G++ – Neue Wege in der Informationstechnologie

Hardwareevolution

Bis vor ungefähr 2500 Jahren standen der Menschheit lediglich 10 Finger für die Lösung mathematischer Probleme zur Verfügung.

Das erste mechanische Rechenhilfsmittel, das sogenannte Rechenbrett oder auch Abakus genannt (lat. von abax, die Tafel) startete bereits 500 vor Christi Geburt von Babylon aus seinen weltweiten Siegeszug. Vor allem in China entwickelte sich die Kunst im Umgang mit den Rechenbrettern (Suanpan) zur wahren Meisterschaft.

Rechenbretter waren bis ins 17. Jahrhundert hinein das konkurrenzlose Kalkulationsgerät schlechthin.

Erst Blaise Pascal gelang es 1641/42 eine mechanische Rechenmaschine zu bauen, mit der man Additionen und Subtraktionen durchführen konnte. Das Universalgenie Gottfried Wilhelm Leibniz entwickelte 1671 mit Hilfe der sogenannten Staffelwalze eine funktionierende Maschine für alle vier Grundrechenarten. Leibniz, der seiner Zeit in sehr vielen Disziplinen sehr weit voraus war schuf unter anderem durch die Entwicklung des Binärsystems die Basis für die moderne Informationstechnologie. Die von ihm eingeführte Infinitisimalmathematik (Integration und Differentiation) ist die Basis für die moderne höhere Mathematik.

Da die Systeme unter der eingeschränkten Reproduzierbarkeit der Einzelbauelemente litten, war die Verbreitung von Rechenautomaten bis ins Industriezeitalter sehr eingeschränkt.

Erst ab den 90er Jahren des 19. Jahrhunderts stellte sich der Erfolg der Rechenmaschinen ein.

Vor allem Hermann Holerith markierte mit der Einführung seiner lochkartengesteuerten Zählmaschine einen neuen Meilenstein in der Geschichte der Kalkulationsautomaten. Holerith stützte sich bei der Entwicklung seiner Maschine unter anderem auf die Entwicklungen des britischen Mathematiker Charleas Babbage, der bereits 1833 das Konzept eines Computers skizzierte, das bereits alle wichtigen Bestandteile (Speicher, ALU, Steuereinheit, Ausgabegerät) eines modernen Rechners hatte.

Es dauerte noch knapp 60 Jahre, bis Konrad Zuse mit der Z3 den ersten richtig funktionierenden programmgesteuerten Rechenautomaten vorstellte, basierend auf Relais.

Kurz nach Ende des 2. Weltkrieges entwickelten sich die Maschinen und die zu Grunde liegenden Komponententechnologien immer schneller. 1941 erschienen die ersten in Stückzahlen kommerziell verfügbaren – auf Elektronenröhren basierenden - Rechner, mit bereits über 1000 Additionen pro Sekunde. Rechner der 2. Generation hatten bereits Halbleiterbauelemente (Transistoren und Dioden) und erreichten schon 1957 etwa die zehnfache Leistung der Maschinen der 40er. Mit der Entwicklung integrierter Schaltkreise 1958 durch Jack Kilby eröffneten sich schlagartig völlig neue Horizonte. Erste Rechner waren bereits Anfang der 60er verfügbar. Seit dieser Zeit gibt es eine Reihe von Generationssprüngen in der Entwicklung der Rechner. Mitte der 80er gab es eine Reihe von interessanten Ansätzen die Computertechnologie zu neuen Ufern zu führen. Vor allem die RISC-basierten Transputer mit ihrer MIMD-Architektur (Multiple Instruction Multiple Data) schienen mit den Flaschenhälsen der Von-Neu-

mann- und Harvard-Architekturen Schluss zu machen. Trotz zahlreicher Unzulänglichkeiten konnten sich jedoch Intel basierte Systeme bis jetzt durchsetzen. Knapp 20 Jahre nach Veröffentlichung des Transputer Konzeptes basierend auf Tony Hoares CSP-Modell (Communicating Sequential Processes) entwickeln die Halbleiterfirmen neue serielle Hochgeschwindigkeitsschnittstellen, die Prozessorknoten miteinander verbinden. Ironischerweise bezeichnet nicht nur Intel diese Schnittstellen als „Links“.

Softwareevolution textbasierend

Die Geschichte der Softwareentwicklung ist noch wesentlich undurchsichtiger als die der Hardwareevolution.

Seit der Erfindung des ersten Computers wurden ungefähr 1000 (!) Programmiersprachen kreiert.

Die erste „richtige“ Programmiersprache entwickelte Konrad Zuse zwischen 1942 und 1945. Er nannte diese Sprache „Plankalkül“. Die Ideen Plankalküls hatten nur wenig Einfluss auf die Programmiersprachen, die Anfang der 60er wie Pilze aus dem Boden spriessen.

Viele der sehr lange verfügbaren Sprachen wie COBOL (im kaufmännischen Bereich) und Fortran (im Bereich wissenschaftlicher Software) halten sich bis jetzt hartnäckig. Auch Assembler ist vor allem im Bereich der Mainframes nicht zu ersetzen.

Vor allem Algol stand Pate für eine Vielzahl von Sprachen, wie z.B. Pascal, C, Simula, PL/1 und ADA. Letztere Sprache implementierte viele Elemente modernen Softwareengineerings, scheiterte jedoch letztendlich an der eigenen Komplexität.

Die Weiterentwicklung dieser imperativen Sprachen mündeten in den siebziger Jahren in erste objektorientierte Ansätze. Simula führte zwar erste Begriffe der Objektorientierung durch Klassen und Objekte ein, einen wesentlich grösseren Einfluss auf die Entwicklung der Objekttechnologie hatte jedoch Smalltalk. Nahezu alle modernen textbasierten Sprachen wie C++ und Java nehmen Anleihen an Smalltalk. Die Majorität der grossen kommerziell verfügbaren Softwarepakete basieren derzeit auf C++. Java kann man als einfacher zu programmierende C++ - Variante bezeichnen, da hier der umständliche Umgang mit Pointern und Pointern auf Pointern und Pointern auf Pointer auf Pointer entfällt.

Nahezu allen textbasierten Sprachen gemeinsam (abgesehen von exotischeren Paradigmen wie OCCAM und Strand88) ist die eindimensionale Strukturierung der Programme, die eine sequentielle Abarbeitung erzwingt. Der Programmierer muss sich mit kleinsten semantischen und syntaktischen Details auseinandersetzen um eine Basis für Problemlösungen zu erhalten.

Softwareevolution grafisch

Grafische Programmierung unterscheidet sich fundamental von herkömmlichen textorientierten Entwicklungstechniken. Die Ursprünge der grafischen Programmierung liegen in der Zeit vor den Digitalrechnern zurück.

Analogrechner mit ihren funktionalen Blöcken wie Addierer, Multiplizier, Differenzierer, Integrierer etc., sowie der Verbindung untereinander mit Hilfe von Drähten bilden den gedankliche Basis für die Entwicklung grafischer Programmiertechniken.

Auch die papierbasierten Beschreibungsmittel zur Formulierung von Programmkonstrukten wie Flußdiagramme, Nassi Schneidermann Diagramme, Petri Netze, u.v.a.m. beeinflußten die Designer grafischer Entwicklungstools.

Der Idealfall einer grafischen Datenflußprogrammierumgebung ist die Zusammenfassung von Problembeschreibung, Problemlösung und aussagekräftiger, selbsterklärender Dokumentation in einem Arbeitsgang. Das Werkzeug sollte ein, mit herkömmlichen Programmiersprachen vergleichbares Laufzeitverhalten haben, eine deklarative- und übersichtliche-, gut strukturierte Darstellung gewährleisten und vor allem Menschen mit unterschiedlichen informationstechnischen Vorkenntnissen einen natürlichen Zugang gewähren.

Mit der Fortentwicklung der Rechnersysteme standen den Softwareentwicklern Plattformen zur Verfügung, die eine Abstrahierung in Form grafischer Elemente ermöglichte. Das Wort „Visual“ kam immer mehr in Mode.

Viele Programmiersprachen jedoch, die „Visual“ in ihrem Namen stehen haben, haben rein gar nichts mit grafischer Programmierung zu tun. Vor allem die Programmiersprachen Visual C++ und Visual Basic bilden hier keine Ausnahme. IBMs Visual AGE bestach zwar durch einige grafische Ansätze, war jedoch weit von einer vollständigen Implementierung einer Programmiersprache entfernt. Die meisten Entwicklungsumgebungen nutzen einen grafischen Ansatz, um mit CASE-Implementierungen Rumpfprogramme zu erzeugen. Die Codierung der Low-Level-Funktionalität geschieht in der Regel jedoch mit textbasierten Beschreibungsmechanismen.

Nahezu alle wirklichen grafischen Tools haben ihren Elfenbeinturm bis heute nicht verlassen.

Der National Instruments Fellow Jeff Kodosky revolutionierte mit seiner Crew Anfang der achtziger Jahre das Mensch-Maschinen-Interface auf der Basis einer grafischen Datenflußmaschine, das nahezu alle Vorteile der modernen grafischen Datenflußprogrammierung in sich vereint, die Entwicklungsumgebung LabVIEW (Laboratory

Virtual Instrument Engineering Workbench), mit seiner immanten Programmiersprache G.

Mittlerweile hat sich LabVIEW zum De-Facto-Standard für grafische Entwicklungsumgebungen im Bereich der technischen Informationstechnologie entwickelt. LabVIEW läuft textbasierten Entwicklungssystemen (C, C++, Basic, Java, Pascal, etc.) bei technischen Anwendungen (vor allem in der Mess- und Automatisierungstechnik) immer mehr den Rang ab. LabVIEW ist die einzige vollständige und kommerziell verfügbare grafische Entwicklungsumgebung weltweit.

