GSI

Dipl. 2007 - 07
August

Entwicklung von Objekt- und Petri-Netzen im Rahmen eines Kontrollsystems

A. Schwinn

(Diplomarbeit Hochschule Darmstadt)

Gesellschaft für Schwerionenforschung mbH
Planckstraße 1 · D-64291 Darmstadt · Germany
Postfach 11 05 52 · D-64220 Darmstadt · Germany

Diplomarbeit zum Thema

Entwicklung von Objekt- und Petri-Netzen im Rahmen eines Kontrollsystems

Referent: Prof. Dr. Karl Erich Wolff
Koreferent: Prof. Dr. Jürgen Groß
Betreuer: Dr. Holger Brand
Erstellt von: Alexander Schwinn
Vor der Lache 1
64521 Groß-Gerau
Matrikelnummer 698810
E-Mail: alexander.schwinn@gmx.de

Inhaltsverzeichnis

Vorwort

Das *CS*-Framework ist eine Software, die, basierend auf LabVIEW, eine objektorientierte, ereignisgesteuerte Entwicklungsumgebung für Kontrollsysteme der Experimente an der GSI und anderen Forschungsinstituten zur Verfügung stellt.
Diese Diplomarbeit befasst sich mit der Erweiterung des *CS*-Frameworks um spezielle Objekt- und Petri-Netze. Das Framework wurde von der Kontrollsystemgruppe der Abteilung Experiment Elektronik an der GSI zur Schaffung einer einheitlichen Software-Architektur entwickelt, die die Wiederverwendbarkeit der entworfenen Software verbessern soll. Hierzu beinhaltet das *CS*-Framework Entwurfsvorlagen für die verschiedenen Aufgaben eines Kontrollsystems. Dazu zählen vor allem die Steuerung externer Geräte sowie das Überwachen und Archivieren gerätespezifischer Parameter.
Hierbei spielen auch die Objekt- und Petri-Netze eine Rolle. Sie dienen dem Benutzer einerseits als übersichtliches UML-Diagramm, sind zum anderen aber auch direkt ausführbarer Code.
Der Anwender hat somit die Möglichkeit, auf einen Blick die Funktionsweise eines Systems zu erfassen und diese gleichzeitig an seine Bedürfniss anzupassen.
Objekt-Netze dienen hierbei dem geregelten Starten und Überwachen der Objekte, Petri-Netze zur Modellierung komplexer Prozesssequenzen. Besonders beachtet wird hierbei die Wiederverwendbarkeit der Teilkomponenten und die hohe Modularität der Netze.

1 Einleitung

1.1 Die GSI[1]

Das von Bund und dem Land Hessen getragene Forschungsinstitut betreibt seit 1969 in Darmstadt eine Beschleunigeranlage für Schwerionen, die einerseits der Grundlagenforschung im Bereich der Kern- und Atomphysik dient, andererseits Anwendung auf den Gebieten der Materialforschung und Strahlenmedizin findet. Besondere Aufmerksamkeit erregte die Großforschungseinrichtung mit der Entdeckung von sechs neuen chemischen Elementen[2] und der Entwicklung einer weltweit einzigartigen Tumortherapie, mit deren Hilfe seit 1997 über 350 Patienten[3] erfolgreich behandelt werden konnten.
Fortschritte im Bereich der Wissenschaft können nur erzielt werden, wenn die Experimentiergeräte ebenfalls Gegenstand der Forschungsaktivität sind. Daher spielt sowohl die Weiterentwicklung der Beschleunigeranlage als auch der Experimente selbst eine zentrale Rolle. Hierfür unterhält die GSI den Bereich wissenschaftlich technische Infrastruktur, deren Abteilungen sich mit der Betreuung des Beschleunigers und der Experimente beschäftigen, um optimale Voraussetzungen für die Experimentatoren zu schaffen. Mehr als 1000 Wissenschaftler aus über 30 Ländern reisen pro Jahr an die GSI, um die Beschleunigeranlage für ihre Forschungen zu nutzen.
Diese Arbeit entstand im Rahmen der Kontrollsystemgruppe der Abteilung Experiment Elektronik an der GSI, deren Aufgabe die Beratung und Unterstützung der Experimentatoren bei der Entwicklung experimentspezifischer Kontrollsysteme ist.

1.2 Das *CS*-Framework - ein Überblick

Zum Kontrollieren von Geräten in verteilten Netzwerken und zur Steuerung mit Hilfe parallel laufender Prozesse an der GSI wurde vor geraumer Zeit "ObjectVIEW", eine kommerzielle, objektorientierte Erweiterung für LabVIEW genutzt. LabVIEW als Programmiersprache bietet sich an, da parallele Prozesse damit unkompliziert verwirklicht werden können.

