellt werden. XStat ermöglicht daher auch die Darstellung von Teilintervallen der gespeicherten Daten. Dies definiert den Begriff der **Anzahl der dargestellten Werte**. Schließlich erlaubt XStat die tupelweise Übergabe von Datenwerten. Diese Tupel können einzeln an XStat weitergeleitet werden oder auch zunächst innerhalb der Funktionen von XStat gesammelt und dann zusammen mit einer Anzahl weiterer Tupel an XStat übergeben werden. Die Anzahl der Werte, die dabei in einer Übertragung an XStat gesendet werden, wird durch den **Batch-Wert** angegeben.

7.2 Aufbau

Der Aufruf von XStat aus einem Prozeß (Applikationsprozeß) heraus erzeugt einen neuen XStat-Prozeß (Darstellungsprozeß). Anschließend wird zwischen den beiden Prozessen eine TCP-Kommunikationsverbindung aufgebaut, über die Daten ausgetauscht werden können. Die folgende Abbildung verdeutlicht diesen Aufbau:

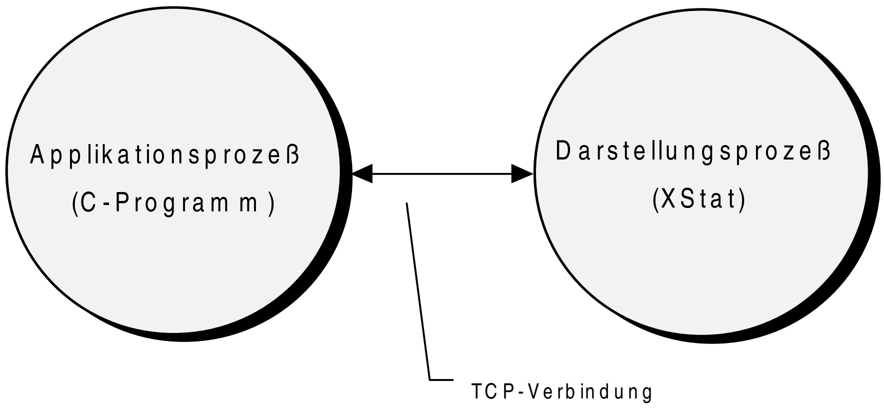


Abbildung 35: Applikations- und Darstellungsprozeß

7.3 Aufrufsyntax

Im folgenden wird nun beschrieben, wie XStat aus einem Applikationsprozeß heraus gestartet werden kann und wie die Datenübergabe geschieht. Dabei werden grundsätzlich zwei Anwendungsweisen von XStat unterschieden: Die konstante Datenübergabe und die dynamische Datenübergabe.

7.4 Konstante Datenübergabe

Die Verwendung der konstanten Datenübergabe entspricht der Anwendungsweise von XSVS. Dabei wird zunächst der Darstellungsprozeß generiert. Anhand verschiedener Funktionen kann dieser dann initialisiert werden (Beschriftungen, Legende, usw.). Anschließend werden die statistischen Daten dann als Array von Werten des Datentyps `double` an XStat übertragen und dort dargestellt. Erfolgt dabei zu einem späteren Zeitpunkt eine neue Übertragung von

Daten, werden die vorher übertragenen Daten von den neueren überschrieben und gehen verloren.

Die Initialisierung eines neuen Darstellungsprozesses für die konstante Datenübergabe erfolgt mit der Funktion

```
void* XSTInitCon(char* hostname, int typ).
```

Als ersten Parameter (`hostname`) benötigt die Funktion den Hostname des Rechers, auf dem der Darstellungsprozeß innerhalb eines Netzes gestartet werden soll. Der zweite Parameter (`typ`) legt den Darstellungstyp fest, mit dem die Darstellung der Daten beginnen soll. Dieser Wert kann zu einem späteren Zeitpunkt über die Menüs von XStat verändert werden. Als Rückgabewert liefert die Funktion eine Referenz (Handle) auf eine Datenstruktur, die den erzeugten XStat-Prozeß bezeichnet. Jede andere XST-Funktion benötigt diese Referenz als Argument.

Mit der Funktion

```
int XSTTitle(void* desc, char* title)
```

kann eine Titelbeschriftung des Diagramms festgelegt werden. Für `desc` wird dabei eine gültige Referenz (Handle) auf einen vorher mit `XSTInit` erzeugten Darstellungsprozeß übergeben. `title` bezeichnet eine Zeichenkette mit der gewünschten Beschriftung.

Die Funktionen

```
int XSTXLabel(void* desc, char* xlabel)
```

und

```
int XSTYLabel(void* desc, char* ylabel)
```

sind analog zu `XSTTitle` zu verwenden. `xlabel` und `ylabel` bezeichnen dabei allerdings die X- bzw. Y-Achsenbeschriftung des Diagramms.

Jedem Diagramm kann eine Legendenbeschriftung zugeordnet werden. Dies geschieht mit der Funktion

```
int XSTLegende(void* desc, char* legende).
```

Als erstes Element erhält die Funktion wieder eine Referenz auf einen XStat-Prozeß. `legende` bezeichnet eine Zeichenkette, in der die einzelnen Beschriftungen entsprechend der

Serienanzahl, jeweils durch Kommata getrennt, enthalten sind. Werden beispielsweise vier Serien gleichzeitig dargestellt, sollte die Zeichenkette folgendes Format besitzen:

„<Beschriftung für die 1.Serie>**,**<Beschriftung für die 2.Serie>**,**<Beschriftung für die 3.Serie>**,**<Beschriftung für die 4.Serie>“.

Nachdem die Datenübergabe an einen Darstellungsprozeß abgeschlossen ist, sollte die zugehörige Kommunikationsverbindung mit der Funktion

```
int XSTClose(void* desc)
```

geschlossen werden. Als Parameter benötigt die Funktion die entsprechende Referenz auf den Darstellungsprozeß (Handle).

Alle Funktionen (außer `XSTInit`) liefern bei erfolgreichem Abschluß ihres Aufrufs den Wert 1 zurück. Als Fehlerwert wird -1 zurückgegeben.

7.5 Dynamische Datenübergabe

Bei der dynamischen Datenübergabe wird der Darstellungsprozeß zunächst gestartet und initialisiert. Dabei muß die Seriengröße und die Serienanzahl festgelegt werden. Der Darstellungsprozeß reserviert entsprechend der festgelegten Werte ein Array des Datentyps `double` mit der Größe „Seriengröße*Serienanzahl“ Anschließend können die Datenwerte tupelweise übergeben werden. Die empfangenen Daten werden von XStat entsprechend der Reihenfolge ihres Eintreffens in dem reservierten `double`-Array abgelegt.

Alle Datenwerte bleiben also bis zum Programmende innerhalb von XStat gespeichert. Dabei kann bei dem Initialisierungsaufruf die Anzahl der gleichzeitig dargestellten Werte festgelegt werden. Durch verschiedene Scrollmechanismen können dann die dargestellten Werte gegen Datenwerte, die zu früheren oder späteren Zeitpunkten gehören, ausgetauscht werden (siehe Abschnitt „Bedienung“).

Ein XStat-Darstellungsprozeß mit dynamischer Datenübergabe wird durch den Aufruf von

```
void* XSTInit(    char* hostname,

                  int         typ,
                  int         series_size,
                  int         no_of_series,
                  int         batch,
```

```
int        showed_values,
char*      title,
char*      xlabel,
char*      ylabel,
char*      legende)
```

gestartet und initialisiert. Der erste Parameter (`hostname`) bezeichnet dabei den Hostname eines Rechners, auf dem der Darstellungsprozeß innerhalb eines Netzwerks gestartet werden soll. Mit `typ` wird der Diagrammtyp festgelegt, mit dem die Darstellung beginnen soll. Dieser Typ kann jederzeit über die Menüs von XStat verändert werden. Zur Verfügung stehen die Diagrammarten:

`BAR`	Balken-Diagramm
`LINE`	Linien-Diagramm
`DOTS`	Punkt-Diagramm
`DOTSLINE`	Punkt-Linien-Diagramm
`MINMAX`	Minimum-Maximum-Diagramm
`BARVALUE`	Balkendiagramm mit Werten
`PIE`	Tortendiagramm

Zusätzlich kann ein Diagrammtyp durch logische Bitverknüpfung (Konjunktion, „|“) mit den später beschriebenen Darstellungsoptionen erweitert werden. Die folgende Tabelle zeigt die möglichen Kombinationen der einzelnen Diagrammtypen mit den verschiedenen Optionen:

	BAR	BARVALUE	LINE	DOTS	DOTSLINE	MINMAX	PIE
LOGSCALE	X	X	X	X	X	X	
SEPARATEBARS	X	X					
HORIZGRID	X	X	X	X	X	X	
VERTGRID			X	X	X	X	
FIRSTSERIESISAXIS	X	X	X	X	X	X	
ELLIPSE							X
LEGENDE	X	X	X	X	X	X	X

Für series_size muß die Serienlänge eingesetzt werden und no_of_series muß die Anzahl der Serien angeben. Die Daten werden tupelweise an den Darstellungsprozeß übergeben. Bei einer Anzahl von $n \in \mathbf{N}$ Serien und einer Serienlänge von $m \in \mathbf{N}$ ist ein Tupel als (x. Wert der 1. Serie, x. Wert der 2. Serie, ..., x. Wert der n. Serie) mit $x \in \mathbf{N}$ und $0 \leq x < m$ definiert. Durch batch wird die Anzahl der Tupel festgelegt, die jeweils gleichzeitig an XStat gesendet werden. Die einzelnen durch XSTData übergebenen Tupel werden also so lange zwischengespeichert, bis der Batchwert erreicht ist. Ein negativer Wert, oder ein Wert größer als die Serienlänge, ist dabei nicht sinnvoll und deshalb auch nicht zulässig. Über shown_values besteht die Möglichkeit, auf die Anzahl der gleichzeitig dargestellten Werte Einfluß zu nehmen. Auch hier ist ein negativer Wert, oder ein Wert größer als die Serienlänge, nicht sinnvoll und ebenfalls nicht zulässig. Beim Aufruf von XSTInit kann bereits eine Beschriftung des Diagramms festgelegt werden. Dabei kann für title eine Zeichenkette mit der Titelbeschriftung und für xlabel bzw. ylabel eine Zeichenkette mit der X- bzw. Y-Achsenbeschriftung übergeben werden. Gerade bei mehreren dargestellten Serien kann die Übersichtlichkeit gesteigert werden, wenn das Diagramm um eine Legende erweitert wird. Zur Definition einer Legende kann das Argument legendstring verwendet werden. Dazu wird eine Zeichenkette übergeben, in der die Bezeichnungen der einzelnen Serien, in derselben Reihenfolge in der auch die Datenübergabe geschieht, durch Kommata getrennt, enthalten sind. title, xlabel, ylabel und legendstring können auch NULL enthalten, um eine entsprechende Beschriftung zu unterbinden.

Die Funktion XSTInit liefert eine Referenz auf den erzeugten Darstellungsprozeß zurück. Dieser Wert wird für den Aufruf der Funktionen XSTData und XSTClose benötigt, um den Darstellungsprozeß zu bezeichnen. Hinter diesem Rückgabewert verbirgt sich die Möglichkeit, aus einem Applikationsprozeß heraus, beliebig viele voneinander unabhängige XStat-Prozesse zu starten, und Daten an diese zu übergeben.

Die Datenübergabe an einen XStat-Prozeß geschieht durch den Aufruf von:

```
XSTData(
      void*        handle,
      double       *data_tupel);
```

handle bezeichnet dabei einen bestimmten XStat-Prozeß und muß der Rückgabewert eines XInit-Aufrufs sein. data_tupel ist ein Zeiger auf ein double-Array der Größe no_of_series und sollte wie folgt belegt sein:

(x. Wert der 1. Serie, x. Wert der 2. Serie, ..., x. Wert der no_of_series. Serie) mit $x \in \mathbf{N}$ und $0 \leq x <$ series_size.

Ist die Anzahl der bereits übertragenen Werte gleich der Serienlänge, werden alle weiteren übergebenen Werte von XStat nicht mehr berücksichtigt. Für den Darstellungsprozeß ist die Datenübertragung mit dem Erreichen der Serienlänge abgeschlossen.

Nachdem die Datenübergabe an XStat abgeschlossen ist, muß die Kommunikationsverbindung vom Applikationsprozeß durch den Aufruf der Funktion

```
XSTClose(void*           handle);
```

geschlossen werden. handle bezeichnet dabei wieder eine durch einen `XSTInit`-Aufruf zurückgelieferte Referenz. Natürlich darf handle anschließend in einem `XSTData`- oder `XSTClose`-Aufruf nicht mehr verwendet werden. Nach dem Schließen der Kommunikationsverbindung laufen beide Prozesse vollständig unabhängig voneinander ab. Die Terminierung des Applikationsprozesses nach dem Aufruf von `XSTClose` ist für den Darstellungsprozeß bedeutungslos. Der XStat-Prozeß wird so lange ausgeführt, bis er durch die Auswahl von *Quit* im Menü *Graphic* terminiert wird.

Auch bei der dynamischen Datenübergabe können die Beschriftungen des Diagramms zu einem späteren Zeitpunkt mit Hilfe der vorgestellten Funktionen `XSTTitle`, `XSTXLabel`, `XSTYLabel` und `XSTLegende` verändert werden.

7.6 Bedienung und Menüsteuerung

Nachdem ein XStat-Prozeß gestartet und ein Fenster auf dem Bildschirm geöffnet wurde, sollten nach einer gewissen Zeitspanne (abhängig vom Batch-Wert und von der Übertragungsrate) die ersten Datenwerte auf dem Bildschirm dargestellt werden. Die folgende Abbildung 36 zeigt ein mögliches Aussehen des XStat-Fensters.