Virtuelle Instrumente

LabVIEW ist eine Anwendungsentwicklungsumgebung, die sich von konventionellen Programmiersprachen wie C oder Basic vor allem in der Art der Kodierung unterscheidet. Im Unterschied zu den textbasierenden Sprachen entwickelt man in LabVIEW Programme grafisch, und zwar mit Hilfe von Blockdiagrammen und Frontplattenelementen. Blockdiagrammbestandteile sind mit Hilfe eines sogenannten Wiring Tools (Verbindungsdrähten) miteinander zu verbinden, damit ein voll funktionsfähiges Virtuelles Instrument entstehen kann.

LabVIEW ist in erster Linie eine Meßwerterfassungs- und -managementprogramm, das auf applikationsspezifische Bibliotheken für die Steuerung von Einsteckkarten und externen Mess- und Automatisierungssystemen zurückgreift. Zusätzliche Libraries zur Datenrepräsentation, Datenanalyse und Filemanagement, sowie eine Reihe von konventionellen Entwicklungshilfsmitteln runden das Bild einer komplexen Programmierumgebung ab.

In Analogie zu den Frontplatten von Meßgeräten, welche Bedien- und Anzeigeelemente zur Verfügung stellen, und als eigentliche Schnittstelle zwischen Anwender und Gerät dienen, ist das Frontpaneel (*Front Panel)* die Bedienoberfläche eines Programmes, das Anwendern ermöglicht, mit dem Programm in Interaktion zu treten und die Ergebnisse der Anwendungen zu visualisieren.

Die Funktionalität eines Meßgerätes ist durch elektronische und elektrische Komponenten bestimmt, die mit Hilfe elektrischer Schaltkreise miteinander in geeigneter Arte und Weise verbunden sind. Analog dazu ist das sogenannte Blockdiagramm (*Block Diagram)* eine Verbindung von grafischen Symbolen mit Hilfe von softwarebasierten Datenleitungen in Form von Drähten.

Das *Block Diagram* beinhaltet den eigentlichen Programmcode - einen grafischen Quellcode, den der Entwickler im Gegensatz zum konventionellen, textorientierten Quellcode als Blockschaltbild „zeichnet“. Bestandteile dieses Blockschaltbildes können sowohl einfache mathematische Operatoren wie Addierer, Multiplizierer, etc. sein, als auch komplexe Funktionen wie FFT oder selbstdefinierte SubVIs (Unterprogramme) sein. Die Schnittstellen eines SubVIs zum aufrufenden Programm nennt man Terminals. Terminals sind entweder als Quellen oder als Senken ausgelegt.

Blockdiagramme sind grundsätzlich in drei Elementgruppierungen einzuteilen. Knoten (nodes) sind Programmausführungelemente, die mit Statements, Operatoren, Funktionen und Subroutinen konventioneller Programmiersprachen zu vergleichen sind. ,

Terminals sind Ports, durch welche die Kommunikation zwischen Blockschaltbild und Frontplatte erfolgt. Alle Ausführungselemente besitzen Terminals. Die sogenannten Wires (Datenflußverbindungsdrähte) bilden die Verbindungspfade zwischen Ein- und Ausgangsterminals der einzelnen Programmausführungselemente.

In LabVIEW ist eine Vielzahl von Terminaltypen implementiert. Generell kann man sagen, daß ein Terminal jeden Punkt darstellt, an den man einen Flußdraht (Wire) anbringen kann. LabVIEW besitzt Kontroll- und Anzeigeterminals, Node Terminals, Konstanten und spezielle Terminals bei Strukturen.

LabVIEW bietet eine Fülle von Funktionen. Neben einer Reihe von Schleifenkonstrukten und (Natur-)Konstanten, arithmetischen-, trigonometrischen-, logarithmischen-, Vergleichs-, Konvertierungs-, String-, Array-, Cluster-, File I/O- und Dialogfunktionen und -Bibliotheken sind noch die Sub-VI und Standardanalysefunktionen zu erwähnen, die durch eine Vielzahl von zusätzlichen Optionen (unter anderem Debugging und Error-Handling) ergänzt werden.

Mit dem sogenannten Wiring Tool (dargestellt als Drahtrolle) - angewählt mit einem Doppelklick auf der Menü-Optionsleiste - realisiert man die notwendigen Datenpfade zwischen einzelnen funktionellen Einheiten, wobei auf die Art der zu verbindenden Terminals zu achten ist (es dürfen zum Beispiel - in Analogie zur Schaltungtechnik - nicht zwei Quellen verbunden sein).

Structures

Wie bei den konventionellen Programmierumgebungen üblich besitzt auch LabVIEW eine Reihe von Strukturkonstrukten, die in modularen Systemen unumgänglich sind. Vor allem Schleifenkonstrukte wie For- und While Loops, sowie sequentielle- und Case-Strukturen sind in jeder moderneneren Programmiersprache anzutreffen. Vor allem die CASE-Anweisungen sind vorbildlich implementiert. Die sogenannten Selectors (Zuordnungszeiger für Case-Konstrukte) können vom Typ String- Enumerated-Type-, Boolean-, Integer- oder Floating-Point sein. Die sequentiellen Konstrukte (Sequence) führen mit Hilfe einer optimalen Abstrahierung des Programmierproblemes zu einer klar gegliederten Strukturierung des Programmflusses.

Formula Node

In den wenigen Fällen, in denen die Lösung eines mathematischen Problemes mit den Standard-Möglichkeiten LabVIEWs zu aufwendig ist, bietet sich die Option ´Formula Node´ an. Mit dem Formelknoten ist es möglich mathematische Ansätze, Gleichungen und Funktionen schnell zu realisieren. Die Funktionen sind vollständig, die in der BNF-Notation angegebene Syntax und die Operatoren sind sehr einprägsam dargestellt. LabVIEW besitzt eine Reihe von verschiedenen Formalknoten inklusive eines Matlab-Knotens.

Hierarchisch und modular

Strukturierte Programmierung ist gekennzeichnet durch einen durchdachten Einsatz von Unterprogrammen. Übersichtlichkeit, Modularität und leichte Wartbarkeit sind der Lohn eines strukturierten Designansatzes. Um einen möglichst grossen Benefit zu generieren, muß ein exakt definierter Abstraktionsmechanismus hinsichtlich der Datenübergabe zwischen Modulen vorhanden sein. In LabVIEW wird diese Anforderung durch das Korrespondieren eines Diagramms (*Block Diagram*) mit einem Symbol (*Icon/Connector*) erfüllt. Ein solches, vom Anwender frei definierbares Symbol, hält Anschlüsse (*Terminals*) bereit, die den Datenquellen und -senken des Diagramms zuzuweisen sind.

Ein SubVI läßt sich sehr gut vergleichen mit einem integrierten Schaltkreis, der ja auch über verschiedene Ein- und Ausgänge beschaltet werden kann. Auf ein derartiges Modul ist von einem übergeordneten Programm aus, über dessen Icon beliebig oft zuzugreifen.

Der Modul- und Hierarchiebildung sind in LabVIEW nur die Grenzen der 32-bit Architektur gesetzt, wobei eine Begrenzung bezüglich der Tiefe an Verschachtelungen in Applikationen nicht auftritt. Einen Überblick über die Hierarchiestruktur von LabVIEW VIs geben die sogenannten Hierarchiefenster. Diese Fenster sind ein ideales Hilfsmittel, um sich in komplexen Programmstrukturen zurechtzufinden.

Parallel und gleichzeitig

Die in LabVIEW eingebettete Datenflußmaschine stellt Parallelisierungs- und Multitaskingkonstrukte zur Verfügung. Anders als bei der herkömmlichen textbasierten Programmiersprachen, die von Natur aus sequentiell arbeiten und nur durch aufwendige Programmierung (z.B. mit Hilfe von Semaphoren) parallelisierten Code erzeugen können, ist es in LabVIEW sehr leicht möglich, konkurrierende, einzeln ablaufende und leicht synchronisierbare, Programmteile zu erzeugen. Der einfachste Fall ist die Plazierung von ein oder mehreren bereit vorhandenen Virtuellen Instrumenten auf ein Diagramm, um die gleichzeitige Ausführung von VIs in einer Applikation zu ermöglichen. Die Ursache liegt in der Tatsache begründet, daß LabVIEW im Unterschied zu textbasierten Paradigmen, die auf Kontrollflußmechanismen basieren, rein datenflußgetrieben ist. Der datenflußbasierte Ansatz ist eine wesentlich elegantere und natürlichere Art der Formulierung von Problemlösungen.

Anstatt sich auf irrelevante implementierungsspezifische Parallelisierungmechanismen stürzen zu müssen, ist es einem G-Programmierer möglich, Probleme zu formulieren und dabei Lösungen zu erhalten.

LabVIEW ermöglicht, das Konzept der Parallelisierung von Virtuellen Instrumenten auf Multiprozessorsysteme auszudehnen. Das Tool benutzt Multithreading, um die optimale Aufteilung von Einzeltasks innerhalb einer Applikation zu gewährleisten.

LabVIEW supported Multithreading auf allen wichtigen Betriebssystemen, die preemptives Multitasking unterstützen (Windows 2000/XP, UNIX/Linux, etc.).