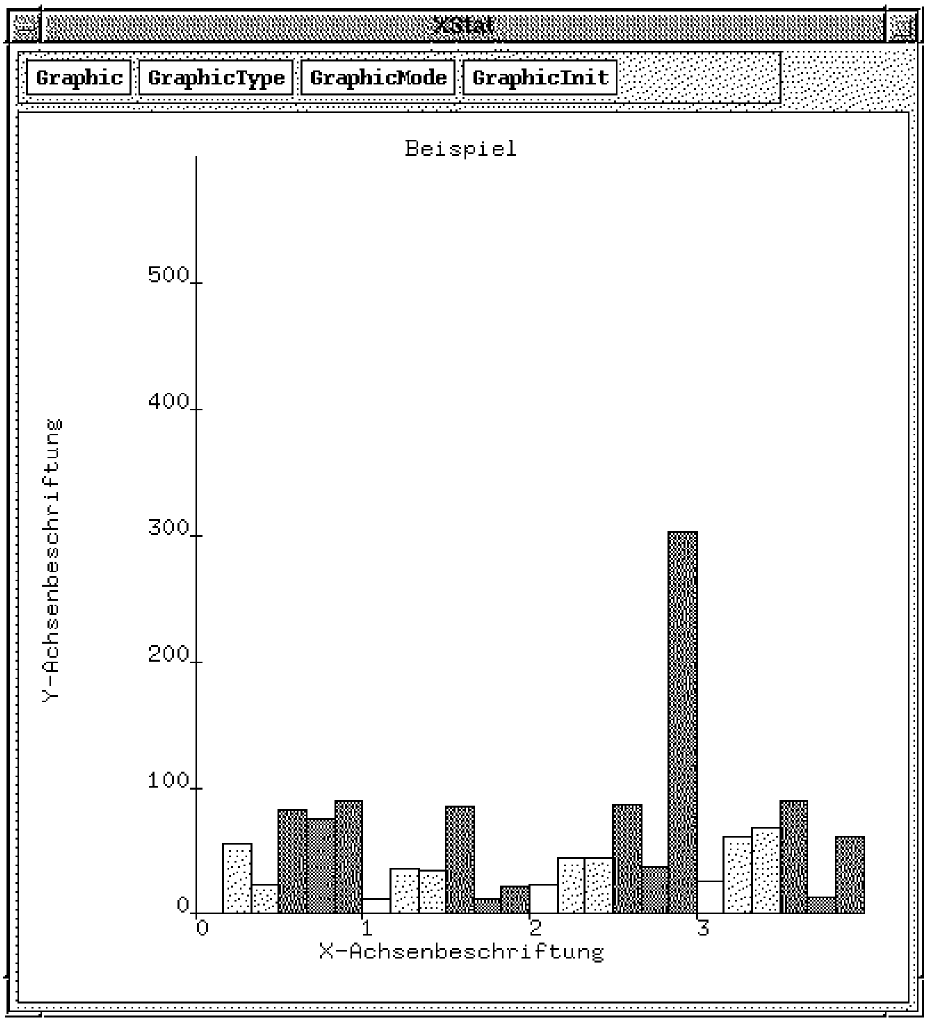


Abbildung 36: Mögliches Aussehen des XStat-Windows

Solange noch Daten übertragen werden, erscheint in der Schalterfläche der *RUN*-Button (in Abbildung 36 nicht sichtbar). Zusätzlich kann man die laufende Erweiterung des Diagramms durch die Darstellung neuer, übermittelter Werte beobachten, falls der rechte Fensterrand noch nicht erreicht bzw. der *RUN*-Button (s.u.) betätigt wurde.

Die Schalterfläche besteht aus den vier permanent dargestellten Menübuttons *Graphic*, *GraphicType*, *GraphicMode* und *GraphicInit*. Daneben erscheint der *RUN*-Button, falls die Datenübergabe noch nicht abgeschlossen ist. Bei der Auswahl des Eintrags *PIE* im Menü *GraphicType* ist die Anzeige noch um die Buttons << und >> erweitert (Abbildung 37).

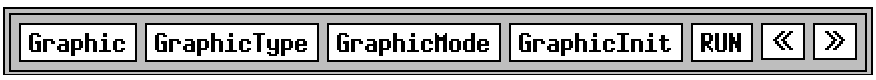


Abbildung 37: Schalterfläche

7.6.1 Der *Graphic*-Button

Bei einem Klick auf den *Graphic*-Button mit der linken Maustaste erscheint ein Menü mit dem Menüpunkt *Quit*. Dieser Menüpunkt erlaubt das Beenden von XStat, wobei der Applikationsprozeß durchaus noch weiter ablaufen und Werte an XStat übertragen kann.

7.6.2 Der *GraphicType*-Button

Bei einem Klick auf den *GraphicType*-Button mit der linken Maustaste erscheint eine Menüauswahl mit den verschiedenen Diagrammarten. Bei der Auswahl von *BAR* werden die Daten als Balkendiagramm dargestellt, und zwar tupelweise (s.o.). Bei *LINE* erfolgt eine Darstellung der Daten als Liniendiagramm und bei *DOTS* werden die Daten durch verschiedenartige Punkte repräsentiert. *DOTSLINE* ist eine Mischung aus *LINE* und *DOTS*. Beim Typ *MINMAX* wird aus jedem Tupel der minimale und der maximale Wert bestimmt und durch die jeweilige Punktdarstellung im Diagramm dargestellt. *BARVALUE* entspricht *BAR*, wobei die Darstellung noch um die jeweiligen numerischen Werte ergänzt wird. Bei *PIE* kann man die einzelnen Tupel, dargestellt als Tortendiagramm, betrachten.

7.6.3 Der *GraphicMode*-Button

Ein Klick auf den *GraphicMode*-Button mit der linken Maustaste führt in ein Menü mit verschiedenen Darstellungsoptionen. Dabei sind je nach Wahl der Diagrammart einige dieser Optionen von vornherein unsensibel, da sie im Zusammenhang mit der gewählten Diagrammart nicht sinnvoll sind. Eine Auswahl im *GrapicMode*-Menü wird mit dem X-Logo markiert, d.h. nach der Auswahl erscheint links vor dem Menüpunkt das X-Logo.

Bei der Menüauswahl *LOGSCALE* werden die Daten logarithmisch-skaliert ausgegeben. Im Falle von Nullwerten kann der Wert des niedrigsten auf der y-Achse dargestellten Exponenten bei Bedarf über das *GrapicInit*-Menü verändert werden. Das Eingabe-Intervall wird in der Dialogbox angezeigt. Allerdings wird bereits beim Empfang der Daten durch XStat eine Überprüfung der Werte durchgeführt und ein entsprechender Exponent für die darzustellenden Werte ermittelt, d.h. gewählt wird der größte Exponent n, für den gilt: $0 < 10^{n} < m$, wobei $n \in \mathbf{Z}$ mit $-10 \leq n \leq 7$ und m der kleinster Datenwert mit $m > 0$.

Die Option *SEPARATEBARS* ermöglicht es dem Benutzer, bei gewähltem Diagrammtyp *BAR* oder *BARVALUE*, die einzelnen Datentupel durch einen Freiraum getrennt anzeigen zu lassen, so daß ein Datentupel im Diagramm als zusammenhängende Gruppe von Balken erscheint. *HORIZGRID* und *VERTGRID* sorgen dafür, daß gemäß der Intervalleinteilung der Achsen durchgezogene horizontale und vertikale Linien das Ablesen erleichtern. Bei der

Menüauswahl *FIRSTSERIESISAXIS* werden die Werte der 1. Datenserie des Diagramms zur Beschriftung der x-Achse benutzt und nicht mehr im Diagramm dargestellt. *ELLIPSE* ist nur über den Diagrammtyp *PIE* zugänglich und sorgt für ein Tortendiagramm in dreidimensionaler Darstellung.

Die Auswahl *LEGENDE* erweitert die jeweilige Darstellung um eine Legende. Die Legende kann durch Drücken der linken Maustaste innerhalb des zugehörigen Rahmens und Verschieben des Mauszeigers auf eine neue Bildschirmposition, neu positioniert werden.

Bei der Auswahl eines neuen Diagrammtyps bleiben die gewählten Darstellungsoptionen erhalten, es sei denn, sie werden von dem gewählten Typ nicht unterstützt.

7.6.4 Der *GraphicInit*-Button

Hinter dem *GraphicInit*-Button verbirgt sich eine Menüauswahl, mit der grundsätzliche Einstellungen vorgenommen werden können.

Die Menüauswahl *Title* erlaubt dem Benutzer den Diagrammtitel zu ändern. Dazu erscheint nach der Auswahl durch den Mauszeiger eine Dialogbox, in der die Änderungen durchgeführt werden können. Zum Abschluß muß dann der *DialogDone*-Button betätigt werden, um die Änderungen zu übernehmen. Während sich der Benutzer im Dialogmodus befindet, sind alle anderen Menübuttons in der Schalterfläche unsensibel. Das Ändern der X- und Y-Achsenbeschriftung funktioniert analog über Auswahl von *X-Axis* und *Y-Axis*.

Die Auswahl von *Shown_values* ermöglicht die Veränderung der Anzahl der gleichzeitig dargestellten Werte. Liegt der eingegebene Wert außerhalb des Intervalls [1, `series_size`], wird der vorher gehende Wert beibehalten. Dieser Menüpunkt birgt einige Vorteile in sich, so kann sich der Benutzer zu Beginn eines XStat-Prozesses zunächst einmal einen Gesamtüberblick über die Werteverteilung verschaffen, indem er den Wert unter *Shown_values* beispielsweise auf die Seriengröße setzt (dies kann auch bereits bei dem `XSTInit`-Aufruf geschehen). Anschließend kann er sich dann durch Herabsetzen des *Shown_values*-Wertes einzelne Ausschnitte genauer anschauen.

Batch kann zum Ändern des Batch-Wertes benutzt werden. Dies ist z.B. dann sinnvoll, wenn der Benutzer nach dem Start des XStat-Prozesses merkt, daß sein in dem `XInit`-Aufruf festgelegter Wert zu hoch oder zu niedrig gewählt war. Als Eingabe sind Werte aus dem Intervall [1, `series_size`] zulässig. Die Veränderung des Batch-Wertes kann unter Umständen einen starken Einfluß auf die Bedienbarkeit des Programms haben: Ist der Batch-Wert sehr klein eingestellt, und sendet der Applikationsprozeß sehr kurz hintereinander seine Daten, so wird auch der Ablauf von XStat sehr häufig von eintreffenden

Kommunikationsanforderungen unterbrochen. XStat arbeitet ereignisorientiert, d.h eine solche Anforderung erzeugt ein Event, das in eine Event-Queue eingetragen wird. Bei Abarbeitung dieses Events wird dann zur Auswertung der Kommunikation verzweigt. Dieses Event befindet sich in derselben Queue, in die z.B. auch die Maus- und Tastatur-Events eingetragen werden. Wird nun die Event-Queue mit den Events von Kommunikationsanforderungen 'überschwemmt', kann es zu einer starken Verzögerung kommen, bis ein Mausklick ausgewertet wird.

Es ist daher ratsam, den Batch-Wert zunächst einmal etwas höher einzustellen und ihn dann später über das *GraphicInit*-Menü herabzusetzen. Ein Batch-Wert von eins ist nur dann ratsam, wenn sichergestellt werden kann, daß die Daten vom aufrufenden Prozeß mit ausreichend großem zeitlichen Abstand übergeben werden, so daß dem XStat-Prozeß genügend Zeit zum Auswerten der Maus- und Tastaturevents bleibt. Eine Veränderung des Batch-Wertes nach dem Erlöschen des *RUN*-Buttons ist nicht mehr möglich, da die Datenübertragung abgeschlossen ist.

Mit der Menüauswahl *Lowest_Log* ist es möglich, den niedrigsten Exponenten bei der Darstellungsoption *LOGSCALE* zu verändern, falls Nullwerte in den Daten enthalten sind.

Colors führt zu einem Fenster zur Farbeinstellung und wird in einem der nächsten Absätze näher beschrieben.

Die Auswahl *Scroll-Width* dient schließlich zum Einstellen der Scrollweite beim automatischen Scrollen. Hierbei ist eine Prozentzahl anzugeben. Der Default-Wert liegt bei 50 und bedeutet, daß beim automatischen Scrollen um 50% des angezeigten Intervalls verschoben wird. Die Verwendung des Scroll-Mechanismus wird später noch genauer beschrieben.

7.6.5 Der *RUN*-Button

Wie bereits erwähnt wurde, erscheint der *RUN*-Button in der Schalterfläche nur solange, bis alle angekündigten Daten übergeben wurden, bzw. bis `XSTClose` aufgerufen wurde. Danach erlischt er. Während der Datenübertragung kann man XStat durch Anklicken des *RUN*-Buttons mit der linken Maustaste dazu veranlassen, immer das Ende der Datenserien anzuzeigen. Sei $n \in \mathbf{N}$ die Ordinalzahl des zuletzt übertragenen Tupels, dann wird das Intervall [n - `shown_values`, n] angezeigt, vorausgesetzt (n - `shown_values`) ≥ 0. Bei einer neuen Datenübergabe wird das Diagramm dann entsprechend aktualisiert.

7.6.6 Farben einstellen

Nach der Menüauswahl *Colors* im *GraphicInit*-Menü öffnet XStat ein neues Fenster, in dem auf der linken Seite alle verfügbaren Farben des Programms als 16×8 Matrix dargestellt sind. Die Matrix besteht aus Farbkästchen und wird im folgenden Farbauswahl genannt. Daneben befindet sich rechts die sogenannte Farbpalette, das sind einzelne Farbkästchen mit den zur Zeit in den Diagrammen verwendeten Farben. Die Anzahl der Farben der Farbpalette hängt von der Anzahl der dargestellten Serien ab. Das Verfahren zur Farbeinstellung ist nun denkbar einfach. Man wählt zunächst eine Farbe aus der Farbauswahl durch einen Klick auf die linke Maustaste aus. Dabei wird das betreffende Kästchen optisch gekennzeichnet. Dann bewegt man den Mauszeiger zu einem Kästchen aus der Farbpalette, dessen Farbe man gegen die gewählte ersetzen möchte. Nach einem weiteren Klick auf die linke Maustaste werden die zwei Farben einfach vertauscht. Gleichzeitig kann man in dem angezeigten Diagramm die Wirkung dieser Aktion beobachten. Durch das Konzept des Vertauschens können unterschiedliche Serien nur durch unterschiedliche Farben gekennzeichnet werden, wodurch die Unterscheidbarkeit der Seriendaten gewährleistet bleibt.

Sind alle Farben zur Zufriedenheit eingestellt, kann das Farbfenster mit Hilfe des *Quit*-Buttons wieder geschlossen werden.

7.6.7 Maussteuerung und Scrollen

Wie bereits oben erwähnt wurde, besteht die Möglichkeit, aus Datenserien nur ein gewisses Teilintervall anzeigen zu lassen (*Shown_values* unter *GraphicInit*). Dies wirft allerdings die Frage auf, wie der Benutzer zur Ansicht der Daten gelangen kann, die gerade außerhalb des angezeigten Intervalls liegen. Dazu bietet XStat verschiedene Scroll-Funktionen an. Um zum Ende der Datenserien zu gelangen, klickt man mit der linken Maustaste den Randbereich rechts der x-Achse an. Für den Anfang der Datenserien gilt Entsprechendes mit dem linken Randbereich.

Darüber hinaus gibt es die Möglichkeit des manuellen Scrollens. Dabei führt man den Mauszeiger zu einem Punkt (Startpunkt) innerhalb des Diagramms im Bereich der x-Achse, drückt die linke Maustaste und verschiebt den Mauszeiger mit gedrückter Maustaste an eine neue Position (Endpunkt) im Bereich der x-Achse, links oder rechts des Startpunktes. Die Maustaste bleibt dabei solange gedrückt, bis sich der Mauszeiger an der gewünschten Position befindet. Die Bewegung des Mauszeigers wird durch eine vertikale Linie verfolgt. Durch Loslassen der Maustaste setzt man das Scroll-Intervall fest. Das Scroll-Intervall gibt die Anzahl der Werte an, um die das Teilintervall der auf dem Bildschirm dargestellten Werte verschoben wird. Liegt der Endpunkt des Intervalls links neben dem Startpunkt, wird in

Richtung Ende der Datenserien gescrollt; liegt er rechts neben dem Startpunkt, in Richtung Anfang.

Zieht man den Mauszeiger nach Setzen des Startpunktes aus dem Bereich der x-Achse hinein in den linken oder rechten Randbereich, wird das automatische Scrollen ausgelöst. Dabei wird das Diagramm automatisch durch das Fenster von XStat gescrollt. Das verwendete Scroll-Intervall kann durch das Verändern der Scrollweite (*Scroll_Width* unter *GraphicInit*) eingestellt werden. Je nachdem, ob der Mauszeiger in den linken oder rechten Randbereich bewegt wurde, wird in Richtung Ende oder Anfang der Datenserien gescrollt.

Falls die Option *Legende* ausgewählt wurde, kann die dargestellte Legende mit Hilfe der Maus innerhalb des Windows von XStat verschoben werden. Dazu drückt man innerhalb der Legende die linke Maustaste und verschiebt dann den Mauszeiger auf eine neue Position. Nach dem Loslassen der Taste wird die Legende mit der linken oberen Ecke an der gewählten Stelle neu positioniert.

Im folgenden soll nun noch auf einige Besonderheiten der Diagrammart `PIE` hingewiesen werden. Bei dieser Diagrammart wird jeweils nur ein Datentupel dargestellt. Eine Scroll-Funktion wie bei den übrigen Diagrammarten ist daher nicht sinnvoll und wird auch nicht unterstützt. Stattdessen ist es möglich, die einzelnen Datentupel mit Hilfe der `<<`- und `>>`-Tasten „durchzuschalten“. Außerdem verfügt die Diagrammart noch über einige außerordentliche Mausfunktionen. So besteht die Möglichkeit, ein Tortendiagramm auf dem Bildschirm zu verschieben. Dies geschieht, indem man den Mauszeiger zum Mittelpunkt des Tortendiagramms bewegt, die linke Maustaste drückt, den Mauszeiger an eine neue Position bewegt und dann die Maustaste wieder losläßt. Das Diagramm erscheint an der neuen Position. Weiter besteht die Möglichkeit, den Radius des Tortendiagramms zu verändern. Dazu bewegt man den Mauszeiger zu einer beliebigen Stelle des Diagrammrandes (Ausnahme: `ELLIPSE`), drückt wiederum die linke Maustaste, verschiebt den Mauszeiger und läßt dann die Taste in gewünschter Entfernung vom Mittelpunkt des Diagramms wieder los. Gewisse minimale und maximale Werte dürfen dabei nicht unter- bzw. überschritten werden. Bei der Option `ELLIPSE` hat das Anklicken des Randes nur dann eine Wirkung, wenn der Rand an seiner äußersten linken oder rechten Ausdehnung angeklickt wird.

8 Anbindung einer Applikation

Es wurde bereits darauf hingewiesen, daß Xartus zunächst unabhängig von einer konkreten Applikation entwickelt wurde. Erst später wurde dann Xcalibur, der Compiler für C-SIM, als Applikation entwickelt. Beide Programme wurden dann im Simulationssystem Lancelot zusammengefügt.

Für die Anbindung einer Applikation stellt Xartus die Funktionenbibliothek *artlib* zur Verfügung. Mit Hilfe der darin zusammengefaßten Funktionen kann die Applikation während der Laufzeit Verbindung mit Xartus aufnehmen und an die eingegebenen Daten gelangen. Außerdem können verschiedene graphische Aktionen, z.B. die Animation von Icons oder das Markieren von Objekten innerhalb der Benutzungsoberfläche, ausgelöst werden. Für die Verwendung von Xartus im Zusammenhang mit einem Modellcompiler wie Xcalibur werden die Funktionen der *artlib* natürlich sowohl in den Quellcode des Compilers als auch in den Quellcode der durch den Compiler generierten Simulationsprozesse eingebunden. Xartus ist also in der Lage, mit mehreren Applikationen gleichzeitig zu kommunizieren und von diesen Befehle zu empfangen. Der Compiler, sozusagen die Hauptapplikation, wird dabei von Xartus zu Beginn der Laufzeit gestartet. Die übrigen Applikationen (hier die Simulationsprozesse) hingegen, werden von Xartus aufgrund eines Befehls der Hauptapplikation (Compiler) gestartet.

Die entsprechenden Funktionen der *artlib* werden nun erläutert. Dabei wird die Kenntnis der grundlegenden Konzepte von Xartus vorausgesetzt, die in [Sterzenbach95] nachgelesen werden können.

8.1 Festlegen der Hauptapplikation

Als Hauptapplikation wird die Anwendung bezeichnet, die von Xartus zu Beginn der Laufzeit gestartet wird. Diese kann innerhalb der Ressourcen-Datei XRes spezifiziert werden. Die zugehörige Ressource lautet *Xartus*applicationname*. Hinter dem Doppelpunkt muß dann der Name der Applikation mit dem entsprechenden Pfad im Verzeichnisbaum angegeben werden.

Die Hauptapplikation ist innerhalb von Lancelot von besonderer Bedeutung, da es die einzige Applikation ist, die direkt über die Menüs im Menü-Bereich des Root-Formulars angesprochen wird. D.h. nachdem eine Menüauswahl innerhalb dieser Menüs stattgefunden hat, wird ein festgelegtes Signal von Xartus über die Kommunikationsverbindung an die

Hauptapplikation gesendet. Die Hauptapplikation wird also über die Benutzermenüs gesteuert. Im Fall eines Modellcompilers für Simulationsprogramme als Hauptapplikation, wird über die Menüs beispielsweise die Compilierung eines Modells angestoßen.

Die Hauptapplikation arbeitet also im Prinzip nur Befehle ab, die sie von Xartus bekommt. Sie muß daher den Aufbau eines ereignis-gesteuerten Programms haben:

```
WHILE (TRUE) DO {
      Warte, solange Befehls-Warteschlange leer
      Hole Befehl aus Warteschlange
      Führe Befehl aus}
```

Die anderen Applikationen (z.B. gestartete Simulationsprozesse) hingegen schreiten kontinuierlich in ihrem Ablauf fort und kommunizieren ausschließlich aus eigener Initiative mit Xartus, um beispielsweise graphische Aktionen auszulösen.

Wie aus diesem Zusammenhang ersichtlich, werden die Funktionen im Menübereich des Root-Formulars von Xartus durch die jeweilige Hauptapplikation bestimmt. Zur Definition der entsprechenden Menüstrukturen mit den zugehörigen Signalen dient der sog. Menü-Editor **im**, der im Verzeichnis ***IConfig*** der Vollversion von Xartus aufgerufen werden kann. Innerhalb dieses Editors kann eine beliebige tiefe Menü-Hierarchie mit Unterpunkten und Untermenüs aufgebaut werden. Bei der Definition von Unterpunkten werden dann auch die entsprechenden Signale festgelegt, die im Falle der Auswahl des Punktes an die Hauptapplikation gesendet werden. Eine genauere Beschreibung des Vorgehens erfolgt am Ende dieses Kapitels.

8.2 Aufbau einer Kommunikationsverbindung

Beim Start einer Applikation übergibt Xartus dem Prozeß einige Kommandozeilenargumente, die für den Aufbau der Kommunikationsverbindung zu Xartus benötigt werden. Das Argument `argv[1]` bezeichnet dabei den Port des Rechners, an dem Xartus seine Dienste für den neu erzeugten Prozeß anbietet, `argv[2]` enthält die Prozeß-ID, die dem neuen Prozeß von Xartus zugeteilt wurde und durch die er von anderen Prozessen referenziert werden kann. Das Argument `argv[3]` bezeichnet den aktuellen Wert des Zeitstempels von Xartus zum Zeitpunkt des Aufrufs. Die Funktion

```
Service* Xs_ConnectServer(char *args[], char *hostname);
```

dient zum Aufbau einer Kommunikationsverbindung zu Xartus. Für den Parameter `args` wird einfach die Variable `argv` der main-Funktion (C-Programm) übergeben, die die oben aufgeführten Kommandozeilenargumente referenziert. Die Funktion baut dann mit Hilfe der Argumente eine Verbindung zu Xartus auf. `hostname` bezeichnet den Rechnernamen des Hosts, auf dem Xartus abläuft. Als Rückgabewert wird eine Referenz auf eine `service`-Struktur zurückgegeben. Innerhalb dieser Struktur ist der Deskriptor, die Prozeß-ID und der Status der Verbindung abgelegt. Fast alle übrigen Funktionen der *artlib* benötigen bei ihrem Aufruf eine Referenz auf eine gültige Service-Struktur, um ihre Aufgaben erledigen zu können. Der Applikations-Programmierer sollte von den in der Struktur enthaltenen Werten allerdings keinen Gebrauch machen. Bei der Funktion

```
Service* Xs_ConnectTimeServer(char *args[], char *hostname, int tid);
```

handelt es sich um eine Erweiterung der Funktion `Xs_ConnectServer`. Zusätzlich zu den bereits beschriebenen Argumenten erhält die Funktion einen Wert für den Parameter `tid`. Die Funktion vergleicht den aktuellen Zeitstempel von Xartus, der über `args` an `Xs_ConnectTimeServer` übergeben wird, mit dem Wert von `tid`, der ebenfalls ein Zeitstempel von Xartus sein muß. Dieser Zeitstempel kann beispielsweise bei der Quellcode-Erzeugung des aufrufenden Prozesses durch einen Modellcompiler als Konstante definiert worden sein. Die Anwendung dieser Funktion ist z.B. sinnvoll, wenn verhindert werden soll, daß Simulationsprozesse gestartet werden, die nicht mehr ihrer Spezifikation innerhalb der Benutzungsoberfläche entsprechen. Der Zeitstempel wird von Xartus bei jeder Änderung an den Modelldaten verändert.

Falls die Werte der Zeitstempel identisch sind, wird die Kommunikationsverbindung ordnungsgemäß aufgebaut. Sonst wird die Verbindung nach ihrem Aufbau sofort wieder geschlossen und der Abbau der Verbindung in der zurückgegebenen Service-Struktur vermerkt. Weitere Aufrufe von Funktionen der *artlib* mit dieser Service-Struktur werden anschließend ignoriert.

Wie bereits oben erwähnt wurde, muß die Hauptapplikation einen ereignis-gesteuerten Aufbau besitzen. D.h. sie wartet solange in ihrem Ablauf, bis sie einen Befehl von Xartus über die Kommunikationsverbindung bekommt, führt diesen dann aus und begibt sich anschließend wieder in den Wartezustand. Die Funktion

```
char* Xs_WaitForSignal(Service *serv, int MaxSeconds);
```

versetzt den aufrufenden Prozeß in einen passiven Wartezustand, bis an der durch `serv` bezeichneten Kommunikationsverbindung ein Kommunikationswunsch vom Server-Prozeß (Xartus) anliegt. Anschließend wird das ankommende Datenpaket empfangen und als

Zeichenkette zurückgegeben. `MaxSeconds` gibt die maximale Anzahl von Sekunden an, die auf eine eintreffende Kommunikationsanforderung gewartet werden soll. Der Wert 0 bezeichnet dabei eine unbegrenzte Zeitspanne.

8.3 Starten weiterer Applikationen

Da Xartus unabhängig von einer speziellen Anwendung entwickelt wurde, erschien es wenig sinnvoll, die Anzahl der zu erzeugenden Applikationsprozesse durch Xartus festzulegen. Aus diesem Grund erlaubt Xartus der Hauptapplikation und allen übrigen Applikationen, weitere Prozesse durch Xartus zu starten. Dazu dient die Funktion

```
int Xs_RunProcess(Service *service, char *processname);
```

Diese Funktion veranlaßt Xartus zum Start des Applikationsprozesses, der über `processname` angegeben wird. Nachdem der Prozeß erfolgreich erzeugt wurde, gibt die Funktion die von Xartus vergebene Prozeß-ID des neuen Prozesses zurück. Anhand dieser ID kann der neue Prozeß innerhalb von Xartus durch den `Xs_RunProcess`-aufrufenden Prozeß identifiziert werden. Dieser Prozeß kann Xartus zu einem späteren Zeitpunkt über die Prozeß-ID dazu veranlassen, die Kommunikationsverbindung zu dem neuen Prozeß zu schließen.

Als weitere Rückgabewerte liefert die Funktion 0, falls der neue Prozeß nicht erzeugt werden konnte (weil z.B. der durch `processname` bezeichnete Prozeß nicht gefunden wurde) und den Wert -1, falls beim Start des Prozesses ein möglicher Zeitstempelvergleich (siehe oben) fehlschlug.

Innerhalb von Lancelot wird die Funktion durch den Modellcompiler Xcalibur (Hauptapplikation) zum Start der Simulationsprozesse verwendet.

8.4 Beenden einer Kommunikationsverbindung

Jeder Applikationsprozeß kann die Kommunikationsverbindung zu Xartus mit der Funktion

```
int Xs_CloseConnection(Service *service);
```

beenden. `service` referenziert dabei wieder eine gültige Service-Struktur. Bei einem Fehler wird 0 zurückgegeben, sonst 1.

Wie bereits oben beschrieben, kann eine Applikation Xartus veranlassen weitere Prozesse zu starten. Über die dabei zurückgegebene Prozeß-ID kann der aufrufende Prozeß Xartus ebenso dazu veranlassen, den gestarteten Prozeß mit der entsprechenden ID wieder zu terminieren oder lediglich die Kommunikationsverbindung zu diesem Prozeß zu schließen. Letzteres wird über die Funktion

```
int Xs_CutOnlyConnection(Service *service, int apid);
```

erreicht. Die Funktion veranlaßt Xartus zum Abkoppeln des Applikationsprozesses mit der Prozeß-ID `apid`. Als Rückgabewert liefert die Funktion 1, falls Xartus die Prozeß-ID akzeptiert, 0 sonst. Dabei ist zu beachten, daß der Rückgabewert 1 nicht bedeutet, daß der betreffende Prozeß bereits tatsächlich abgekoppelt wurde. Dies kann Xartus erst während der nächsten Kommunikation mit dem entsprechenden Prozeß veranlassen. Eine solche Kommunikation wird typischerweise von einem Applikationsprozeß durch dem Aufruf von Funktionen der *artlib* initiiert. Werden ab einem bestimmten Zeitpunkt während eines Prozeßablaufs keine Funktionen der *artlib* mehr abgearbeitet und erfolgt der Aufruf von `Xs_CutOnlyConnection` erst nach diesem Zeitpunkt, so existiert für Xartus keine Möglichkeit, diese Anforderung an den Applikationsprozeß zu übermitteln. Um die Funktion dennoch für das abkoppeln eines später gestarteten Applikationsprozesses zu verwenden, muß die Funktion `Xs_ResetCutFlag` (s. unten) verwendet werden.

Nach dem Aufruf von `Xs_CutOnlyConnection` setzt der abgekoppelte Prozeß seinen Ablauf fort. Der Abbau der Kommunikationsverbindung zu Xartus wird innerhalb der Service-Struktur vermerkt, so daß weitere Aufrufe von Funktionen der *artlib* sofort zurückkehren. Die Funktion wird bei Lancelot dazu eingesetzt, den Ablauf eines Simulationsprozesses, nach der Überprüfung durch den Benutzer, zu beschleunigen. Im Gegensatz dazu bewirkt die Funktion

```
int Xs_CutConnection(Service *service, int apid);
```

die Terminierung des abgekoppelten Prozesses. D.h. der entsprechende Prozeß wird abgebrochen.

```
int Xs_ResetCutFlag(Service *service)
```

Wie bereits oben beschrieben wurde, kann es sein, daß ein Applikationsprozeß nicht abgekoppelt oder terminert werden kann, weil er keine Kommunikation mehr mit Xartus durchführt. Bevor sich ein erneutes `Xs_CutOnlyConnection` oder `Xs_CutConnection` auf einen später neu gestarteten Prozeß beziehen kann, muß der Cut-Mechanismus von Xartus durch den Aufruf von `Xs_ResetCutFlag` zurückgesetzt werden. Bei einer gültigen `service`-Struktur gibt die Funktion den Rückgabewert 1 zurück, sonst 0.