Plattformunabhängigkeit

Trotz der extrem weiten Verbreitung der Windows Betriebssysteme konnte sich bis heute keine einzige Plattform als echter Standard etablieren. Ein Grund dafür sind sicherlich die unterschiedlichen Vor- und Nachteile, die den einzelnen Systemen zu eigen sind. Um so wichtiger ist es für eine Softwarelösung, daß die am weitest verbreiteten Rechnersysteme unterstützt und deren systemspezifische Eigenschaften genutzt werden können. Von einer leistungsfähigen Umgebung darf der Anwender also erwarten, daß ein auf der einen Plattform entwickeltes Programm ohne zeitraubende Änderungen auf einer anderen Hardwareumgebung lauffähig ist. Einschränkungen seitens der Hardware sind hier natürlich die bereits erwähnten systembedingten Eigenschaften wie z.B. DLLs die nur unter MS Windows unterstützt werden können, nicht aber auf Apple Macintosh oder HP Workstation, während ein TCP/IP Protokoll auf mehreren Plattformen existiert und somit auch portierbar sein muß. Unter LabVIEW sind diese Anforderungen weitestgehend erfüllt.

Offenheit

Offenheit wird in Zusammenhang mit Softwareumgebung häufig als Schlagwort gebraucht und hat sich zu einer Art Modewort der Branche entwickelt. Doch beim näheren Betrachten gewährleisten viele, als offen angepriesene, Systeme nicht das, was der Anwender unter Offenheit erwartet. Offenheit kann im praktischen Einsatz einer SW-Entwicklungsumgebung viele Bedeutungen haben, wobei jeder Anwender seine individuellen Schwerpunkte setzen wird. Folgende Merkmale, deren Gewichtung applikationsabhängig ist, sind häufig gefordert:

- Standardisierte Systemeigenschaften wie ActiveX, DDE, DLL, TCP/IP, shared libraries etc. müssen unterstützt werden
- Es muß eine Möglichkeit geben, in Form einer offengelegten Schnittstelle, eigene Speziallösungen in Form von vorhandenem Quellcode in die Umgebung einzubetten
- Gängige I/O Schnittstellen wie RS232, RS422, RS485, GPIB, PCI, AT-Bus, PCMCIA (PCCARD), Cardbus, VXI, PXI, USB, Irda, Fire Wire, CAN, Profibus o.ä. sind zu unterstützen
- SW-Treiber für solche I/O Schnittstellen, die die Entwicklungszeit von Programmen drastisch verkürzen können, sollen jedem Entwickler zugänglich sein.

LabVIEW ist für eine Vielzahl von Geräten (seriell, IEEE 488, VXI, etc.), Bussystemen (PCI, PXI/Compact PCI, ISA/EISA, PCMCIA, MXI etc.) und Datenerfassungskarten (Multi-I/O, A/D, D/A, CTR, Dig I/O, DSP, etc.) offen. Durch die Unterstützung standardisierter Systemeigenschaften ist es beispielsweise über DLLs möglich, beliebige I/O-Schnittstellen zu nutzen (wie z.B die in der Prozess-Visualisierung und Steuerung weitverbreiteten Feldbusse wie Interbus-S, CANBus etc.).

Graphischer Compiler

Was nützt einem jedoch ein universelles System, wenn die Ausführungsgeschwindigkeiten der Applikationen zu gering sind. Nahezu alle grafische Werkzeuge setzen (im Unterschied zur G-Implementierung) auf einen Interpreter. Interpretierter Code läuft jedoch vergleichsweise langsam ab. Ein Compiler über-

setzt den vom Programmierer generierten Quellcode in mehr oder minder optimierten Maschinencode und gewährleistet somit eine raschere Programm-Abarbeitung als ein Interpreter. Immer leistungsfähigere Hardware und schneller getaktete Prozessoren erlauben zwar, daß Interpreter für manche Applikationen ausreichend sind, doch der typische Industrieeinsatz von Rechnern, auf Gebieten, die noch vor wenigen Jahren Spezialgeräten vorbehalten waren, setzt die Leistungsfähigkeit eines optimierten Compilers voraus. Die unter LabVIEW erstellten Blockdiagramme werden von einem „graphischen" Compiler in optimierten Maschinencode übertragen. G Programme stehen C++ Code in Nichts nach.

Qualität

Anders als in konventionellen Programmiersprachen ist es in LabVIEW möglich, Einzelmodule (virtuelle Instrumente) vollständig zu testen, um sie später in hierarchisch höher stehenden Virtuellen Instrumenten einzusetzen. Die ausgeprägte Modularität und Hierarchie führt zu einer, mit konventionellen Systemen nicht erreichbaren, Wiederverwendbarkeit des Programmcodes. Im Gegensatz zu den ständigen Veränderungen im Bereich der sogenannten APIs (Application Programming Interfaces) von Betriebssystemen und der Inflation von neuen und veränderten Klassenbibliothekselementen gibt es in LabVIEW nur Erweiterungen des Funktions- und Bibliotheksumfanges, sowie Performanceverbesserungen. Die sogenannten Revolutionen in der Softwaretechnologie erfolgen - vor dem Entwickler verborgen - in den Tiefen des LabVIEW Systems. Die Einbindung der Funktionalität von sogenannten Add-In-Modulen (z.B. in Form von Code Interfaces Nodes oder Dynamic Link Libraries) ist bei LabVIEW jedoch jederzeit möglich. In der Regel greift man hier auf bereits bestehende DLLs zurück oder editiert bestehende DLLs.

Dies führt zu einer außergewöhnlich guten Sprachbeherrschung bei gleichzeitig geringem Einarbeitungsaufwand.

Der ´G´-Programmierer baut auf bekanntem, gewachsenem Code auf und hat eine weitaus höhere Sicherheit, Qualität und Zuverlässsigkeit bei der Implementierung neuer ´Parts´ als ein Entwickler herkömmlicher Software. Da sich LabVIEW-Entwickler durch die Abschirmung von implementierungsspezifischen Details voll auf die Anwendungsentwicklung konzentrieren können, entsteht ein größeres Vertrauen in das Laufzeitverhalten der entwickelten Software. Die ganze Intuition, die Kreativität und die Energie des Anwendungsentwicklers fließt in die Funktionalität, Ergonomie, Performance, Qualität, Zuverlässigkeit und Verfügbarkeit der Applikation.

Durch die Konzentration auf die Qualität, Zuverlässigkeit, Funktionalität und vor allem auf die Ergonomie entstehen Applikationen, die den Endanwender ein Höchstmaß an Flexibilität und Anwendungssicherheit bieten. Ein „beherrschter" Code ist sehr einfach zu warten, das heißt, daß die sogenannte Maintainability sehr ausgeprägt ist. Eine optimale Maintainability führt dazu, daß Programmentwickler viel eher bereit sind, ihre Programme zu optimieren. „G"-Programmierer lernen spielerisch den Umgang mit ihrem System und mit Problemlösungen. Durch das ausgeprägte „Rapid Prototyping" kann man in verschiedenen Entwicklungsphasen unterschiedliche Optionen durchspielen und kommt so - im Vergleich zu herkömmlichen Ansätzen - in der Regel zu effizienteren Lösungen. Rapid Prototyping hat aber auch noch andere gravierende Vorteile. Durch die Programmentwicklungsgeschwindigkeit gewinnt der Programmierer schnell umfassende Kenntnisse über das Laufzeitverhalten von „G". Laufzeitfehler treten dadurch wesentlich seltener auf als bei konventionellem Code.

Eine der Domänen der grafischen Flußprogrammierung mit LabVIEW ist das Laufzeit-Fehlermanagement. Mittels sogenannter Error Clusters (vergleichbar mit sogenannten Structures konventioneller Programmiersprachen wie Pascal und Fortran) ist es sehr leicht möglich, Fehlerpositionen, Fehlerarten

und Fehlerbeschreibungen zu „handeln“. Dadurch ist es sehr leicht möglich, auf verschiedene Fehlerklassen angemessen programmtechnisch zu reagieren. In der Praxis bedeutet dies, daß beim Auftreten eines minder schweren Fehlers, das Gesamtsystem nicht unbedingt versagen muß. Macht der Entwickler ausgiebig Gebrauch von den Fehlermanagementmöglichkeiten, so erzeugt er eine vertrauenserweckendere und ausführungssicherere Applikation. Der geringe Mehraufwand den man betreiben muß (Error Clusters müssen zwischen VIs und SubVIs im Daisy Chain Verfahren verbunden werden) wird durch qualitativ hochwertigeren und weitaus zuverlässigeren und beherrschbareren Code mehr als ausgeglichen.

Neben dem Laufzeit-Fehlermanagement sind für eine grafische Programmiersprache leistungsfähige Fehlererkennungsfunktionen unerläßlich. Hier bietet LabVIEW eine Fülle von komfortablen Debuggingfunktionen, wie zum Beispiel Breakpoints, Probes, sowie Step In/Over/Out-Funktionalitäten für Unterprogramme, welche die Fehlersuche in größeren Applikationen erheblich erleichtern. Mit Breakpoints ist es möglich, den Programmablauf an beliebigen Stellen zu unterbrechen um so einen Überblick über den Programmstatus verschaffen zu können. Mittels Probes lassen sich aktuelle, zur Laufzeit generierte Daten visualisieren.

Mit Hilfe der Step In/Over/Out-Funktionen lassen sich Unterprogramme überspringen, verlassen oder man hat die Möglichkeit, in SubVIs zu springen. Eine ausgezeichnete Möglichkeit, dem Programmfluß visuell zu folgen ist die sogenannte Highlighting-Function, ein Tool, mit dessen Hilfe man den Datenfluß mittels wandernder Punkte (Bubbles) und einzelner Momentanwerte hervorragend analysieren kann.