8.5 Funktionen für den Erhalt der Modelldaten

Für die Applikation sind die von Xartus gespeicherten Modelldaten von großer Bedeutung. Diese Modelldaten können mit Hilfe der Funktionen der *artlib* abgefragt werden. Für die Datenübertragung zwischen der Applikation und Xartus werden die Daten in Zeichenketten abgelegt und anschließend übertragen. Für die Ablage der Daten wird von Xartus ein eigenes Format definiert. Für das Verständnis dieses Formats sind Kenntnisse über die interne Objektverwaltung von Xartus erforderlich. Entsprechende Informationen können in [Sterzenbach95] nachgelesen werden.

Die einzelnen Daten werden innerhalb einer Zeichenkette mit Hilfe von festgelegten Begrenzungszeichen voneinander getrennt. Diese Trennzeichen werden in der Datei *artus.h* (im Verzeichnis *Lancelot/Xartus*) festgelegt und können im Bedarfsfall verändert werden. Dabei muß selbstverständlich darauf geachtet werden, keine Zeichen zu verwenden, die über die Tastatur eingegeben werden können. Im folgenden werden wir das Startsymbol mit '<' und das Endsymbol mit '>' bezeichnen, wobei diese Zeichen als tatsächliche Begrenzer natürlich ungeeignet sind. Die tatsächlichen von Xartus verwendeten Begrenzer können vom Programmierer über die Konstanten 'SB' und 'SE' abgefragt werden, die verfügbar sind, sobald die Datei ???Artlib++.h von einem Programm includiert wurde.

```
char* Xs_GetModelData(Service *service);
```

Diese Funktion liefert als Rückgabewert die im Root-Objekt von Xartus gespeicherten Modelldaten als Zeichenkette zurück. Im Fall eines Fehlers liefert die Funktion den Wert NULL. Der Benutzer dieser Funktion muß selbst für die Freigabe des für den Rückgabestring reservierten Speicherplatzes sorgen, nachdem er nicht mehr benötigt wird.

Im folgenden wird nun das Format des Rückgabestrings beschrieben. Für das Verständnis ist wichtig, daß die einzelnen Objekte innerhalb von Xartus durch sog. Path-ID’s bezeichnet werden. Alle Referenzen, die in der Objekthierarchie zwischen den Objekten bestehen, werden in dem Rückgabestring über die Path-ID’s realisiert. Das Format wird mit den folgenden Produktionsregeln beschrieben. Nichtterminalsymbole auf der rechten Seite werden dabei zur besseren Unterscheidung durch // begrenzt:

ROOT ::= <<’Name des Modells’><’Path-ID’><’Wert des Zeitstempels’> /JOBICON_L/ /JOB_L/
<’Main_Code’> /B_LIST/ /CPX_LIST/ /IN_LIST/ /EX_LIST/>

JOBICON_L ::= <{/JOBICON/}>

JOB_L ::= <{/JOB/}>

JOBICON ::= <<’Name des Icons’><’Path-ID’>>

JOB ::= <<’Name des Typobjekts’><’Path-ID’><’Code’>>

B_LIST ::= <{/B_OBJECT/}>

B_OBJECT ::= <<’Name des Basisobjekts’><’Path-ID’> /PORTS/ <’Anzahl der Childs’><’Code’>>

PORTS ::= <{<<’Name des Ports’><’Path-ID des entsprechenden Verbindungsobjekts’>>}>

CPX_LIST ::= <{/COMPLEX/}>

COMPLEX ::= <<’Name des komplexen Objekts’><’Path-ID’><’Path-ID des Vaters im Baum'><‘Code des komplexen Objekts‘>[1]/B_LIST/ /CPX_LIST/ /IN_LIST/ /EX_LIST/>

IN_LIST ::= <{/CONNECTION/}>

EX_LIST ::= <{/CONNECTION/}>

CONNECTION ::= <<'Name des Connection-Objects'><'Path-ID'><'Code'><'Path-ID des ersten Basisobjekts'><'Path-ID des zweiten Basisobjekts'><'Path-ID des Vaters im Baum'>>

Bei Lancelot werden innerhalb des C-SIM-Programmcodes Referenzierungen zwischen den Objekten durch Pfadnamen realisiert. Soll beispielsweise ein Job `job` mit Hilfe einer `Send`-Funktion von einem Prozeß PROCESS_A, der sich innerhalb des komplexen Objekts COMPLEX_A befindet, an einen Prozeß PROCESS_B innerhalb des komplexen Objekts COMPLEX_B gesendet werden, so wird die folgende Referenzierung verwendet:

Send(job, COMPLEX_BPROCESS_B);

D.h. die einzelnen Objektnamen innerhalb eines Pfads im Hierarchiebaum von der Wurzel zu dem betreffenden Objekt werden einfach konkateniert. Um nun eine Zuordnung zwischen den im C-SIM-Code verwendeten Pfadnamen und den intern von Xartus benutzten Path-ID’s vornehmen zu können, kann die Funktion

---

[1] Der Code eines komplexen Objektes wird nur dann in die Modelldaten eingefügt, wenn innerhalb der Ressourcen von Xartus die Ressource *Xartus*extended* auf 1 gesetzt ist.

```
char* Xs_GetAllIDs(Service *service);
```

verwendet werden. Als Rückgabewert gibt diese Funktion in Form einer Zeichenkette zu allen im Modell enthaltenen logischen Objekten jeweils ihren Pfadnamen und ihre Path-ID im folgenden Format zurück:

<<<Pfadname><PathID>><<...><...>>...>.

Im Fall eines Fehlers liefert die Funktion den Wert NULL zurück

Um die Applikation bei der Analyse der Modelldaten zu unterstützen, wurde die Funktion

```
char* Xs_GetObjectData(Service *service, char *pid)
```

eingeführt. Die Funktion gibt die in Xartus innerhalb des Root-Objekts gespeicherten Modelldaten eines Objektes als Zeichenkette zurück. Als Argument erwartet die Funktion die Path-ID (`pid`) des gewünschten Objekts. Die Applikation kann anschließend den zurückgelieferten String analysieren, um die benötigten Informationen zu dem Objekt zu erhalten.

Das Format der Zeichenkette ist ähnlich zu dem von `Xs_GetModelData` verwendeten Format und wird nun beschrieben. Entsprechend der unterschiedlichen Daten, unterscheiden sich auch die Rückgabestring der einzelnen Objekte:

Jeder String beginnt mit einen Teilstring der Länge 3, der den Typ des Objekts anzeigt. Er hat das Format '<i>', i bezeichnet dabei den Typ mit

i=1 : Basisobjekt,
i=2 : komplexes Objekt,
i=3 : Verbindungsobjekt,
i=5 : Typobjekt
i=6 : Iconobjekt

Im Anschluß an die Typbezeichnung folgt dann der jeweilige Rückgabestring:

Basisobjekt

B_OBJECT ::= <<'Name des Basisobjekts'><'Path-ID'>/PORTS/<'Anzahl der Childs'><'Code'>>

PORTS ::= <{<<'Name des Ports'><'Path-ID des entsprechenden Verbindungsobjekts'>>}>

Verbindungsobjekt

CONNECTION ::= <<'Name des Connection-Objects'><'Path-ID'><'Code'><'Path-ID des ersten Basisobjekts'><'Path-ID des zweiten Basisobjekts'><'Path-ID des Vaters im Baum'>>

Komplexes Objekt

CPX_LIST ::= <{/COMPLEX/}>

COMPLEX ::= <<'Name des komplexen Objekts'><'Path-ID'><'Path-ID des Vaters im Baum'><'Code des komplexen Objekts'>[2] /B_LIST/ /CPX_LIST/ /IN_LIST//EX_LIST/>

IN_LIST ::= <{/CONNECTION/}>

EX_LIST ::= <{/CONNECTION/}>

Typobjekt

JOB ::= <<'Name des Typobjekts'><'Path-ID'><'Code'>>

Iconobjekt

JOBICON ::= <<'Name des Icons'><'Path-ID'>>

Im Fall eines Fehlers liefert die Funktion den Wert NULL zurück.

8.6 Funktionen zur Graphiksteuerung

Es wurde bereits darauf hingewiesen, daß ein Applikationsprozeß über die Schnittstelle der *artlib* diverse graphische Aktionen innerhalb der graphischen Benutzungsoberfläche auslösen kann. Dazu gehört die Funktion

```
int Xs_MoveJob(Service *service, char *pid1_icon, char *pid2_conn,
                char *pid3_base);
```

[2] Der Code eines komplexen Objektes wird nur dann in die Modelldaten eingefügt, wenn innerhalb der Ressourcen von Xartus die Ressource *Xartus*extended* auf 1 gesetzt ist.

Der Aufruf dieser Funktion bewirkt eine Animation des Iconobjekts mit der Path-ID `pid1_icon` über die Verbindungslinie des Verbindungsobjekts mit der Path-ID `pid2_conn` innerhalb der graphischen Benutzungsoberfläche. Die Animation beginnt dabei bei dem Basisobjekt mit der Path-ID `pid3_base` (Startpunkt).

Während der für die Animation erforderlichen Kommunikation kann Xartus die Applikation anweisen, die Kommunikationsverbindung zu schließen und sich damit von Xartus abzutrennen (siehe `Xs_CutOnlyConnection`). Dieser Vorgang wird durch den Wert 0 als Rückgabewert von `Xs_MoveJob` signalisiert. Der Wert 1 belegt die fehlerfreie Ausführung der Funktion ohne Trennung. Diese Rückgabewerte sind für alle im folgenden beschriebenen Funktionen zur Graphiksteuerung analog.

```
int Xs_MoveJobExt(Service *service, char *pid1, char *pid2, char *pid3);
```

Diese Funktion unterscheidet sich von `Xs_MoveJob` lediglich durch eine unterschiedliche Belegung der Argumente. `pid1` ist dabei die Path-ID des Basisobjekts (Startobjekt), bei dem die Animation beginnt und `pid3` die Path-ID des Basisobjekts (Zielobjekt), bei dem sie endet. `pid2` bezeichnet die Path-ID des Iconobjekts, das animiert wird.

Beim Aufruf dieser Funktion ermittelt Xartus das zugehörige Verbindungsobjekt zwischen den beiden Basisobjekten. Falls ein solches Objekt nicht existiert wird der Aufruf der Funktion einfach ignoriert. Falls zwischen den beiden Basisobjekten mehrere Verbindungen bestehen, wird diejenige Verbindung gewählt, die als erste erzeugt wurde. Die Rückgabewerte sind identisch zu denen von `Xs_MoveJob`.

Außerdem gibt es Funktionen mit denen die symbolische Darstellung der Objekte auf dem Bildschirm markiert werden kann. Eine Funktion dazu ist

```
int Xs_MarkObject(Service *service, char *pid1);
```

Durch den Aufruf dieser Funktion kann ein Basisobjekt oder komplexes Objekt in der Benutzungsoberfläche markiert oder unmarkiert werden. Dabei wird ein unmarkiertes Objekt nach dem Aufruf dieser Funktion mit der ihm zugeordneten Highlightfarbe und ein bereits markiertes Objekt wieder mit der regulären Hintergrundfarbe dargestellt. Die zweifache Ausführung eines identischen Funktionsaufrufs versetzt das angesprochene Objekt also wieder in seinen Ausgangszustand. `pid1` gibt dabei die Path-ID des Objekts an, das markiert werden soll.

```
int Xs_ColorObject(Service *service, char *pid1, int colorno);
```

Diese Funktion besitzt eine ähnliche Funktionalität wie `Xs_MarkObject`. Dabei kann aber nun die Farbe, mit der das durch `pid` bezeichnete Objekt markiert wird, durch die Applikation explizit festgelegt werden. Dies geschieht über das Argument `colorno`. Folgende Belegungen sind möglich:

`colorno` =1 Festgelegte Hintergrundfarbe des Objekts
=2 Festgelegte Highlightfarbe des Objekts
>2 und <14 Farbe, die durch das entprechende Farbfeld in der Auswahlbox unter Options/Colors (Root-Formular) festgelgt ist.

In manchen Situationen erscheint es für die Applikation wünschenswert, Xartus in einen Zustand zu versetzen, in dem keine Eingaben durch den Benutzer möglich sind. Dazu dient die Funktion

```
int Xs_SetUnsensitive(Service *service);
```

Dadurch läßt sich von der Applikation beispielsweise die Veränderung der Modelldaten durch den Benutzer während einer bestimmten Betriebsphase unterbinden.

```
int Xs_SetSensitive(Service *service);
```

Diese Funktion ist das Gegenstück zur Funktion `Xs_SetUnsensitive`.

Das Root-Formular von Xartus besitzt ein Simulations-Display. Es befindet sich rechts neben dem Steuerbereich und unterteilt sich in zwei Felder, in die vom Applikationsprozeß bestimmte Ausgaben geleitet werden können.

Das erste Feld ist als Beschriftung für die im zweiten Feld ausgegebenen Werte gedacht. Es läßt sich mit der Funktion

```
int Xs_SimLabel(Service *service, char* str);
```

beschriften. `str` bezeichnet dabei die darzustellende Zeichenkette.

In dem zweiten Feld können sowohl Zeichenketten als auch Integer- und Real-Zahlen dargestellt werden. Die entsprechenden Funktionen lauten:

```
int Xs_SimField(Service *service, char* str);

int Xs_SimFieldInt(Service *service, int a);

int Xs_SimFieldDouble(Service *service, double a, int point);
```

In einigen denkbaren Fällen möchte die Hauptapplikation vermeiden, daß bestimmte graphische Funktionen, die durch andere Applikationsprozesse initiiert wurden, von Xartus ausgeführt werden. Dies kann mit Hilfe der Funktion

```
int Xs_Control(Service *service, int flag);
```

gesteuert werden. Durch die Belegung von `flag` können dabei verschiedene Funktionen der Oberfläche deaktiviert oder aktiviert werden:

`flag` = 0: Alle nachfolgenden Kommunikationsanforderungen zum Markieren eines Objekts werden ignoriert.
`flag` = 1: Aktivierung der Markierungsfunktion.
`flag` = 2: Alle nachfolgenden Kommunikationsanforderungen zur Animation werden ignoriert.
`flag` = 3: Aktivieren der Animationsfunktion.
`flag` =4: Alle nachfolgenden Kommunikationsanforderungen zur Aktualisierung des Simulations-Displays werden ignoriert.
`flag` =5: Aktivierung der Simulations-Display-Aktualisierung.

Der Rückgabewert ist immer 1.

8.7 Funktionen zur Objektsteuerung

Jedes logische Objekt innerhalb von Xartus besitzt ein Edit-Formular, in dem die interne Struktur des Objekts, z.B. der Code oder die enthaltenen Objektstrukturen, dargestellt wird. Eine Applikation verarbeitet die vom Benutzer innerhalb dieser Formulare abgelegten Daten und stellt dabei mögliche Eingabefehler fest. Um eine bequeme Korrektur dieser Fehler zu ermöglichen, erlaubt Xartus der Applikation, die Edit-Formulare einzelner Objekte zu öffnen, eine Fehlermeldung auszugeben und zu einer bestimmten Position im Code zu springen. Dies geschieht mit Hilfe der Funktion

```
int Xs_EditObject(Service *service, char *pid, char* string, int pos);
```

Die Funktion veranlaßt ein logisches Objekt (Basisobjekt, Verbindungsobjekt, Typ-Objekt oder komplexes Objekt) mit der Path-ID `pid` zum Öffnen des Edit-Formulars auf dem Bildschirm. Innerhalb des Formulars wird der durch `string` referenzierte Text dargestellt und `pos` bezeichnet die Cursorposition, zu der im Code dieses Objekts gesprungen wird. Der Rückgabewert 1 signalisiert, daß das Formular geöffnet wurde. Beim Wert 0 konnte das betreffende Formular nicht geöffnet werden, entweder weil es bereits geöffnet ist oder es anhand der Path-ID nicht innerhalb der Objektstrukturen identifiziert werden konnte. Auch das Formular für den Main-Code des Root-Objekts kann über eine entsprechende Funktion geöffnet werden:

```
int Xs_EditRoot(Service *service, char* string, int pos);
```

Die Parameter haben hierbei die gleiche Bedeutung wie bei `Xs_EditObject`.

8.8 Zusätzliche Eingabeformulare

Xartus erledigt alle Bildschirmausgaben über entsprechende Dialogelemente des X-Window-Systems. Auch für die Applikationsprozesse gibt es die Möglichkeit, einen einfachen Dialog mit dem Benutzer über entsprechende Dialogelemente zu führen. Die Erzeugung und Ansteuerung dieser Elemente wird dabei von Xartus realisiert. Die einzelnen Funktionen kehren erst dann wieder von ihrer Ausführung zurück, wenn der von ihnen initiierte Dialog auf dem Bildschirm beendet wurde. Im folgenden werden nun die einzelnen Funktionen und Dialogelemente vorgestellt.

```
int Xs_PrintObject(Service *service, char* label, int grabbing);
```

Erzeugt eine einfache Textbox auf dem Bildschirm, in der die durch `label` bezeichnete Zeichenkette dargestellt wird. Diese Box kann durch einen zugehörigen OK-Button geschlossen werden. Der Parameter `grabbing` bestimmt, ob das erzeugte Dialogelement den exclusiven Eingabefocus vom Window-Manager bekommt. D.h. ob der Benutzer gezwungen wird, zuerst diesen Dialog zu bearbeiten, bevor er mit anderen Aufgaben fortfahren kann. Es gibt zwei Möglichkeiten

`grabbing`=1: Exclusiver Eingabefocus

`grabbing`=0: Alle anderen Dialogelemente bleiben aktiv.

Der Rückgabewert 1 signalisiert das Schließen des Formulars; der Wert 0 einen aufgetretenen Fehler.

```
char* Xs_QuestionObject(Service *service, char* label,
                        char* yes_button, char* no_button, int grabbing);
```

Mit Hilfe dieser Funktion kann die Applikation eine binäre Anfrage an den Benutzer richten. Mit `label` wird der entsprechende Text der Anfrage bezeichnet. Durch `yes_button` und `no_button` wird die Beschriftung für die Buttons zur Auswahl der jeweiligen Alternative festgelegt. Als Rückgabewert wird die Beschriftung des ausgewählten Buttons zurückgegeben, also entweder der Wert von `yes_button` oder der Wert von `no_button`. Der Parameter `grabbing` bestimmt wieder die Art des Eingabefocus (siehe oben).

```
char* Xs_DialogObject(Service *service, char* label,
                      char* def, int grabbing);
```

Diese Funktion erzeugt eine Dialogbox zur Eingabe einer Zeichenkette auf dem Bildschirm. Mit `label` kann dabei die geforderte Eingabe erläutert werden. `def` legt einen Defaultwert fest, der in der Eingabezeile angezeigt wird. Als Rückgabewert wird nach Betätigung eines zum Formular gehörenden OK-Buttons, die in der Eingabezeile angezeigte Zeichenkette zurückgegeben. Das Auslösen des CANCEL-Buttons bewirkt den Rückgabewert NULL.

```
int Xs_QuestionObjectIndex(Service *service, char* label,
                       char* yes_button, char* no_button, int grabbing);
```

Diese Funktion besitzt die gleiche Funktionalität wie `Xs_QuestionObject` mit dem einzigen Unterschied, daß sie als Rückgabewert den Wert 1 bei der Auswahl des YES-Buttons und den Wert 0 bei Auswahl des NO-Buttons des Dialogobjekts zurückgibt. Im Fehlerfall ist der Rückgabewert -1.

```
char* Xs_TextObject(Service *service, char* label,
                    char* def, int grabbing);
```

Mit dieser Funktion kann ein mehrzeiliges Texteingabe-Formular auf dem Bildschirm erzeugt werden. `label` bezeichnet dabei die Überschrift des Formulars und `def` einen vorgegebenen Defaultwert, der im Textbereich angezeigt wird. Als Rückgabewert wird nach Betätigung des zum Formular gehörenden OK-Buttons, die im Textbereich angezeigte Zeichenkette zurückgegeben. Nach Auslösen des CANCEL-Buttons ist der Rückgabewert NULL.

```
char* Xs_ListObject(Service *service, char* label,
                    char* alternates, int grabbing);
```

Das durch diese Funktion erzeugte Dialogobjekt ermöglicht dem Benutzer die Auswahl eines Item aus einer Liste von Alternativen. Die dabei möglichen Alternativen werden durch `alternates` angegeben. Innerhalb dieser Zeichenkette sind sie durch Kommata getrennt. `label` definiert die Überschrift des Formulars. Durch Betätigung des zum Formular gehörenden OK-Buttons wird eine ausgewählte Alternative als Rückgabewert zurückgegeben. Falls keine Auswahl stattgefunden hat oder der CANCEL-Button gedrückt wird, ist der Rückgabewert NULL.

```
int Xs_ListObjectIndex(Service *service, char* label,
                        char* alternates, int grabbing);
```

Diese Funktion besitzt die gleiche Funktionalität wie `Xs_ListObject` mit dem Unterschied, daß sie die Position der ausgewählten Alternative innerhalb der Liste `alternates` zurückgibt. Falls keine Auswahl stattgefunden hat oder *Cancel* betätigt wurde, wird 0 zurückgegeben.

```
char* Xs_FileObject(Service *service, char* initpath,
                        char* filter, int grabbing);
```

Diese Funktion erzeugt eine Dialogbox zur Auswahl eines Dateinamens im Verzeichnisbaum des jeweiligen Systems. Mit `initpath` wird der Pfad zum Startverzeichnis festgelegt, `filter` erlaubt das Setzen eines Filters für die Dateinamen. Nach Betätigung des OK-Buttons im Formular, wird ein ausgewählter Dateiname zusammen mit seinem vollständigen Pfad zurückgegeben. Falls keine Auswahl stattgefunden hat oder der CANCEL-Button gedrückt wird, ist der Rückgabewert NULL.

Das Erscheinungsbild und die Beschriftungen der hier vorgestellten Dialogelemente können größtenteils über die jeweiligen Ressourcen in der Datei XRes verändert werden (siehe Anhang, Abschnitt „Dialog-Objekte“).

8.9 Der Menü-Editor *im*

Der Menü-Editor *im* gestattet die applikationsspezifische Definition der Menüs im Menübereich des Root-Formulars. Die Signale, die bei Auswahl der Menüpunkte an die Hauptapplikation gesendet werden, können ebenfalls mit dem *im* festgelegt werden. Nach Aufruf von *im* im Verzeichnis *Lancelot/IConfig* erscheint das Startformular des Menü-Editors auf dem Bildschirm.

Abbildung 38: Startformular des *im*

Über *InterfaceManager* gelangt man in das Hauptformular des Menü-Editors, welches im oberen Bereich des Fensters die Bezeichnungen der festgelegten Menüs der obersten Hierarchiestufe zeigt. In der folgenden Abbildung sind noch keine Menüs definiert.

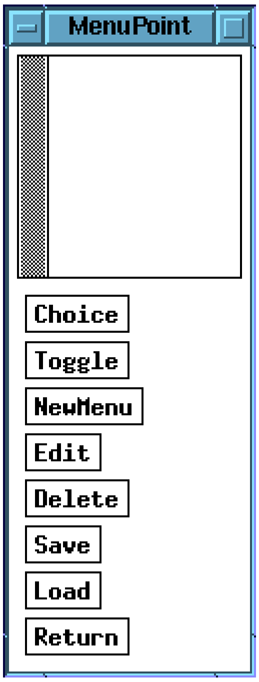


Abbildung 39: Hauptformular des Menü-Editors

Außerdem enthält das Formular verschiedene Buttons zum Definieren der Menühierarchie. Mit *NewMenu* wird ein neues Menü auf der obersten Hierarchiestufe definiert. Dazu wird folgendes Formular geöffnet:

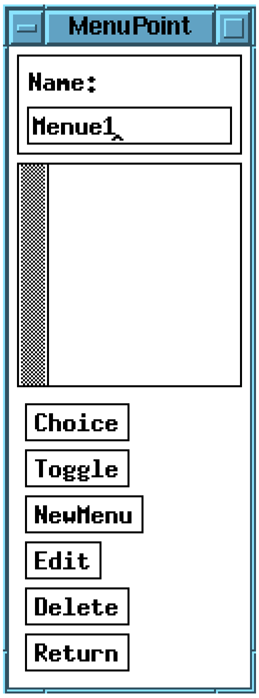


Abbildung 40: Festlegen eines neuen Menüs

Zunächst kann nun ein Name für das Menü festgelegt werden (im Beispiel: Menue1). Anschließend können dann weitere Menüs - nun auf der zweiten Hierarchiestufe- oder Menüpunkte definiert werden. Bei den Menüpunkten werden zwei verschiedene Arten unterschieden. Durch Auswahl von *Choice* gelangt man in das folgende Formular:

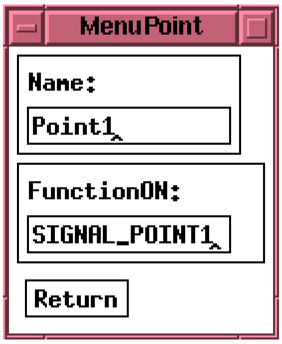


Abbildung 41: Definition eines einfachen Menüpunkts

Hier kann nun zunächst der Name eines einfachen Menüpunkts und dann das zugehörige Signal, das bei Auswahl dieses Punkts an die Applikation gesendet wird, festgelegt werden. Nach Betätigung von *Return* erscheint der festgelegte Name im Formular des entsprechenden Menüs; allerdings erst, nachdem der Mauszeiger in dieses Formular bewegt wurde.

Durch die Auswahl von *Toggle* gelangt man in ein anderes Formular:

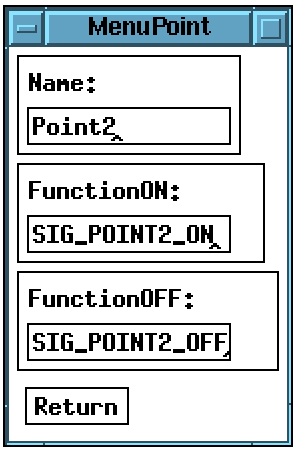


Abbildung 42: Festlegen eines Toggle-Menüpunkts

In diesem Formular kann ein sog. Toggle-Menüpunkt festgelegt werden. Bei mehrfacher Auswahl dieses Menüpunkts werden in alternierender Reihenfolge die festgelegten Signale (im Beispiel: SIG_POINT2_ON und SIG_POINT2_OFF) an die Applikation gesendet. Innerhalb des Menüs wird der Zustand des jeweiligen Toggle-Menüpunkts durch eine entsprechende Markierung angezeigt („X“ für ON bzw. „ “ für OFF). Die Toggle-Menüpunkte eignen sich also insbesondere zur Festlegung von Optionen einer Applikation. Nach Betätigung von *Return* und dem Einfügen eines weiteren Menüs (Menue1.1) ergibt sich folgendes Bild:

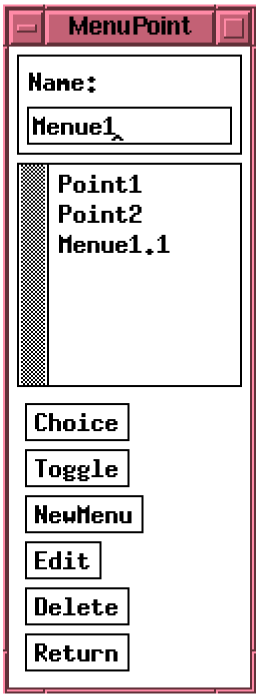


Abbildung 43: Festgelegte Menüpunkte auf der 2.Hierarchiestufe

Durch Auswahl von *Edit* oder *Delete* kann man die Menüeinträge des jeweiligen Menüs verändern bzw. entfernen. Über *Return* gelangt man wieder in das Hauptformular, in dem man entweder weitere Menüs definieren kann oder die festgelegte Menüstruktur über *Save* abspeichern kann. Die Menüdaten werden dabei in der Datei *i_config* im Verzeichnis *IConfig* abgelegt. Ein *Load* lädt die entprechenden Daten dieser Datei. Außerdem bewirkt die Auswahl von *Save* die Erzeugung der Datei *UserMenu.cc* im Verzeichnis *Lancelot/Xartus*, die den Programmcode zur Erzeugung der definierten Menüstrukturen enthält. Nach Beendigung des Menü-Editors über *Return* (Hauptformular) und *Quit* (Startformular), kann die Datei *UserMenu.cc* im Verzeichnis *Xartus* durch Aufruf von *make* neu übersetzt und in den aktuellen Maschinencode eingefügt werden. Nach dem Start von Lancelot enthält der Menübereich des Root-Formulars dann die festgelegte Menüstruktur.

Natürlich muß der Applikationsprogrammierer nach der Definition der Menüstrukturen dafür sorgen, daß die Applikation die festgelegten Signal auch versteht. Damit die gewählten Einstellungen der Toggle-Menüpunkte auch über die jeweilige Lancelot-Sitzung hinaus bestehen bleiben, speichert Xartus diese bei Beendigung des Programms ab. Damit die

Applikation beim Programmstart von Lancelot die aktuellen Einstellungen erfährt, sendet Xartus der Applikation gleich zu Beginn ihrer Laufzeit die aktuell eingestellten Signale der Toggle Menüpunkte.

9 Anhang

9.1 Die Ressourcen von Xartus für Lancelot

Wie bereits erwähnt, wurde Xartus für den Einsatz mit verschiedenen Applikationen entwickelt. Bei Lancelot wird Xartus beispielsweise in Verbindung mit dem Modellcompiler für C-SIM-Code (Xcalibur) eingesetzt. Eine Verwendung für andere Compiler mit ähnlichen Anforderungen oder entsprechende Anwendungen ist ebenfalls möglich.

Xartus stellt eine große Menge sogenannter Ressourcen zur Verfügung, mit denen sich das Aussehen und die Funktionalität der grafischen Benutzungsoberfläche verändern und gestalten läßt. Auf diese Weise kann das Programm an die speziellen Anforderungen einer jeweiligen Einsatzumgebung angepaßt werden.

Für den größten Teil der Ressourcen macht sich Xartus ein Konzept des X-Window-Systems zunutze, das die Festlegung und Veränderung bestimmter Attribute einer X-Anwendung zu jedem beliebigen Zeitpunkt nach der Übersetzung erlaubt. Dabei kann zunächst einmal das Aussehen und Verhalten der einzelnen Dialogobjekte, aus denen eine X-Anwendung (grafische Benutzungsoberfläche) zusammengesetzt ist, durch Belegung der entsprechenden Attribute spezifiziert werden. Außerdem kann eine Anwendung auch eigene Attribute (Ressourcen) erhalten. Dieses Konzept des X-Window-Systems ermöglicht eine sehr flexible Gestaltung und Anpassung von grafischen Benutzungsoberflächen ohne einen erneuten Übersetzungsvorgang.