Weiterhin besteht die Möglichkeit, mit Hilfe des sogenannten Profilers, die Speicherauslastung, die Unterprogrammaufrufhäufigkeit, und das Laufzeitverhalten von SubVIs zu analysieren. Besonders hilfreich ist der Profiler im Auffinden sogenannter Hot Spots (Stellen im Programm, die eine hohe Arbeitslast bewerkstelligen müssen) durch Analyse der tabellarisch wiedergegebenen SubVI-Statistik (Mittel-Min- und Maxwerte der Speicherauslastung, SubVI-Frequentierung, d.h. Anzahl der Abläufe).

Eine der Hauptvorteile LabVIEWs ist die deklarative Herangehensweise an die Problemlösung. Bei der grafischen Beschreibung des Problems erzeugt sich die Applikation automatisch. Als wertvolles „Abfallprodukt“ erhält man dabei die Dokumentation. Da die Code-Dokumentierung bei herkömmlichen Programmiersprachen eine sehr lästige Aufgabe darstellt, wird sie nicht immer konsequent betrieben. Vor allem Schnittstellenbeschreibungen sind oft nur sehr mühsam zu realisieren.

Bestehender konventioneller Code einer bestimmten Größenordnung ist dadurch kaum noch nachvollziehbar. Teambasierte Entwicklung ist durch diese Nachteile nur mit großen Einschränkungen möglich.

LabVIEW fördert durch deklarativen Code, Modularität und Hierarchie die Anwendungsentwicklung in Teams. Die Schnittstellen (in LabVIEW Terminals genannt) erzeugen LabVIEW Entwickler während der normalen Anwendungsentwicklung. Module, die von Teammitgliedern generiert wurden sind sehr einfach in eigene Applikationen einzubinden. Diese Eigenschaften der Programmiersprache „G“ haben dazu geführt, daß in den letzten Jahren eine Vielzahl von allgemeinen und spezialisierten VIs den Markt erreichten. Diese Virtuellen Instrumente haben den Vorteil, daß sie sozusagen „out of the box“ einsetzbar sind. Alle lästigen Aufgaben (Compilieren, Linken, Environmentmanagent, Speichermanagement, u.v.a.m.) managt LabVIEW - nicht transparent für den Entwickler - im Hintergrund.

Der Vorteil beim Einsatz solcher Virtuellen Instrumente ist, daß dieser Code in der Regel von Spezialisten in diesen Teilbereichen entwickelt wurde und in den allermeisten Fällen qualitativ hochwertige und zuverlässige Anwendungen zu erwarten sind. Das LabVIEW selbst bietet eine Vielzahl von Virtuellen Instrumenten und Beispielapplikationen, die ideale Templates für eigene Entwicklungen darstellen können. Muß aus Zeitgründen der fremdentwickelte Bedarf an VIs größer sein, so bieten sich diverse Toolkits, die eine Vielzahl von sofort einsetzbaren Virtuellen Instrumenten zur Verfügung stellen.

Für nahezu alle Einsatzbereiche sind Toolkits auf dem Markt verfügbar.

Durch dieses Angebot ist ein Entwicklerteam in der Lage auch sehr komplexe Projekte in sehr kurzer Zeit auf einem sehr hohen Qualitätsniveau zu realisieren. Auch Spezialisten, die bisher relativ wenig mit der Programmentwicklung betraut waren sind relativ schnell in synergetisch agierende Entwicklerteams integrierbar. Mit LabVIEW ist ein fraktaler Entwicklungsansatz optimal realisierbar, das heißt, daß jeder einzelnen seine persönlichen Stärken voll zur Geltung bringen kann.

Aufgrund der gravierenden Vorteile gegenüber herkömmlichen textbasierten Programmierumgebungen reduzieren sich die Programmentwicklungszeiten bei kleinen bis mittleren Projekten um Faktor fünf bis zehn.

Einsatzbereiche

Aufgrund seiner positiven Eigenschaften setzt sich LabVIEW in immer mehr Bereichen durch. Längst ist es zetnraler Bestandteil bei der technischen Umsetzung von Stunts. Die Polizei in Los Angeles baut auf LabVIEW basierte Systeme um Schüsse innerhalb gefährdeter Gegenden des Stadtgebietes zu lokalisieren. Die Physiker des CERN vertrauen bei ihren Experimenten auf LabVIEW. Die NASA benutzte LabVIEW um den Sojeurner auf dem Marsboden zu überwachen. Die Software ist mittlerweile aus keinem Industriebereich wegzudenken. Vor allem die Automotive- und die Kommunikationsindustrie setzen voll auf G.

Nachteile

LabVIEW bietet gegenüber textbasierten Ansätzen eine Vielzahl von strategischen Vorteilen wie die bessere Eignung zu RAD (Rapid Application Development) und RAD (Rapid Application Prototyping). LabVIEW ist ausserdem wesentlich leichter erlern- und wartbar.

Was LabVIEW bis jetzt nahezu völlig gefehlt hat, sieht man vom Ende der 90er Jahre rudimentär implementierten GOOP-Ansatz (Graphical Object Oriented Programming) ab, ist Grafische Objektorientierung mit Mehrfachvererbung und ein Ereignisflussmodell.

Traditionelle Programmentwicklung in LabVIEW beruht auf dem hierarchisch und modular aufgebauten Datenflussmodell.

Die Entwicklung von LabVIEW-Programmen ist jedoch leider oft getragen von anachronistischen Design-Methodologien wie dem Top-Down Designansatz. Häufig ist die grafische Benutzeroberfläche der zentrale Bestandteil des Toplevel-VIs und die SubVIs bilden in der Regel keine strategischen Bestandteile innerhalb der Problemdomäne. Die sogenannte Applikationsschicht (applications layer) ist in herkömmlichen Ansätzen kaum berücksichtigt.

Bis zu einem gewissen Grad sind traditionelle LabVIEW-Programme zwar noch gut skalier- und wartbar. Übersteigen derartige – monolithische - VIs eine gewisse Grössenordnung, sind sehr grosse Anstrengungen zu unternehmen, will man massive Performanceeinbrüche vermeiden. Viele monolithische Programme bestehen aus parallelen State-Machines mit einer Vielzahl von Schleifen und Schieberegistern, die alle parallel abzuarbeiten sind und Ressourcen beanspruchen. Um ein optimales Ressourcenmanagement bei nicht objektorientierten Ansätzen zu bewerkstelligen, ist der breitbandige Einsatz von Softwarekomponenten wie Occurencies, Queues, Notifications, Call by Reference Nodes, Semaphoren und Rendesvousz unumgänglich. Da viele LabVIEW-Programmierer jedoch nur relativ geringe Softwareengineering-Kenntnisse besitzen stellt dies eine bedeutende Hürde bei der Lösung komplexerer Probleme dar.

Monolithische Applikationen haben ausserdem das grosse Problem, dass eine Veränderung des Programmcodes bei Main- oder bei SubVIs leicht zu negativen Seiteneffekten innerhalb der Applikationen führen kann. Der Grund hierfür ist unter anderem darin zu suchen, dass viele Status- und Parametrierinformationen als Globale Variablen implementiert sind. Eine vollständige Applikationskapselung ist damit unmöglich.

Die Wartung sehr grosser monolithischer Anwendungen ist auf Grund der oben beschriebenen Eigenschaften sehr zeit- und damit natürlich auch kostenintensiv.

Da in der Industrie die Mess- und Automatisierungslösungen jedoch sehr lange im Einsatz sind (teilweise über 10 Jahre) ist eine kosteneffiziente Wartung der Applikationen notwendig.

Einen Ausweg aus dieser Krise zeigen die Objektorientierten Toolsets für LabVIEW (OTT) der Firma Vogel Automatisierungstechnik. Die Toolsets sind unter dem Technologie Oberbegriff ObjectVIEW zusammengefasst.

Anwendungsstruktur und Schichtenmodell

Applikationen sind üblicherweise in Schichtenstrukturen realisiert. Dabei hat sich eine 3-Schichtstruktur durchgesetzt.

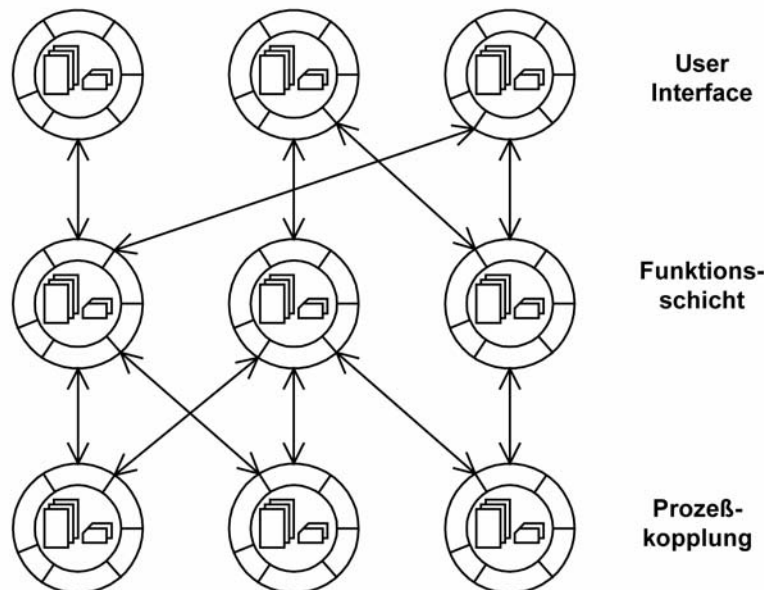

Bild 1: Applikationsstruktur

Die oberste Schicht nennt man User Interface Schicht. In dieser Schicht befinden sich alle Prozesse, die die direkte Interaktion mit dem Bediener (Dateneingabe und Visualisierung) ermöglichen. Die mittlere Schicht (auch Funktionsschicht genannt) stellt das Bindeglied zwischen User Interface Schicht und Prozess Kopplungs Schicht dar. In dieser Schicht laufen alle Verarbeitungsvorgänge der Applikationen ab.