.Für die Gestaltung grafischer Benutzungsoberflächen unter dem X-Window-System existieren verschiedene Toolkits, die den Programmieraufwand reduzieren sollen. Dazu stellen sie dem Programmierer vorgefertigte Dialogobjekte (z.B. Button, Label, Menüs, usw.) zur Verfügung. Aus der Auswahl der angebotenen Dialogobjekte kann der Programmierer dann eine Benutzungsoberfläche entwickeln. Die einzelnen Dialogobjekte besitzen dabei bereits eine große Anzahl von Ressourcen, die entweder im Programmcode oder in einer Ressourcen-Datei mit Werten belegt werden können.

Xartus wurde mit dem Athena-Widget-Set entwickelt und besitzt dementsprechend auch eine große Anzahl an X-Ressourcen, über die sich das Aussehen der Benutzungsoberfläche gestalten läßt. Dazu gehören selbstverständlich auch die Ressourcen der verwendeten Dialogobjekte. Die X-Ressourcen von Xartus werden in der Datei **XRes** im Verzeichnis

Lancelot/ festgelegt. Diese Datei gliedert sich in 17 Abschnitte, in denen jeweils Attribute für das Aussehen und die Funktionalität charakteristischer Organisationseinheiten des Programms zusammengefaßt sind. Darüber hinaus können nachträglich auch andere Ressourcen der verwendeten Dialogobjekte mit Werten belegt werden. In der folgenden Tabelle werden die wesentlichen Ressourcen und mögliche Belegungen vorgestellt und erläutert. Die Datei XRes enthält darüber hinaus viele weitere Ressourcen, die für die direkte Funktionalität von Lancelot allerdings von geringerer Bedeutung sind und deshalb nicht weiter erläutert werden. Im Bedarfsfall sei hier auf die Kommentierung der XRes-Datei verwiesen.

Für die Verwendung der Statistikkomponente XStat muß zusätzlich die Ressourcen-Datei **XPro30** im Verzeichnis **Lancelot/** in den X-Server geladen werden.

Der Vollständigkeit halber sei erwähnt, daß eine Ressourcen-Datei normalerweise während einer X-Window-Sitzung mit dem Kommando

xrdb -merge <Ressourcen-Datei>

zu den Ressourcen eines X-Servers geladen wird. Bei Lancelot wird diese Aufgabe bis auf weiteres durch die Ausführung des Shell-Skriptes **lancelot** zum Programmstart erledigt (**XRes** und **XPro30**).

9.1.1 Abschnitt 1: Allgemeines

Allgemeine Festlegungen, die sich auf alle erzeugten Dialogobjekte beziehen, insofern keine speziellen Belegungen angegeben werden.

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus.title	Text	Type-Manager	Titelbeschriftung des Typ-Managers
Xartus*font	Font	-adobe-helvetica-medium-r-normal--14-140-75-75-p-77-iso8859-1	Sofern kein spezieller Font gewählt wurde, wird für alle Textausgaben und Beschriftungen der hier spezifizierte Font gewählt

Xartus*shadowWidth	Cardinal	2	Breite der Schattierung bei den 3-D-Athena-Widgets, falls kein spezieller Wert festgelegt wird.
Xartus*workpath	Verzeichnis		Die Ressource `Xartus*workpath` muß mit dem vollständigen Verzeichnispfad zum Zielverzeichnis (im Benutzer-Verzeichnisbaum) belegt werden, der um das Verzeichnis *Lancelot* erweitert wird. Wurde Lancelot also z.B. im Home-Verzeichnis des Benutzers *artus* installiert, und lautet der Pfad zu diesem Benutzer-Verzeichnis */home/student/artus*, so lautet die korrekte Belegung */home/student/artus/Lancelot*.
Xartus*programpath	Verzeichnis		Der Pfad `Xartus*programpath` lokalisert die benötigten Programmdateien von Lancelot in den Systemverzeichnissen. Er lautet in der Umgebung des ORLAB's auf allen Maschinen */usr/orlab/lancelot/Lancelot*.
Xartus*modelpath	Verzeichnis		Für `Xartus*modelpath` muß der Pfad zu den Benutzermodellen eingesetzt werden. Er lautet *Models*.
Xartus*fileServer*initialDirectory	Verzeichnis		Für `Xartus*modelpath` muß der Pfad zu den Benutzermodellen eingesetzt werden. Er lautet *Models*.
Xartus*applicationpath1	Verzeichnis		`Xartus*applicationpath1` wird schließlich mit dem gleichen Pfad wie `Xartus*programpath` initialisiert.

9.1.2 Abschnitt 2: Applikation

In diesem Abschnitt werden die Ressourcen festgelegt, die speziell für Xartus definiert wurden, also nicht zu verwendeten Dialogobjekten gehören. Dadurch kann beispielsweise eine bestimmte Farbwahl oder der Text von Fehlermeldungen festgelegt werden.

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
*Xartus*convert: 0	0 oder 1	0	Ab Mai 1997 wurden die Modelldaten um die lokalen Code-Bereiche der komplexen Objekte erweitert. Diese Daten müssen selbstverständlich auch mit abgespeichert werden, so daß eine Änderung des Dateiformats zum Speichern notwendig war. Wenn diese Ressource auf 1 gesetzt ist, können alte Modelle geladen und im neuen Format gespeichert werden. Nachdem alle Modelle konvertiert wurden, muß daß Flag unbedingt wieder auf 0 gesetzt werden! Durch eine zusätzliche Versionsnummer im Dateiformat sind solche Konvertierungen in Zukunft überflüssig.
*Xartus*cut	Cardinal (ASCII)	46	Tennzeichen für die Pfadangabe in der Infozeile des Root-Formulars.
*Xartus*c_name	Text	„Connection-Object“	Infotext der Verbindungsobjekte in der Infozeile.(1)
*Xartus*c_info1	Text	„ connected with“	Infotext der Verbindungsobjekte in der Infozeile.(1)
*Xartus*cp_name	Text	„Complex-Object“	Infotext der komplexen Objekte in der Infozeile.(2)

*Xartus*b_name	Text	„Base-Object“	Infotext der Basisobjekte in der Infozeile.(3)
*Xartus*p_name	Text	„Port“	Infotext der Basisobjekte in der Infozeile.(3)
*Xartus*p_info1	Text	„connected with“	Infotext der Basisobjekte in der Infozeile.(3)
*Xartus*p_info2	Text	„is free“	Infotext der Basisobjekte in der Infozeile.(3)
*Xartus*applicationname	Text	../Xcalibur/c1	Name und Pfad der Applikation mit der Xartus zusammenarbeiten soll.
*Xartus*animationspeed	Cardinal	10000	Durch diesen Faktor läßt sich die Geschwindigkeit der Animation zusätzlich beeinflussen. (größerer Wert -> langsamere Bewegung)
*Xartus*backpixmap	Pixmap	keine	Hintergrundbild für das Root-Formular. Als Wert muß der vollständige Pfad eines gewählten Pixmaps angegeben werden.
*Xartus*gitter	Cardinal	16	Abstand der Linien des Gitternetzes im Formular der komplexen Objekte und des Root-Objekts.
*Xartus*extended: 0	0 oder 1	1	Ab Mai 1997 wurden die komplexen Objekte um einen lokalen Code-Bereich erweitert. Diese zusätzlichen Daten werden bei Belegung der Ressource mit 1 zusammen mit den übrigen Modelldaten zur Applikation übertragen. Falls die Ressource auf 0 gesetzt ist werden die Daten nicht übermittelt. Dann ist auch keine Eingabe von Daten in den lokalen

			Code-Bereich der Formulare der komplexen Objekte möglich.
*Xartus*message1	Text	\ W A R N U N G!\n\ Waehrend der Simulation duerfen\n\ keine Objekte entfernt werden!	Meldung bei dem Versuch, ein Objekt während einer Animation zu entfernen.
*Xartus*message2	Text	\ W A R N U N G!\n\ Waehrend der Simulation darf\n\ kein neuen Modell geladen werden!	Meldung bei dem Versuch, ein neues Modell während einer Animation zu laden.
*Xartus*message3	Text	\ A C H T U N G !\n\ Bei diesem Vorgang werden\n\ alle Modelldaten geloescht!	Meldung beim Aufruf von „Clear“ im Root-Formular.
*Xartus*message4:	Text	\ W A R N U N G!\n\ Es gibt ungespeicherte	Meldung beim Versuch, Xartus mit ungespeicherten Änderungen zu beenden.

		Aenderungen!\n\ Lancelot wirklich beenden?	
*Xartus*message5	Text	\ W A R N U N G!\n\ Es gibt ungespeicherte \n\ Aenderungen am aktuellen Modell!	Meldung beim Versuch, ein neues Modell zu laden ohne die Änderungen an dem aktuellen Modell zu speichern.
*Xartus*message6	Text	\Der Simulationsprozess existiert nicht!\n\ Möglicherweise wurde er noch nicht\n\ compilert oder der zugehörige\n\ Simulationscode geloescht!	Meldung beim Versuch, einen nicht vorhandenen Prozeß durch den Aufruf der entsprechenden Funktion der artlib-Schnittstelle zu starten.
*Xartus*message7	Text	\ A C H T U N G !\n\ Der Simulationsprozess\n\	Meldung nach dem Absturz eines Prozesses, der durch die entsprechende Funktion der artlib gestartet wurde.

		reagiert nicht mehr!	
*Xartus*message8	Text	\ A C H T U N G !\n\ Die Applikation reagiert nicht mehr!\n\ Neu Starten?	Meldung nach dem Absturz der Applikation (mit Option des Neustarts).
*Xartus*logo	Bitmap	lance	Im Root-Formular dargestelltes Logo. Als Belegung wird der Pfad einer entsprechenden Bitmap-Datei erwartet.
*Xartus*root_min_width	Cardinal	480	Minimale Breite des Root-Formulars.
*Xartus*root_max_width	Cardinal	980	Maximale Breite des Root-Formulars.
*Xartus*root_min_height	Cardinal	380	Minimale Höhe des Root-Formulars.
*Xartus*root_max_height	Cardinal	980	Maximale Höhe des Root-Formulars.
*Xartus*root_width	Cardinal	600	Standardbreite des Root-Formlars.
*Xartus*root_height	Cardinal	420	Standardhöhe des Root-Formlars.

*Xartus*complex_min_width	Cardinal	300	Minimale Breite des Eingabeformulars der komplexen Objekte.
*Xartus*complex_max_width	Cardinal	800	Maximale Breite des Eingabeformulars der komplexen Objekte.
*Xartus*complex_min_height	Cardinal	340	Minimale Höhe des Eingabeformulars der komplexen Objekte.
*Xartus*complex_max_height	Cardinal	800	Maximale Höhe des Eingabeformulars der komplexen Objekte.
*Xartus*complex_width	Cardinal	300	Standardbreite des Eingabeformulars der komplexen Objekte.
*Xartus*complex_height	Cardinal	200	Standardhöhe des Eingabeformulars der komplexen Objekte.
*Xartus*appcolor2	Text	UNDEFINED	Beschriftung des 2. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor3	Text	UNDEFINED	Beschriftung des 3. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor4	Text	UNDEFINED	Beschriftung des 4. Farbfeldes der Auswahlbox unter *Options/Colors*

*Xartus*appcolor5	Text	UNDEFINED	Beschriftung des 5. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor6	Text	UNDEFINED	Beschriftung des 6. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor7	Text	UNDEFINED	Beschriftung des 7. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor8	Text	UNDEFINED	Beschriftung des 8. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor9	Text	UNDEFINED	Beschriftung des 9. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor10	Text	UNDEFINED	Beschriftung des 10. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor11	Text	UNDEFINED	Beschriftung des 11. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor12	Text	UNDEFINED	Beschriftung des 12. Farbfeldes der Auswahlbox unter *Options/Colors*
*Xartus*appcolor13	Text	UNDEFINED	Beschriftung des 13. Farbfeldes der Auswahlbox unter *Options/Colors*

Format des Infostrings der Verbindungsobjekte: <C_INFOSTRING> ::=
<C_INFOSTRING1> <*Xartus*c_info1> <B_INFOSTRING1><B_INFOSTRING1>

<C_INFOSTRING1> ::= <*Xartus*c_name> ObjectID

Format des Infostrings der komplexen Objekte: <*Xartus*cp_name> ObjectID

Format des Infostrings der Basisobjekte: <B_INFOSTRING> ::= <B_INFOSTRING1>, {<p_name> <p_info1> <C_INFOSTRING1>|<p_info2>}

<B_INFOSTRING1> ::= <*Xartus*b_name> ObjectID

9.1.3 Abschnitt 3: Typ-Manager

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*NewTypes.width	Cardinal	60	Breite des Button zur Erzeugung eines neuen Typs.
Xartus*NewTypes.label	Text	New	Beschriftung des Buttons s.o.
Xartus*NewTypes.background	Farbname aus rgb.txt	grey80	Farbe des Buttons s.o.
Xartus*LoadTypes			Ressourcen für den Button zum Laden von Typen. Gleiche Ressourcen wie `Xartus*NewTypes`.
Xartus*SaveTypes			Ressourcen für den Button zum Speichern von Typen. Gleiche Ressourcen wie `Xartus*NewTypes`.
Xartus*TBase*label	Text	Base	Name des Menüeintrags zum Erzeugen, Laden und Speichern eines Basisobjekt-Typs.
Xartus*TBase*width	Cardinal	80	Breite des Menüeintrags s.o.
Xartus*TBase*shado	Cardinal	0	Schattierung des Menüeintrags s.o. bei

wWidth			Verwendung der 3D-Athena-Widgets.
Xartus*TBase*justify	Orientierung: (Left|Right|Middle).	Left	Orientierung des Menüeintrags s.o.
Xartus*TConnection			Ressourcen für den Menüeintrag zum Erzeugen, Laden und Speichern eines Verbindungsobjekt-Typs. Gleiche Ressourcen wie Xartus*Tbase.
Xartus*TComplex			Ressourcen für den Menüeintrag zum Erzeugen, Laden und Speichern eines komplexen Objekttyps. Gleiche Ressourcen wie Xartus*Tbase.
Xartus*TJob			Ressourcen für den Menüeintrag zum Erzeugen, Laden und Speichern eines Typobjekt-Typs. Gleiche Ressourcen wie Xartus*Tbase.
Xartus*TIcon			Ressourcen für den Menüeintrag zum Erzeugen, Laden und Speichern eines Iconobjekt-Typs. Gleiche Ressourcen wie Xartus*Tbase.
Xartus*B_typ_label*label	Text	BaseTypes	Beschriftung des Typbereichs der Basisobjekte.
Xartus*B_typ_label.width:	Cardinal	180	Breite der Beschriftung des Typbereichs s.o.
Xartus*B_typ_label.font	Font	-*-helvetica-medium-o-	Schriftart der Beschriftung des Typbereichs s.o.

		normal-*-12-*-*-*-*-*-*-*	
Xartus*Cp_typ_label			Ressourcen für die Beschriftung des Typbereichs der komplexen Objekte. Gleiche Ressourcen wie Xartus*B_typ_label.
Xartus*C_typ_label			Ressourcen für die Beschriftung des Typbereichs der Verbindungsobjekte. Gleiche Ressourcen wie Xartus*B_typ_label.
Xartus*Job_typ_label			Ressourcen für die Beschriftung des Typbereichs der Typobjekte. Gleiche Ressourcen wie Xartus*B_typ_label.
Xartus*JobIcon_typ_label			Ressourcen für die Beschriftung des Typbereichs der Iconobjekte. Gleiche Ressourcen wie Xartus*B_typ_label.

9.1.4 Abschnitt 4: Root-Formular

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*Root_logo.shadowWidth	Cardinal	0	Breite der Schattierung des Root-Logos durch die 3D-Athena-Widgets.
Xartus*Root_logo.borderWidth	Cardinal	0	Breite des Randes des Root-Logos.
Xartus*Root_logo.background	Farbname aus rgb.txt	grey90	Hintergundfarbe des Root-Logos.

Xartus*sbar.shadowWidth	Cardinal	2	Breite der Schattierung des Speedbars durch die 3D-Athena-Widgets.
Xartus*sbar.width	Cardinal	120	Breite des Speedbars.
Xartus*sbar.height	Cardinal	16	Höhe des Speedbars.
Xartus*sbar.minimumThumb	Cardinal	6	
Xartus*sbar.topOfThumb	Float (0.0-1.0)	0.0	
Xartus*sbar.shown	Float (0.0-1.0)	0.1	
Xartus*simlabel.width	Cardinal	140	Breite der Beschriftung des Simulations-Display.
Xartus*simtime.width	Cardinal	140	Breite des Simulations-Display.
Xartus*entry_label.borderWidth	Cardinal	0	Randbreite der Beschriftung der Applikationsmenüs.
Xartus*entry_label*shadowWidth	Cardinal	0	Breite der Schattierung der Beschriftung s.o.
Xartus*menu_entry*shadowWidth	Cardinal	0	Breite der Schattierung der Menüeinträge in den Applikationsmenüs.
Xartus*menu_entry.width	Cardinal	140	Breite der Menüeinträge in den Applikationsmenüs.
Xartus*menu_entry.borderWidth	Cardinal	0	Randbreite der Menüeinträge in den Applikationsmenüs.
Xartus*menu_entry.justify	Orientierung (Left|Right|Middle).	left	Orientierung der Menüeinträge in den Applikationsmenüs.

Xartus*menuButton.background	Farbname aus rgb.txt	grey80	Hintergrundfarbe der Menübuttons in den Applikationsmenüs.
Xartus*menuButton.width	Cardinal	160	Breite der Menübuttons in den Applikationsmenüs.
Xartus*menuButton.justify	Orientierung (Left|Right|Middle).	left	Orientierung der Menübuttons in den Applikationsmenüs.
Xartus*code_root.width	Cardinal	300	Breite des Code-Formulars.
Xartus*code_root.height	Cardinal	200	Höhe des Code-Formulars.
Xartus*okbutt_code*label	Text	OK	Beschriftung des Buttons zum Schließen des Code-Formulars mit Speicherung.
Xartus*okbutt_code.background	Farbname aus rgb.txt	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*okbutt_code*width	Cardinal	70	Breite des Buttons s.o.
Xartus*cancelbutt_code			Ressourcen für den Button zum Schließen des Code-Formulars ohne Speicherung. Gleiche Ressourcen wie Xartus*okbutt_code.
Xartus*Cp_ed_box_code_O*label	Text	Code	Beschriftung des Buttons zum Öffnen des Code-Formulars.
Xartus*Cp_ed_box_code_O*width	Cardinal	90	Breite des Buttons s.o.
Xartus*Cp_ed_box_code_O.background	Farbname aus rgb.txt	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*Cp_ed_box_j			Ressourcen des Buttons zum Öffnen

ob_O			des Type-Formulars im Root-Formular. Gleiche Ressourcen wie Xartus*Cp_ed_box_code_O.
Xartus*Cp_ed_box_jobicon_O			Ressourcen des Buttons zum Öffnen des Icon-Formulars im Root-Formular. Gleiche Ressourcen wie Xartus*Cp_ed_box_code_O.
Xartus*modellbutt			Ressourcen des Buttons zum Öffnen des File-Menüs im Root-Formular. Gleiche Ressourcen wie Xartus*Cp_ed_box_code_O.
Xartus*Cp_ed_box_iconO			Ressourcen des Buttons zum Aufruf des Icon-Managers im Root-Formular. Gleiche Ressourcen wie Xartus*Cp_ed_box_code_O.
Xartus*Cp_ed_box_quitO			Ressourcen des Buttons zum Beenden des Programms.
Xartus*Cp_ed_box_loadO*label	Cardinal	Load	Beschriftung des Buttons zum Laden eines Modells im File-Menü.
Xartus*Cp_ed_box_loadO*width	Cardinal	90	Breite des Buttons s.o.
Xartus*Cp_ed_box_loadO.borderWidth	Cardinal	0	Randbreite des Buttons s.o.
Xartus*Cp_ed_box_loadO*shadowWidth	Cardinal	0	Breite der Schattierung des Buttons s.o. durch die 3D-Athena-Widgets.
Xartus*Cp_ed_box_saveO			Ressourcen für den Button zum Speichern eines Modells. Gleiche Ressourcen wie Xartus*Cp_ed_box_loadO.
Xartus*Cp_ed_box_loadall			Ressourcen für den Button zum Laden aller Modelldaten mit Typen. Gleiche

			Ressourcen wie Xartus*Cp_ed_box_loadO.
Xartus*Cp_ed_box_svalO			Ressourcen für den Button zum Speichern aller Modelldaten mit Typen. Gleiche Ressourcen wie Xartus*Cp_ed_box_loadO.
Xartus*Cp_ed_box_xfimO			Ressourcen für den Button zum Aufruf des Dateimanagers. Gleiche Ressourcen wie Xartus*Cp_ed_box_loadO.
Xartus*Cp_ed_box_clearO			Ressourcen für den Button zum Löschen aller Modelldaten. Gleiche Ressourcen wie Xartus*Cp_ed_box_loadO.
Xartus*Cp_ed_box_options_colors			Ressourcen für den Button zum Öffnen der Farbauswahl für die Applikation im Options-Menü des Root-Formulars. Gleiche Ressourcen wie Xartus*Cp_ed_box_loadO.
Xartus*Cp_ed_box_options_gitter			Ressourcen für den Button zum Einstellen der Gitternetzdichte im Options-Menü des Root-Formulars. Gleiche Ressourcen wie Xartus*Cp_ed_box_loadO.
Xartus*Cp_ed_box_options_speed			Ressourcen für den Button zum Regulieren der Animationsgeschwindigkeit im Options-Menü des Root-Formulars. Gleiche Ressourcen wie Xartus*Cp_ed_box_loadO.
Xartus*Cp_ed_box_options_pixmap			Ressourcen für den Button zum Festlegen eines Hintergundbildes im Options-Menü des Root-Formulars. Gleiche Ressourcen wie

			Xartus*Cp_ed_box_loadO.

9.1.5 Abschnitt 5: Basisobjekte

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*a_typ_box_text0*font	Font	-misc-fixed-*-*-*-*-14-*-*-*-*-*-*-*	Schriftart für den Texteingabebereich (Name) des Eingabeformulars der Basisobjekte.
Xartus*a_typ_box_text2*font:	Font	-misc-fixed-*-*-*-*-14-*-*-*-*-*-*-*	Schriftart für den Texteingabebereich (Code) des Eingabeformulars der Basisobjekte.
Xartus*b_object_lb0.shadowWidth	Cardinal	2	Breite der Schattierung des Iconbereichs der grafischen Repräsentation der Basisobjekte.
Xartus*b_object_lb1.background	Farbname aus rgb.txt	white	Hintergrundfarbe des Namensbereichs der grafischen Repräsentation der Basisobjekte.
Xartus*b_object_lb1.font	Font	-*-helvetica-medium-r-normal-*-12-*-*-*-*-*-*-*	Schriftart des Namensbereichs der grafischen Repräsentation der Basisobjekte.
Xartus*a_typ_box_label0*label	Text	Name:	Beschriftung des Eingabebereichs für den Namen im Eingabeformular der Basisobjekte.
Xartus*a_typ_box_label0.width	Cardinal	390	Breite der Beschriftung für den Eingabebereich des Namens im Eingabeformular der Basisobjekte.
Xartus*a_typ_box_text0.width	Cardinal	390	Breite des Eingabebereichs für den Namen im Eingabeformular der

			Basisobjekte.
Xartus*a_typ_box_text2.width	Cardinal	400	Breite des Eingabebereichs für den Code im Eingabeformular der Basisobjekte.
Xartus*a_typ_box_text2.height	Cardinal	460	Höhe des Eingabebereichs für den Code im Eingabeformular der Basisobjekte.
Xartus*a_typ_box_com1*label	Text	OK	Beschriftung des Buttons zum Schließen des Eingabeformulars der Basisobjekte mit Speicherung.
Xartus*a_typ_box_com1.background	Farbname aus rgb.txt	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*a_typ_box_com1.width	Cardinal	70	Breite des Buttons s.o.
Xartus*a_typ_box_com2			Ressourcen des Buttons zum Schließen des Eingabeformulars der Basisobjekte ohne Speicherung. Gleiche Ressourcen wie Xartus*a_typ_box_com1.
Xartus*a_typ_box_com3			Ressourcen des Buttons zum Aufruf der Iconauswahl. Gleiche Ressourcen wie Xartus*a_typ_box_com1.
Xartus*B_Menu2Label*label	Text	Functions	Titelbeschriftung des Funktionsmenüs der Basisobjekte.
Xartus*B_Menu2Label*font	Font	-*-helvetica-medium-o-normal-*-10-*-*-*-*-*-*	Font der Titelbeschriftung s.o.
Xartus*B_Menu2Label*width	Cardinal	70	Breite der Titelbeschriftung s.o.
Xartus*B_Menu2Copy*label: Copy			Ressourcen des Menüeintrags zum Kopieren im Funktionsmenü der

			Basisobjekte. Gleiche Ressourcen wie Xartus*B_Menu2Label.
Xartus*B_Menu2Child			Ressourcen des Menüeintrags zum Erzeugen eines Childs im Funktionsmenü der Basisobjekte. GleXartus*conn_typ_box_text0*font: -misc-fixed-*-*-*-*-14-*-*-*-*-*-*-* iche Ressourcen wie Xartus*B_Menu2Label.
Xartus*B_Menu2Edit			Ressourcen des Menüeintrags zum Editieren im Funktionsmenü der Basisobjekte. Gleiche Ressourcen wie Xartus*B_Menu2Label.
Xartus*B_Menu2Delete			Ressourcen des Menüeintrags zum Löschen im Funktionsmenü der Basisobjekte. Gleiche Ressourcen wie Xartus*B_Menu2Label.
Xartus*B_Menu2CopyToType			Ressourcen des Menüeintrags zum Kopieren in die Typauswahl im Funktionsmenü der Basisobjekte. Gleiche Ressourcen wie Xartus*B_Menu2Label.

9.1.6 Abschnitt 6: Verbindungsobjekt

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*conn_typ_box_text0*font:	Font	-misc-fixed-*-*-*-*-14-*-*-*-*-*-*-*	Schriftart für den Texteingabebereich (Name) des Eingabeformulars der Verbindungsobjekte.
Xartus*conn_typ_bo	Font	-misc-fixed-*-*-	Schriftart für den Texteingabebereich

x_text2*font		*-*-14-*-*-*-*-*-*-*	(Code) des Eingabeformulars der Verbindungsobjekte.
Xartus*conn_typ_box_label0.width	Cardinal	250	Breite der Beschriftung des Eingabebereichs (Name) im Eingabeformular der Verbindungsobjekte.
Xartus*conn_typ_box_text0.width	Cardinal	250	Breite des Eingabebereichs (Name) im Eingabeformular der Verbindungsobjekte.
Xartus*conn_typ_box_text2.width	Cardinal	260	Breite des Eingabebereichs (Code) im Eingabeformular der Verbindungsobjekte.
Xartus*conn_typ_box_text2.height	Cardinal	80	Höhe des Eingabebereichs (Code) im Eingabeformular der Verbindungsobjekte.
Xartus*c_typA_box_com1*label	Text	OK	Beschriftung des Buttons zum Schließen des Eingabeformulars der Verbindungsobjekte mit Speicherung.
Xartus*c_typA_box_com1.background	Farbname aus rgb.txt	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*c_typA_box_com1.width	Cardinal	70	Breite des Buttons s.o.
Xartus*c_typA_box_com2*label			Ressourcen des Buttons zum Schließen des Eingabeformulars der Verbindungsobjekte ohne Speicherung. Gleiche Ressourcen wie Xartus*c_typA_box_com1.
Xartus*c_typA_box_com3*label: Icon			Ressourcen des Buttons zum Aufruf des Line-Formulars. Gleiche Ressourcen wie Xartus*c_typA_box_com1.

Xartus*B_Menu4Label*label	Text	Functions	Titelbeschriftung des Funktionsmenüs der Verbindungsobjekte.
Xartus*B_Menu4Label*font	Font	-*-helvetica-medium-o-normal-*-10-*-*-*-*-*-*-*	Font der Titelbeschriftung s.o.
Xartus*B_Menu4Label*width	Cardinal	70	Breite der Titelbeschriftung s.o.
Xartus*B_Menu4Copy			Ressourcen des Menüeintrags zum Kopieren im Funktionsmenü der Verbindungsobjekte. Gleiche Ressourcen wie Xartus*B_Menu4Label.
Xartus*B_Menu4Edit			Ressourcen des Menüeintrags zum Editieren im Funktionsmenü der Verbindungsobjekte. Gleiche Ressourcen wie Xartus*B_Menu4Label.
Xartus*B_Menu4Delete			Ressourcen des Menüeintrags zum Löschen im Funktionsmenü der Verbindungsobjekte. Gleiche Ressourcen wie Xartus*B_Menu4Label.
Xartus*B_Menu4CopyToType			Ressourcen des Menüeintrags zum Kopieren in die Typauswahl im Funktionsmenü der Verbindungsobjekte. Gleiche Ressourcen wie Xartus*B_Menu4Label.
Xartus*line_box_label0.label	Text	LineWidth	Beschriftung des Eingabebereichs für die Linienstärke im Line-Formular.
Xartus*line_box_label2.label	Text	Number of Dashpoints:	Beschriftung des Eingabebereichs für die Dashpoints im Line-Formular.