Die Funktionsschicht ist – sinnvollerweise - hierarchisch zu strukturieren. Die einzelnen Objekte innerhalb dieser Schicht repräsentieren logische Funktionseinheiten, die Eingaben-, Verarbeitungs- und Ausgabeaufgaben übernehmen. Für die Abbildung der Struktur eignen sich in besonderer Weise aktive

Objekte, die untereinander ereignisgesteuert kommunizieren. Mit Hilfe von Objektnetzen sind die Kommunikationsbeziehungen und die hierarchische Struktur auf anschauliche Weise darstellbar. Die Erstellung von Objektnetzen mit dem Object Net Toolset (ONT) innerhalb eines LabVIEW-Diagrammes vereinigt Design, Implementierung und Dokumentation in einem Arbeitsgang. Auf Grund der Kapselung und hierarchischen Strukturierung der Funktionseinheiten (aktive Objekte mit aktiven Subobjekten) entstehen sehr gut wartbare-, wiederverwendbare-, leicht modifizier- und skalierbare Applikationen.

In der unteren Schicht (Prozesskopplung) ist die Verbindung zwischen funktionaler Ebene und I/O Komponenten realisiert. In dieser Schicht befinden sich unter anderem die Treiber für interne und externe Hardwarekomponenten (PCI, PXI, Profibus, CAN, RS232, GPIB, u.v.a.m.).

Bei zeitunkritischen Applikationen sollten hier Standardkommunikationsmechanismen wie OPC (OLE for Process Control), eine Variante des COM+ (Component Object Model) Implementierung Microsofts zum Einsatz gelangen.

Mit Hilfe des OPC Interfaces sind nahezu alle I/O-Komponenten (die über unterschiedlichste Bussysteme verfügen können) einheitlich ansprechbar.

Der Trend hin zu verteilten intelligenten Systemen in der Automatisierungstechnik führt zwangsläufig zum Einsatz objektorientierter Kommunikationsprotokolle.

Zur Zeit entstehen zwei Ansätze, die sich mit der Kommunikation in verteilten Applikationen auf Objektebene befassen. ProfiNet ist ein Ansatz der Profibus Nutzer Organisation (PNO), der Profibuskommunikation über das Internetprotokoll IP (üblicherweise auf Ethernet Hardware) abbildet. Das IAONA Konsortium (Industrial Automation Open Networking Alliance) favorisiert das iDA-Modell (Interface for Distributed Automation), die eine direkte Anbindung von intelligenten und verteilten I/O-Elementen über Ethernet auf IP-Basis bietet.

Beide Ansätze basieren auf gleichen Grundprinzipien wie Objektnetze. Verteilte Applikationen mit G-Objektnetzen und iDA- und ProfiNET-Komponenten bilden eine ideale Plattform für völlig offene Automatisierungssysteme.

Objektorientierung und Ereignissteuerung

Grundprinzipien und Definitionen

Das theoretische Fundament der Objekttechnologie bilden drei Grundideen. Die wichtigste davon ist die Kapselung. Mit Hilfe der Kapselung ist es (im Unterschied zu herkömmlichen Funktions- und Datenmodellen) möglich, alle wichtigen Verarbeitungsprozeduren und die zugehörigen Daten zu einer Einheit zusammen zu fassen. Zugriffe auf diese Einheit erfolgen ausschliesslich über exakt definierte Operationen (in der Objektterminologie als Methoden bekannt). Ein direkter externer Zugriff auf Daten und Prozeduren ist nicht möglich. Methoden sind nur durch definierte Operationen aufrufbar (andere Operationen können nicht zugreifen). Die gekapselten Einheiten nennt man Objekte.

Objekte sind definitionsgemäss selbständige, abgeschlossene Softwareelemente mit Eigenschaften, die nachfolgend beschrieben sind.

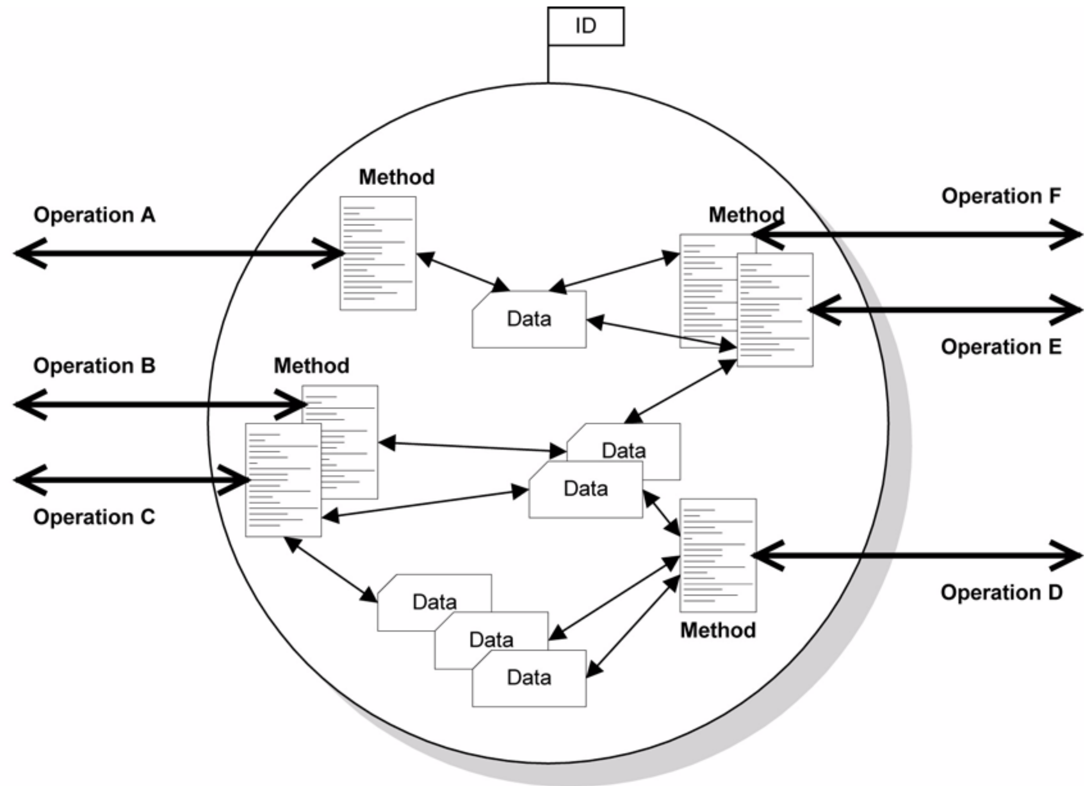

Bild 2: Kapselung von Objekten

Die zweite Grundidee der Objekttechnologie ist die Abstraktion. Unter Abstraktion ist die Reduktion eines komplexen Gebildes auf seine wesentlichen Eigenschaften und Verhaltensweisen zu verstehen. Die Zusammenfassung von Objekten mit gleichen Grundeigenschaften nennt man Klassenbildung.

Klassen sind hierarchisch strukturierbar. Subklassen (Unterklassen) besitzen alle Eigenschaften hierarchisch höher stehender Klassen (Eltern- oder Superklassen). Die Bildung von Subklassen aus Superklassen bezeichnet man als Spezialisierung oder (häufiger) als Vererbung.

Den Umkehrvorgang bezeichnet man als Generalisierung. Es entstehen Klassenhierarchien, die baumähnliche Strukturen aufweisen.

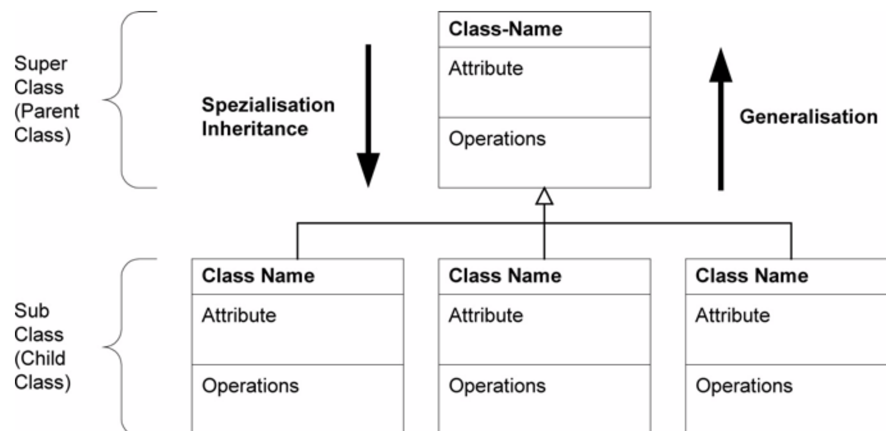

Bild 3: Klassenhierarchie

Die dritte Grundidee der Objekttechnologie ist der Polymorphismus (übersetzt: Vielgestaltigkeit). Objekte verschiedener Klassen können auf gleiche Operationen (Methoden) unterschiedlich reagieren.

Objekte sind Instanzen einer Klasse (Objekte entstehen durch Instanzierungen einer Klasse). Klassen können Elemente von Klassenhierarchien sein. Klassenhierarchien entstehen durch Vererbung der Klasseneigenschaften (Attributen und Methoden) und deren Spezialisierung. Alle Objekte besitzen eine eindeutige Identifikation (Name und Referenz).

Das Verhalten des Objektes ist durch sein Interface (Attribute, Methoden und Ereignisse) bestimmt. Jedes Objekt befindet sich in einem definierten Zustand, bestimmt durch vorangegangene Interaktionen mit seiner Umwelt.

Jedes erzeugte Objekt befindet sich nach der Instanzierung in einem definierten Initialzustand.

Grundsätzlich unterscheidet man zwischen passiven und aktiven Klassen. Instanzen passiver Klassen (Objekte ohne Prozesscharakter) kapseln Daten und Methoden und gewährleisten in einer Multitaskingumgebung einen wechselseitigen Ausschluss der Methodenzugriffe. Aktive Objekte (Instanzen aktiver Klassen) stellen zusätzlich eine oder mehrere Prozesse zur Verfügung. Aktive Objekte sind in der Lage synchron oder asynchron mit der Umwelt zu interagieren. Die Synchronisation der Prozesse aktiver Objekte ist mit Hilfe von Zustandsautomaten (State Machines) oder Petrinetzen beschreibbar. Die Gesamtheit aller Zustandsübergänge von der Erzeugung- bis zur Beendigung eines Prozesses nennt man Prozesslebenszyklus.

Zustandsautomaten

Endliche Automaten oder Zustandsautomaten (finite state machine; sequential machines) besitzen eine endliche Anzahl von Zuständen (interne Konfigurationen).

Zustandswechsel definieren den Übergang zwischen zwei Zuständen. Zustandswechsel erfolgen auf Grund von Ereignissen, Methodenaufrufen oder ablaufzeitgesteuert. Innerhalb einzelner Zustände sind Aktionen und Aktivitäten auslösbar. Es sind insgesamt drei Aktionen (actions) zu unterscheiden. Unter der „Entry-Action“ versteht man diejenige Aktion, die beim Eintritt in den Zustand ausgeführt wird.

Die Zustandsaktivität ist die fortlaufende Ausführung der „DO-Action“. Beim Verlassen des Zustandes ist eine „Exit-Action“ ausführbar.

Zustandsautomaten sind hierarchisch strukturierbar. Man unterscheidet zwischen verschiedenen Typen. Die am weitesten entwickelte Type stellte Prof. Dr. David Harel 1987 vor. Die nach ihm benannten hybriden Zustandsmaschinen sind zentraler Bestandteil der UML Notation (Unified Modelling Language).

Petrinetze

Ein Petri-Netz besteht aus drei Netzelementen. Das sind einmal die Plätze (places) und des weiteren die Übergänge zwischen ihnen. Diese Übergänge bezeichnet man als Transitionen (transitions). In einem Petri-Netz sind Plätze und Transitionen durch Kanten (edges) so verkettet, daß sie sich jeweils abwechseln. Zwischen zwei Plätzen befindet sich also immer ein Übergang (Transition). Die Verkettung der Netzelemente erfolgt durch sogenannte gerichtete Kanten. Durch das Petri-Netz bewegt sich eine oder auch mehrere Marken. Ein Platz ist dann aktiv, wenn er mindestens eine Marke besitzt. Die Transitionen bestimmen, ob und wann ein Platz die Marke (token) an seinen Nachfolger abgeben muß. Dafür besitzt jede Transition eine sogenannte Weiterschaltbedingung. Ist diese Bedingung erfüllt, dann bewirkt die Transition die Weitergabe der Marke zwischen den zwei Plätzen, zwischen denen sie sich befindet. Die Weitergabe erfolgt immer gerichtet, entsprechend den Kanten, mit denen Plätze und Transitionen miteinander verkettet sind. Eine Marke kann sich während ihres Flusses durch das Petri-Netz in mehrere Marken aufteilen, die sich später wieder zu einer Marke vereinigen können.

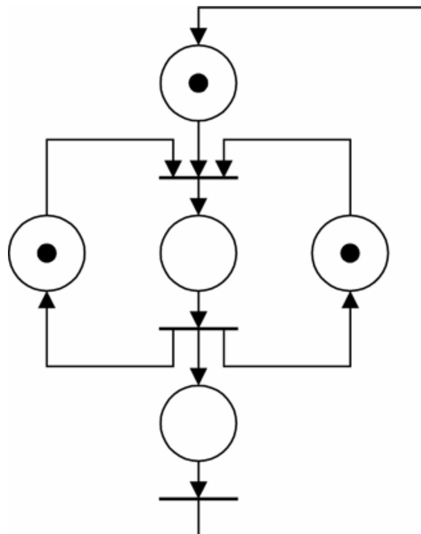

Bild 4: *Bild 2-x4 : Petrinetz mit boolschen Marken (Bedingungs/Ereignis-Netz; Condition/Event Net)*

Im Gegensatz zu einer Zustandsmaschine kann ein Petri-Netz mehrere aktive Zustände haben. In einem Petri-Netz besteht die Möglichkeit des Parallellaufs von Zuständen sowie der Einrichtung von Unterzuständen. Ein Petri-Netze ist darüber hinaus universeller als ein Programmablaufplan, weil es sich auf einer höheren Abstraktionsebene befindet. Auch ein Programmablaufplan kann nur eine Marke besitzen.

Mit dem Petri-Netz-Konzept ist bei hoher Übersichtlichkeit eine schrittweise Annäherung an die Problemlösung möglich. Damit bietet es sich für den Entwurf von Lösungen mit anfangs nicht genau bestimmbarer Struktur an, wie z.B. komplexe Automatisierungslösungen oder Workflow-Applikationen.

Man unterscheidet drei Petrinetztypen. In Bedingungs/Ereignis-Netzen (Condition/Event Net) kann jede Stelle (place) eine oder keine Marke enthalten. Eine Transition kann schalten, wenn jede Eingabestelle eine Marke enthält und jede Ausgabestelle leer ist.

In Stellen/Transitions-Netzen (Place/Transition Net) können die Stellen mehr als eine Marke enthalten. Die Transitionen (transitions) müssen so viele Marken (tokens) beim Schalten wegnehmen oder hinzufügen, wie die Gewichte an den Pfeilen angeben. Jede Stelle besitzt eine maximale Kapazität. Sie definiert die Anzahl der maximal zulässigen Marken.

In Prädikat/Transitions-Netzen (Predicate/Transition Nets) kommen individuelle (gefärbte) Marken zum Einsatz. Man spricht aus diesem Grunde auch von kolorierten Petrinetzen. Die Marken sind „farbabhängig“ in der Lage, beliebige Datentypen aufzunehmen.

Eine Transition kann nur für die Marken schalten, für die die Schaltbedingung erfüllt ist.

Objektnetze

Ein aktives Objekt, welches wiederum aktive Objekte enthält, bezeichnet man als Objektnetz. Dieses Konzept ist auch unter den Begriffen UML-RT (Unified Modelling Language for Real Time) oder ROOM (Realtime Object Oriented Modeling Language) bekannt. Mit Hilfe von Objektnetzen sind die Kommunikationsbeziehungen und die hierarchische Struktur einer Applikation auf anschauliche Weise darstellbar. Die Kommunikation zwischen diesen Objekten geschieht ereignisgesteuert über Ports. Jedes Objekt stellt eine gekapselte Einheit dar. Die Ports bilden die Verbindungstechnologie zwischen aktivem Objekt und Umwelt. Dabei verbirgt der Port die physikalische Adresse des externen Kommunikationspartners vor dem Objekt. Umgekehrt kapselt der Port die interne Verarbeitung der Signale. Das nachfolgende Bild zeigt ein aktives Objekt, das ein Objekt und einen Prozess kapselt. Dieses Objekt kommuniziert ereignisgesteuert und indirekt adressiert über Ports mit seiner Umwelt.

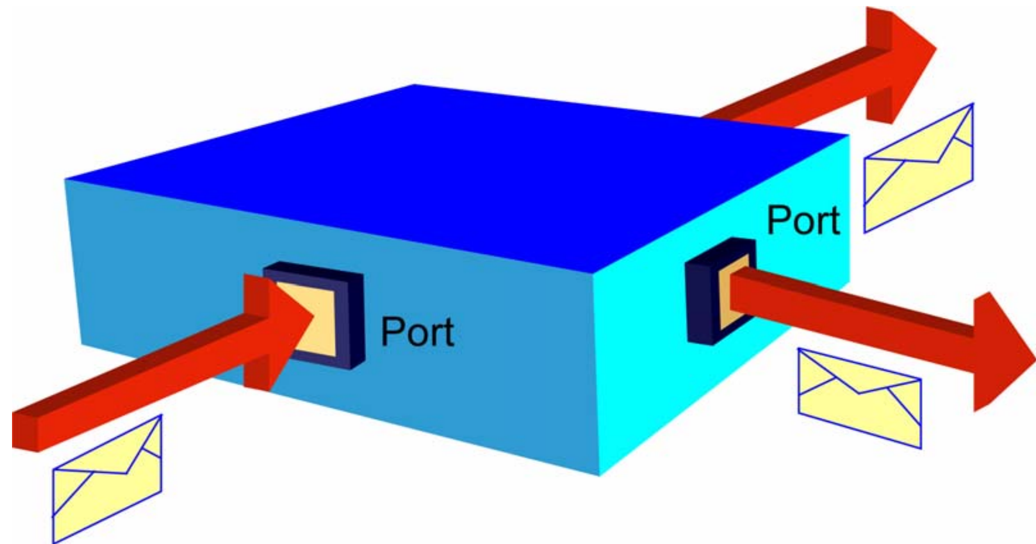

Bild 5: Bild 2-x5: Aktives Objekt (Active Objects)

Ein Port ist mit einem SubVI-Terminal LabVIEWs vergleichbar, wobei über Ports nur Adressen von ereignisgesteuerten Kommunikationsbeziehungen übermittelt werden. Objektnetze sind beliebig tief hierarchisch strukturierbar.