9.1.7 Abschnitt 7: Komplexe Objekte

Ressourcen-Name	Wertebereich	Default	Erläuterung
Xartus*cp_typ_box_text0*font	Font	-misc-fixed-*-*-*-*-14-*-*-*-*-*-*-*	Schriftart für den Eingabebereich des Namens im Eingabeformular der komplexen Objekte.
Xartus*cp_object_lb1.background	Farbname aus rgb.txt	white	Hintergrundfarbe des Namensbereichs der grafischen Repräsentation der komplexen Objekte.
Xartus*cp_object_lb1.background	Farbname aus rgb.txt	white	Hintergrundfarbe der Drawing-Area im Eingabeformlar der komplexen Objekte und im Root-Formular.
Xartus*B_Menu3Label*label	Text	Functions	Titelbeschriftung des Funktionsmenüs der komplexen Objekte.
Xartus*B_Menu3Label*font	Font	-*-helvetica-medium-o-normal-*-10-*-*-*-*-*-*-*	Font der Titelbeschriftung s.o.
Xartus*B_Menu3Label*width	Cardinal	70	Breite der Titelbeschriftung s.o.
Xartus*B_Menu3Edit			Ressourcen des Menüeintrags zum Editieren im Funktionsmenü der komplexen Objekte. Gleiche Ressourcen wie Xartus*B_Menu3Label.
Xartus*B_Menu3Copy			Ressourcen des Menüeintrags zum Kopieren im Funktionsmenü der komplexen Objekte. Gleiche Ressourcen wie Xartus*B_Menu3Label.
Xartus*B_Menu3Del			Ressourcen des Menüeintrags zum

ete			Löschen im Funktionsmenü der komplexen Objekte. Gleiche Ressourcen wie Xartus*B_Menu3Label.
Xartus*B_Menu3CopyToType			Ressourcen des Menüeintrags zum Kopieren in die Typauswahl im Funktionsmenü der komplexen Objekte. Gleiche Ressourcen wie Xartus*B_Menu3Label.
Xartus*Cp_nlabel*label	Text	New Object	Beschriftung des Funktionsmenüs in der Drawing-Area.
Xartus*Cp_nlabel*width	Cardinal	70	Breite der Beschriftung des Funktionsmenüs in der Drawing-Area.
Xartus*Cp_nlabel*font	Font	-*-helvetica-medium-o-normal-*-10-*-*-*-*-*-*	Schrifttyp der Beschriftung des Funktionsmenüs in der Drawing-Area.
Xartus*Cp_nbase*label	Text	Base	Beschriftung des Menüeintrags im Funktionsmenü s.o.
Xartus*Cp_nbase.background	Farbname aus rgb.txt	grey80	Hintergrundfarbe des Menüeintrags s.o.
Xartus*Cp_nbase*width	Cardinal	70	Breite des Menüeintrags s.o.
Xartus*Cp_nbase*font	Font	-*-helvetica-medium-r-normal-*-10-*-*-*-*-*-*	Schrifttyp des Menüeintrags s.o.
Xartus*Cp_nconnection			Ressourcen für den Menüeintrag zum Erzeugen eines neuen Verbindungsobjekts im Funktionsmenü der Drawing-Area. Gleiche Ressourcen wie Xartus*Cp_nbase

Xartus*Cp_ncomplex			Ressourcen für den Menüeintrag zum Erzeugen eines neuen komplexen Objekts im Funktionsmenü der Drawing-Area. Gleiche Ressourcen wie Xartus*Cp_nbase
Xartus*info_text.*thickness	Cardinal	6	Höhe der Infozeile im Eingabeformular der komplexen Objekte und im Root-Formular.

9.1.8 Abschnitt 8: Iconobjekte

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*job_text0*font:	**Font**	-misc-fixed-*-*-*-*-14-*-*-*-*-*-*-*	Schriftart für den Eingabebereich des Namens im Eingabeformular der Iconobjekte.
Xartus*mover.shadowWidth		0	Breite der Schattierung des Animationsicons durch die 3D-Athena-Widgets.
Xartus*mover.background	Farbname aus rgb.txt	white	Hintergrundfarbe des Animationsicons.
Xartus*job_label2.background	Farbname aus rgb.txt	white	Hintergrundfarbe des Namensbereichs der grafischen Repräsentation der Iconobjekte.
Xartus*job_box_label0.width	Cardinal	370	Breite der Beschriftung des Eingabebereichs für den Namen im Eingabeformular der Iconobjekte.
Xartus*job_text0.width	Cardinal	370	Breite des Eingabebereichs für den Namen im Eingabeformular der Iconobjekte.

Xartus*job_box_text1.width	Cardinal	380	Breite des Eingabebereichs für den Code im Eingabeformular der Iconobjekte.
Xartus*job_box_text1.height	Cardinal	280	Höhe des Eingabebereichs für den Code im Eingabeformular der Iconobjekte.
Xartus*job_com1.label	Text	OK	Beschriftung des Buttons zum Schließen des Eingabeformulars der Iconobjekte mit Speicherung.
Xartus*job_com1.background	Farbname aus rgb.txt	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*job_com1.width	Cardinal	70	Breite des Buttons s.o.
Xartus*job_com2			Ressourcen für den Button zum Schließen des Eingabeformulars der Iconobjekte ohne Speicherung. Gleiche Ressourcen wie Xartus*job_com1.
Xartus*job_com3			Ressourcen für den Button zum Aufruf der Iconauswahl im Eingabeformular der Iconobjekte. Gleiche Ressourcen wie Xartus*job_com1.

9.1.9 Abschnitt 9: Typ-Objekte

Die Typobjekte unterscheiden sich nur geringfügig von den Iconobjekten. Aus diesem Grund benutzen sie auch, bis auf wenige Ausnahmen, dieselben Ressourcen. Veränderungen dieser Ressourcen wirken sich somit auf die Dialogobjekte beider Klassen aus.

Ressourcen-Name	Wertebereich	Default	Erläuterung
Xartus*job_box_text1*font	Font	-misc-fixed-*-*-*-*-14-*-*-*-*-*-	Schriftart für den Eingabebereich des Namens im Eingabeformular der

		-	Typobjekte.
Xartus*job_label2			Referenz auf gleichnamige Ressource der Iconobjekte.
...			Siehe Ressourcen der Iconobjekte.

9.1.10 Abschnitt 10: Code-Formular

Ressourcen-Name	Wertebereich	Default	Erläuterung
Xartus*code_root*font	Font	-misc-fixed-*-*-*-*-14-*-*-*-*-*-*-*	Schriftart für die Eingabe des Codes im Code-Formular des Root-Objekts.

9.1.11 Abschnitt 11: Typ-Formular

Ressourcen-Name	Wertebereich	Default	Erläuterung
Xartus*new_ModelJob*label	Text	New	Beschriftung des Buttons zum Erzeugen eines neuen Typs im Typ-Formular des Root-Objekts.
Xartus*new_ModelJob*width	Cardinal	70	Breite des Buttons s.o.
Xartus*new_ModelJob.background	Farbname aus rgb.txt.	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*edit_ModelJob			Ressourcen für den Button zum Editieren eines Typs im Typ-Formular s.o.
Xartus*delete_Model			Ressourcen für den Button zum

Job			Löschen eines Typs im Typ-Formular s.o.
Xartus*close_ModelJob			Ressourcen für den Button zum Schließen des Typ-Formulars s.o.

9.1.12 Abschnitt 12: Icon-Formular

Ressourcen-Name	Wertebereich	Default	Erläuterung
Xartus*new_ModelJobIcon*label	Text	New	Beschriftung des Buttons zum Erzeugen eines neuen Iconobjekts im Icon-Formular des Root-Objekts.
Xartus*new_ModelJobIcon*width	Cardinal	70	Breite des Buttons s.o.
Xartus*new_ModelJobIcon.background	Farbname aus rgb.txt.	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*edit_ModelJobIcon			Ressourcen für den Button zum Editieren eines Iconobjekts im Icon-Formular s.o.
Xartus*delete_ModelJobIcon			Ressourcen für den Button zum Löschen eines Iconobjekts im Icon-Formular s.o.
Xartus*close_ModelJobIcon			Ressourcen für den Button zum Schließen des Icon-Formulars s.o.

9.1.13 Abschnitt 13: Icon-Auswahl

Ressourcen-Name	Wertebereich	Default	Erläuterung

Xartus*ok_icon.label	Text	Ok	Beschriftung des Buttons zum Schließen der Icon-Auswahl.
Xartus*ok_icon.background	Farbname aus rgb.txt.	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*ok_icon.width	Cardinal	90	Breite des Buttons s.o.
Xartus*set_for			Ressourcen für den Button zum Festlegen der Vordergrundfarbe eines Icons. Gleiche Ressourcen wie Xartus*ok_icon.
Xartus*set_back			Ressourcen für den Button zum Festlegen der Hintergrundfarbe eines Icons. Gleiche Ressourcen wie Xartus*ok_icon.
Xartus*set_hight			Ressourcen für den Button zum Festlegen der Highlightfarbe eines Icons. Gleiche Ressourcen wie Xartus*ok_icon.

9.1.14 Abschnitt 14: Icon-Manager

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*creat_ic.label	Text	NewIcon	Beschriftung des Buttons zum Aufruf des Iconeditors.
Xartus*creat_ic.width	Cardinal	60	Breite des Buttons s.o.
Xartus*edit_ic+			Ressourcen des Buttons zum Editieren eines Icons. Gleiche Ressourcen wie Xartus*creat_ic.

Xartus*delet_ic			Ressourcen des Buttons zum Löschen eines Icons. Gleiche Ressourcen wie Xartus*creat_ic.
Xartus*ok_icon+			Ressourcen des Buttons zum Schließen des Icon-Managers. Gleiche Ressourcen wie Xartus*creat_ic.

9.1.15 Abschnitt 15: Buffer

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*buffbox.background	Farbname aus rgb.txt.	grey90	Hintergrundfarbe des Buffer-Formulars.
Xartus*buffbox_label.font	Font	-*-courier-medium-r-normal--10-*-*-*-*-*-*-*	Schriftart einer Zeilenbeschriftung eines Buffers.
Xartus*buffbox_label.borderWidth	Cardinal	1	Randbreite einer Zeilenbeschriftung eines Buffers.
Xartus*buffbox_label.internalWidth	Cardinal	2	Abstand einer Zeilenbeschriftung eines Buffers zum Rand seiner Darstellungsfläche.
Xartus*buffbox_label*background	Farbname aus rgb.txt	grey80	Hintergrundfarbe einer Zeilenbeschriftung eines Buffers.
Xartus*buffbox_label.shadowWidth	Cardinal	2	Breite der Schattierung einer Zeilenbeschriftung eines Buffers durch die 3D-Athena-Widgets.
Xartus*buffbox_label	Font	-*-courier-	Schriftart eines Buffereintrags.

1*font		medium-r-normal--10-*-*-*-*-*-*-*	
Xartus*buffbox_label1.borderWidth	Cardinal	0	Randbreite eines Buffereintrags.
Xartus*buffbox_label1*internalWidth	Cardinal	4	Abstand eines Buffereintrags zum Rand seiner Darstellungsfläche.
Xartus*buffbox_label1*justify	Orientierung (Left | Right | Middle).	left	Orientierung eines Buffereintrags.
Xartus*buffbox_label1*background	Farbname aus rgb.txt.	grey90	Hintergrundfarbe eines Buffereintrags.
Xartus*buffbox_label1.shadowWidth	Cardinal	0	Breite der Schattierung eines Buffereintrags durch die 3D-Athena-Widgets.

9.1.16 Abschnitt 16: File-Box

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*select.label	Text	Select	Beschriftung des Buttons zur Auswahl in der File-Box.
Xartus*select.background	Farbname aus rgb.txt.	grey80	Hintergrundfarbe des Buttons s.o.
Xartus*path.label	Text	Path	Ressourcen des Buttons zu Auswahl eines Directory-Pfads. Gleiche Ressourcen wie Xartus*select.
Xartus*cancel.label	Text	Cancel	Ressourcen des Buttons zum Abbruch der File-Box. Gleiche Ressourcen wie Xartus*select.

Xartus*fileServer*initialDirectory	Text	../Modells/	Pfad zum Verzeichnis mit den Modelldaten (relativ zum Verzeichnis von Xartus).

9.1.17 Abschnitt 17: Dialog-Objekte

Ressourcen-Name	**Wertebereich**	**Default**	**Erläuterung**
Xartus*dialogobject_com1.label	Text	OK	Beschriftung des Button für die positive Auswahl eines Dialog-Objekts.
Xartus*dialogobject_com2.label	Text	CANCEL	Beschriftung des Button für die negative Auswahl eines Dialog-Objekts.
Xartus*printobject_com.label	Text	OK	Beschriftung des Button für die Bestätigung eines Print-Objekts.
Xartus*textobject_label.borderWidth	Cardinal	0	Randbreite des Textbereichs eines Text-Objekts.
Xartus*textobject_com1.label	Text	OK	Beschriftung des Button für die positive Auswahl eines Text-Objekts.
Xartus*textobject_com2.label	Text	CANCEL	Beschriftung des Button für die negative Auswahl eines Text-Objekts.
Xartus*listobject_label.borderWidth	Cardinal	0	Randbreite eines List-Objekts.
Xartus*listobject_com1.label	Text	OK	Beschriftung des Button für die positive Auswahl eines List-Objekts.
Xartus*listobject_com2.label	Text	CANCEL	Beschriftung des Button für die negative Auswahl eines List-Objekts.

9.1.18 Abschnitt 18: Canvas

Xartus*canvas_text.shadowWidth	Cardinal	0	Breite der Schattierung bei den dargestellten Text- und Iconobjekten in einem Canvas.
Xartus*canvas_text.font	Font	-*-helvetica-medium-r-normal-*-12-*-*-*-*-*-*	Verwendeter Font der Textobjekte.
Xartus*c_button.label	Text	Close	Beschriftung des Buttons zum Schließen eines Canvas.

9.1.19 Abschnitt 19: Modell-Manager

Xartus*mm_com1.label	Text	Remove	Beschriftung des Buttons zum Löschen einer Modelldatei.
Xartus*mm_com2.label	Text	Rename	Beschriftung des Buttons zum Umbenennen einer Modelldatei.
Xartus*mm_com3.label	Text	Close	Beschriftung des Buttons zum Schließen des Modell-Managers.

9.1.20 Abschnitt 20: Options

Literatur

[Block95] Block, G.: *Entwurf und Implementierung für das grafische System Grasival*, Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, 1995

[Brede91] Brede, H.-J., Josuittis, N., Lemberg, S., Lörke, A.: *Programmieren mit OSF/Motif*, Addison-Wesley, 1991

[Gottheil92] Gottheil, K., Kaufmann, H.-J., Kern, T., Zhao, R.: *X und Motif: Einführung in die Programmierung des Motif-Toolkits und des X-Window-Systems*, Springer, Berlin, 1992

[Jones91] Jones, O.: *Einführung in das X-Window-System*, Hanser, München, 1991

[Kernighan90] Kernighan, B. W., Ritchie, D. M.: *Programmieren in C*, Hanser, München, 1990

[Kowalk94] Kowalk, W.: *Simulation in C-SIM*, Rechnernetze und Tele-Kommunikation, Universität Oldenburg, 1994

[Kowalk93] Kowalk, W.: *Principles of Modeling Simulation-Techniques and Simulation-Tools*, Rechnernetze und Tele-Kommunikation, Universität Oldenburg, 1993

[Ono-Tesfaye93] Ono-Tesfaye, T., *Dokumentation zu X-SimView*, Studienarbeit, Fachbereich Informatik, Universität Oldenburg, 1993

[O’Reilly90/1] O’Reilly, T., Nye, A.: *X Toolkit Intrinsics Programming Manual*, Second Edition, Volume 4, O’Reilly & Associates, Inc., 1990

[O’Reilly90/2] O’Reilly, T.: *X Toolkit Intrinsics Reference Manual*, Second Edition, Volume 5, O’Reilly & Associates, Inc., 1990

[O’Reilly90/3] O’Reilly, T., Nye, A.: *X Toolkit Intrinsics Programming Manual*, OSF/Motif Edition, Volume 4, O’Reilly & Associates, Inc., 1990

[Sterzenbach95] Sterzenbach T.: *Xartus-Eine graphische Benutzungsoberfläche zur interaktiven Erstellung, Steuerung und Auswertung eines Simulationsprogramms für Hochgeschwindigkeitsprotokolle*, Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, 1995