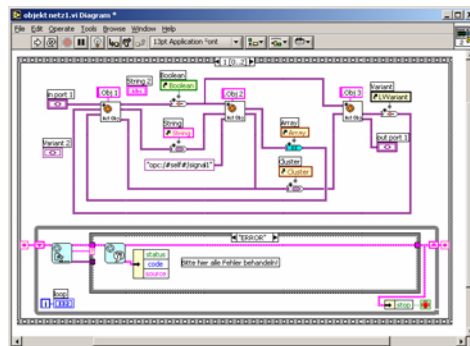

Bild 6: Objekt-Netz

Man kann Ports in verschiedene Grundtypen aufteilen. Synchrone- und asynchrone- Sende- und Empfangsports, Daten- sowie Steuerports. Die Verbindung zwischen Sende- und Empfangsports erfolgt mit Hilfe eines Kommunikationsobjektes, auch Connector genannt.

Die Realisierung von Objektnetzen mit Hilfe des Object Net Toolsets (ONT) vereinigt Design, Implementierung und Dokumentation von Software in einem Arbeitsgang. Basis für sehr gut wartbare-, wiederverwendbare-, leicht modifizier- und skalierbare Applikationen ist die Kapselung und hierarchische Strukturierung der Funktionseinheiten durch aktive Objekte mit aktiven Subobjekten innerhalb der Objektnetze.

Verteilte Anwendungen

LabVIEW besitzt eine Reihe von Mechanismen, um verteilte Applikationen zu realisieren. Die drei wichtigsten sind TCP/IP (Transmission Control Protocol/Internet Protocol), VI Server und DataSocket.

DataSocket ist ein Programmiertool, das eine API (Application Programming Interface) zur Verfügung stellt, die einen Datenaustausch zwischen verteilten Applikationen ermöglicht.

Die API kapselt Low Level Funktionen wie File I/O, TCP/IP und FTP/http Kommunikationsrequests. Mit Hilfe DataSockets ist es möglich, auf einer höheren Ebene mit Hilfe einer einheitlichen Schnittstelle verschiedene Datenstrukturen, sowie Kontroll- und Kommunikationsmechanismen zu verwenden, ohne Kenntnis der jeweiligen Mechanismen und Strukturen. DataSocket erkennt automatisch unterschiedliche Protokolle und Datenformate. Aufgrund des Overheads, der durch die automatische Anpassung an die verschiedenen Protokolle entsteht ist eine DataSocket-Verbindung nicht so performant wie eine VI-Server Kommunikation.

DataSocket stellt Mechanismen für das Publizieren (Publishing) und Abonnieren (Subscribing) von Daten zur Verfügung. Die Veranlassung der Übermittlung von Daten geschieht ereignisgesteuert. Diese Ereignisse können Aktionen in Objekten auslösen. DataSocket ist gut geeignet, wenn Informationen eines Servers vielen Clients verfügbar gemacht werden sollen. Es sind Daten beliebigen Typs (Variant) transferrierbar. Objekte können gleichzeitig Publisher und Subscriber enthalten, wobei die Kommunikation ereignisgesteuert und asynchron erfolgt.

Der VI-Server Mechanismus ermöglicht Remote Procedure Calls (RPC) innerhalb von LabVIEW-Umgebungen und basiert auf TCP/IP. Der Kommunikationsmechanismus ist Grundlage für Methodenaufrufe auf entfernte Objekte. VI Server in Verbindung mit Objekt Technologie Toolset Komponenten (Objekte, Petri- und Objektnetze) bilden eine ausgezeichnete Plattform für die Entwicklung hochperformanter intelligenter verteilter Applikationen. Der Vorteil bei der Verwendung einer VI-Server-Kommunikation liegt (neben der höheren Kommunikationsgeschwindigkeit) in der Multiplattformfähigkeit. Alle Betriebssysteme, die LabVIEW unterstützt, bieten VI-Server-Kommunikationsmöglichkeiten. Somit sind verteilte und komplexe OTT-Applikationen in heterogenen LAN- und WAN-Environments realisierbar (anders als in proprietären COM+ Umgebungen).

Natürlich unterstützten OTT-Komponenten auch plattformspezifische Kommunikationsmechanismen.

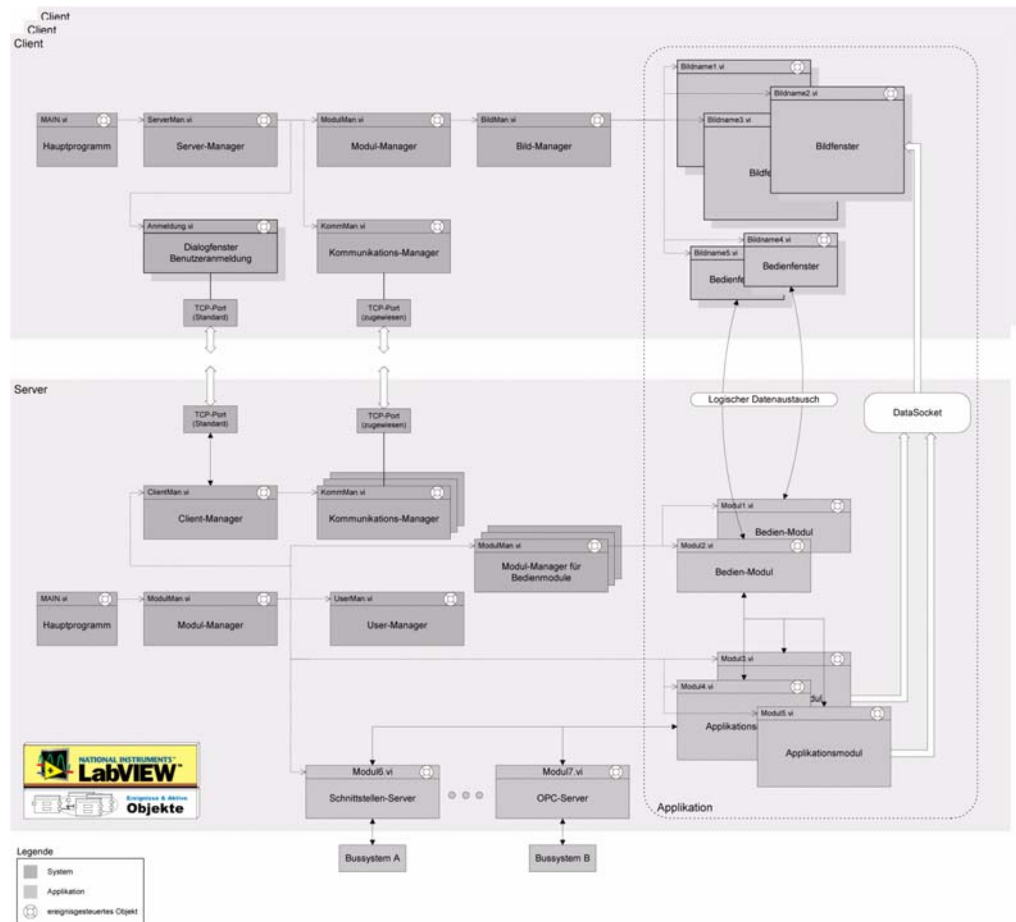

Bild 7: Application Framework Toolset

Für das optimale Kommunikationsmanagement stehen im Application Framework Toolset (AFT) Kommunikationsklassen zur Verfügung. Diese Komponenten verwalten und kapseln die Kommunikation innerhalb verteilter Applikationen. Die Verbindung zwischen einzelnen Knoten (Teilnehmern) wird zyklisch überwacht. Der Anwender nutzt diese Funktionalität durch einfache Angabe des gewünschten Hostnamens in einer Adresse (URL – Uniform Ressource Locator).

Jeder Knoten stellt Informationen über verfügbare Dienste auf Wunsch (request) zur Verfügung (Browsingfunktion). Dazu sind alle Namen öffentlicher Objekte und deren Klassenbeschreibungen (class descriptions) bei gegebener Zugriffsberechtigung innerhalb der Applikationen kommunizierbar.

Übersicht Object Technology Toolsets

OT-Toolsets repräsentieren eine Synthese unterschiedlicher Designkonzepte angefangen vom LabVIEW-immanenten Datenflussansatz über Objektorientierte Ansätze mit Klassen und Mehrfachvererbung , Ereignis-, Signal- und Botschaftenmanagement, verteilten – inherent parallelen- intelligenten Objekten, Aggregation, Ports, Petrinetzen und Objektnetzen.

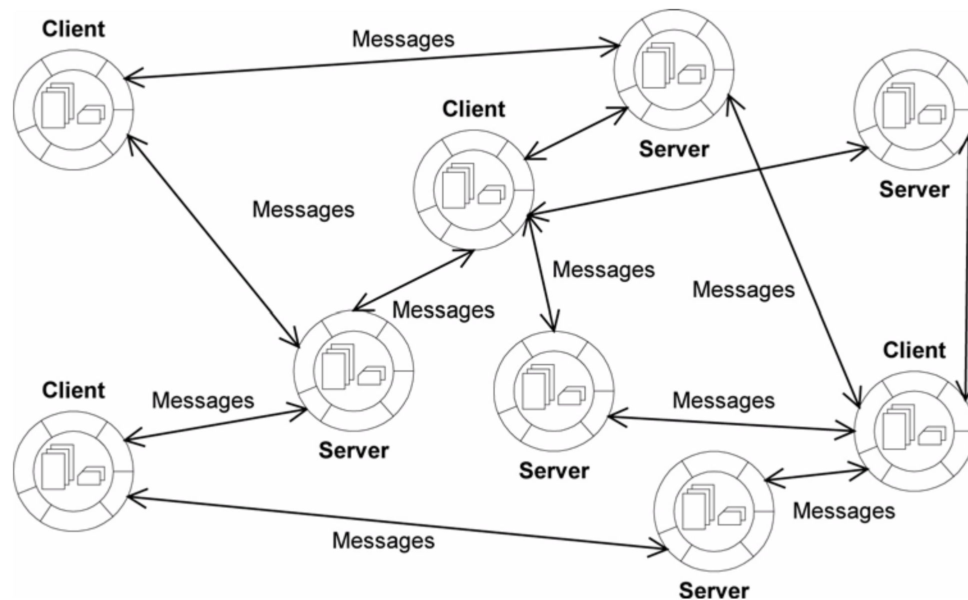

Bild 8: Objektmodell (Client- und Server-Objekte)

Die Verwendung dieser Technologien führt zu verteilten-, leicht wiederverwendbaren-, extrem skalier- und wartbaren-, redundanten-, robusten und sehr zuverlässigen intelligenten Objekten, die eine direkte Problembeschreibung und –abstrahierung ermöglichen. Die Objekte sind im LAN und eingeschränkt auch in WAN-Umgebungen (inklusive Internet) verteil- und administrierbar.

Die Basisversion (Object Event Toolset – OET) unterstützt Aktive Objekte und Eventmanagement basierend auf asynchronen Botschaften (mit Hilfe von Queues), Signalisierungen (über Notifikationen) und weit verbreitete Kommunikationstechnologien wie ActiveX und OPC/DataSocket.

Die Professional Version (Class Inheritance Toolset, CIT) stützt sich auf OET Features und bietet darüber hinaus Passive Objekte, Klassenbibliotheken und Klassenmanagement, Mehrfachvererbung, Methoden, Methodenaufrufe sowie Flache Zustandsautomaten. Das Developer Toolset (Object Net Toolset, ONT) ist ein Superset des Professional Toolset. Harel State Machines, Petrinetze (Bedingungs/Ereignis-Netze, Stellen/Transitons-Netze, Prädikat/Transitions-Netze, Hierarchische Netze, Objektnetze und Ports).

Das Universal Toolset (Application Framework Toolset, AFT) nutzt alle Features der untergeordneten Toolsets und bietet weitere Klassenbibliotheken zur Erstellung von SCADA (Supervisory Control and Data Acquisition) und MES Applikationen (Manufacturing Execution Systems) wie Client/Server, User-Management, Node Communication/node Management, Alarm/Meldungs-Management, Trends, Error Management, sowie eine Beispielapplikation aus dem Bereich Facility Management zur Demonstration
des Zusammenspieles aller Module.

Es ist sehr leicht möglich neue Objekte aus Elternklassen abzuleiten, wobei die Klassenhierarchietiefe nur durch den Hauptspeicherausbau und das Laufzeitverhalten bei der Ausführung der Objekte begrenzt ist. Die Abstrahierungsmöglichkeiten der Objekt Technologie Toolsets gehen weit über die immanenten Möglichkeiten LabVIEWs hinaus.

Eine herausragende Möglichkeit nebenläufige, nicht deterministische Vorgänge zu abstrahieren besteht in der Verwendung sogenannter Petrinetz. Carl Adam Petri entwickelte bereits in den sechziger Jahren diesen genialen Ansatz der Beschreibung paralleler Abläufe und Prozesse.

Fazit

Grafische Datenflussprogrammierung unterscheidet sich fundamental von herkömmlichen textorientierten Entwicklungstechniken. Der Idealfall einer grafischen Datenflussprogrammierumgebung ist die Zusammenfassung von Problembeschreibung, Problemlösung und aussagekräftiger, selbsterklärender Dokumentation in einem Arbeitsgang. Das Werkzeug sollte ein, mit herkömmlichen Programmiersprachen vergleichbares Laufzeitverhalten (bezogen auf die Ausführungsgeschwindigkeit; Compiler) haben, eine deklarative- und übersichtliche-, gut strukturierte Darstellung gewährleisten und vor allem Menschen mit unterschiedlichen informationstechnischen Vorkenntnissen einen natürlichen Zugang gewähren.

Objekt- und Eventmanagementtechnologien ermöglichen LabVIEW den Sprung in die Liga der Universalprogrammiersprachen ohne Einschränkungen.

Die Objekttechnologietoolsets für LabVIEW (OTT) der Firma Vogel (www.lvot.de; www.gooptech.com) erweitern LabVIEW um Ojektorientierung mit Mehrfachvererbung, Eventmanagement und unterschiedliche zusätzliche Abstrahierungsmechanismen wie Objekt- und Petrinetze. Alle wichtigen Kommunikationsmechanismen und -protokolle (TCP/IP, ActiveX, OPC, Notifications, Queues, XML, DataSocket) sind unterstützt. Die Toolsets sind ideal zur Entwicklung verteilter intelligenter Systeme (unter anderem gemäss IEC 61499).

Verschiedene Designmethoden sind gleichzeitig verwendbar. Die Formulierung des Problems ist nahezu identisch mit der Codierung und Dokumentation. Anders als bei anderen Entwicklungssystemen sind die Designansätze durchgängig (bei herkömmlichen UML-Ansätzen wird in der Regel nur ein Template erzeugt, das dann als Basis für die weitere Codierung benutzbar ist). Durch diese zusätzlichen Möglichkeiten gibt es keine Einschränkungen mehr in Bezug auf Performance und Skalierbarkeit. Die Erstellung von Klassenbibliotheken mit intelligenten verteilten Objekten ist eine Domäne der Toolsets. OTT-Technologie wird in vier unterschiedlichen Varianten angeboten (OET Object Event Toolset, OIT Object Inheritance Toolset, ONT Object Network Tooset, AFT Application Framework Toolset), wobei die ersten beiden bereits verfügbar sind. LabVIEW-Entwickler sind jetzt in der Lage in bisher völlig neue Terrains der Informationstechnologie vorzustossen. Neben der Automatisierungs- und Messtechnik (mit zehntausenden von I/O-Punkten) werden völlig neue Applikationen in den unterschiedlichsten Bereichen denkbar. Die Kombination von Grafischer Datenflussprogrammierung, Objektorientierung mit Mehrfachvererbung und der zusätzlichen Verwendung von Petrinetzen führt zu einer bisher nicht dagewesenen Freiheit in der Realisierung auch extrem komplexer Systeme. Natürlich profitieren auch kleine Projekte von den Möglichkeiten der Objektorientierung in LabVIEW. Endlich ist es möglich, langfristig wartbare und hochperformante Software (völlig ereignisorientiert) in noch kürzerer Zeit und damit zu noch geringeren Kosten dem Kunden an die Hand zu geben. Auch Entwickler, die bisher LabVIEW aufgrund eingeschränkter Objektorientierung gemieden haben, können jetzt auf auf eine universell einsetzbare Entwicklungsplattform setzen, die ein Optimum an Rapid Prototyping- und Rapid Application Development-Optionen bietet und dazu noch echt multiplattformfähig- und auf verschiedenen Plattformen echtzeitfähig ist.

Referenzen

[1] www.gooptech.com

[2] www.objectview.de

[3] www.vat.de

[4] www.systec-gmbh.com

[5] www.ni.com

[6] Herbert Pichlik, Jens Vogel, Object Technology Toolsets for LabVIEW - OTT, NI-Week 2001 Proceedings, Austin TX 8/2001

[7] Jens Vogel, Automating Shipping in a Cement Works Using LabVIEW, NI-Week 2001 Proceedings, Austin TX 8/01

[8] Herbert Pichlik, Universal Test- and Automation System using an event driven object oriented _LabVIEW approach (UTAS), NI-Week 2001 Proceedings, Austin TX 8/01

[9] OTT User Manual, Vogel/SYSTEC, Dornburg/Nürnberg, 6/01

[10] Rahman Jamal, Herbert Pichlik, „LabVIEW Applications and Solutions“, Prentice Hall 8/98

[11] Herbert Pichlik, „Universal testing of household equipment using LabVIEW“, Best applications of virtual instrumentation, NI-Week 96 Proceedings, Austin TX 8/96

[12] Herbert Pichlik, „Networkcentric test and measurement system“, Best applications of virtual instrumentation, NI-Week 97 Proceedings, Austin TX 8/97

[13] Herbert Pichlik: Bilder-Code, Grafische Programmierumgebung National Instruments LabView 6.i Prüfstand, Grafische Programmierung, Softwareentwicklung, Programmierung, LabView, G, VI, Virtual Instruments c't 3/01, Seite 88

[14] Martin Klein, Herbert Pichlik: Software in Bildern, Grafisches Programmieren mit LabView Report, Grafische Programmierung, visuelle/grafische Programmierung, G, Virtuelle Instrumente, Gerätesteuerung, Messen/Steuern/Regeln, PC-Meßtechnik, IEEE-/IEC-Bus, LabView 5, National Instruments c't 5/98, Seite 230