Da ObjectVIEW für den Anwendungs-Rahmen an der GSI nicht ausreichend war, beschloss man ein eigenes, objektorientiertes Kontrollsystem-Framework zu entwickeln, das *CS*-Kontrollsystem.
CS ist ein objektorientiertes, Ereignis gesteuertes, verteiltes Kontrollsystem mit einem sicheren Attributzugriff. Das *CS*-Framework bietet zudem die Möglichkeit, Anwendungen in einem Netzwerk einfach zu verteilen und einen schnellen Objektattribut-Zugriff.

1.3 Themenstellung

Die Zielsetzung dieser Arbeit ist die Implementierung der Objekt- und Petri-Netze in das *CS*-Framework. Die Idee der Objekt- und Petri-Netze wird aus ObjectVIEW aufgegriffen. Die Implementierung jedoch wird komplett neu gestaltet. Zum einen, weil ObjectVIEW ein kommerzielles Tool ist, und man dadurch den Quellcode nicht einsehen kann. Zum anderen, da das CS-Kontrollsystem im Vergleich zu ObjectVIEW einen völlig anderen Aufbau besitzt.
Ergänzend zu den Netzen selbst gehört es weiterhin zu meinen Aufgaben, einige einführende Beispiele zu implementieren, welche dem Benutzer den Einstieg in die Objekt- und Petri-Netze erleichtern.

[1] Gesellschaft für Schwerionenforschung; Mehr Informationen siehe http://www.gsi.de.
[2] Bei den sechs neuen Elementen handelt es sich namentlich um Darmstadtium, Hassium, Meitnerium, Roentgenium, Bohrium, Ununbium mit den Ordnungszahlen 107 bis 112
[3] Stand 2007

2 Entwicklungsumgebung - Das *CS*-Framework

Die folgenden Kapitel (2.1 bis 2.3) wurden zu großen Teilen aus der Diplomarbeit von Maximilian Kugler (siehe [Kugler06]) übernommen. Seine Arbeit beschäftigt sich mit der Erweiterung des *CS*-Frameworks um einen generischen Sequenzer und benötigt daher, genauso wie diese Arbeit, einen kurzen Einstieg ins *CS*-Kontrollsystem. Da das *CS*-Framework seit dieser Arbeit grundlegen verändert wurde (Version 3.10), unterscheiden sich viele Abschnitte, sowie der gesamte Ereignismechanismus von Maximilian Kugler's Arbeit.

2.1 Überblick

2.1.1 Idee

Die Motivation für ein universelles Werkzeug zur Entwicklung von experimentspezifischen Kontrollsystemen entsteht aufgrund der geringen Wiederverwendbarkeit bereits existierender Software. Obwohl die Anforderungen, die die Experimente im Allgemeinen an ein Kontrollsystem stellen, viele Gemeinsamkeiten aufweisen, unterscheiden sich die Umsetzungen der Experimente im Einzelnen stark voneinander. Dies führt dazu, dass beim Entwurf und der Realisierung eines neuen Systems immer komplett von vorne begonnen werden muss, ohne auf bereits getätigte Entwicklungen zurückgreifen zu können und erhöht damit den benötigten Arbeitsaufwand.
Oftmals kommen ähnliche oder sogar gleiche Geräte bei den Experimenten zum Einsatz. Die Wiederverwendbarkeit der Software zur Gerätesteuerung beschränkt sich jedoch meist auf die Treiber. Werden jedoch Regelungsmechanismen benötigt, die die Kommunikation von Geräten untereinander voraussetzten, müssen diese komplett neu implementiert werden. Abhilfe kann an dieser Stelle nur durch eine modulare Programmierung, die einheitliche Schnittstellen bereitstellt, geschaffen werden. Dies ermöglicht nach Bedarf des Experiments einzelne Software-Module beliebig miteinander kombinieren zu können, unabhängig von der Art des Gerätes, dessen Schnittstellen oder Treibersoftware.
Auf diesem Grundgedanken aufbauend, besteht das Konzept des *CS*-Frameworks in der Kombination von generischen Software-Modulen, die vom Framework zur Verfügung gestellt werden, mit experimentspezifischen Erweiterungen. Abbildungen 1 und 2 verdeutlichen diese Idee. Wie beim Lego-Stecksystem kann man einzelne Komponenten des *CS*-Frameworks miteinander verbinden, um daraus ein funktionierendes Kontrollsystem zu konstruieren.

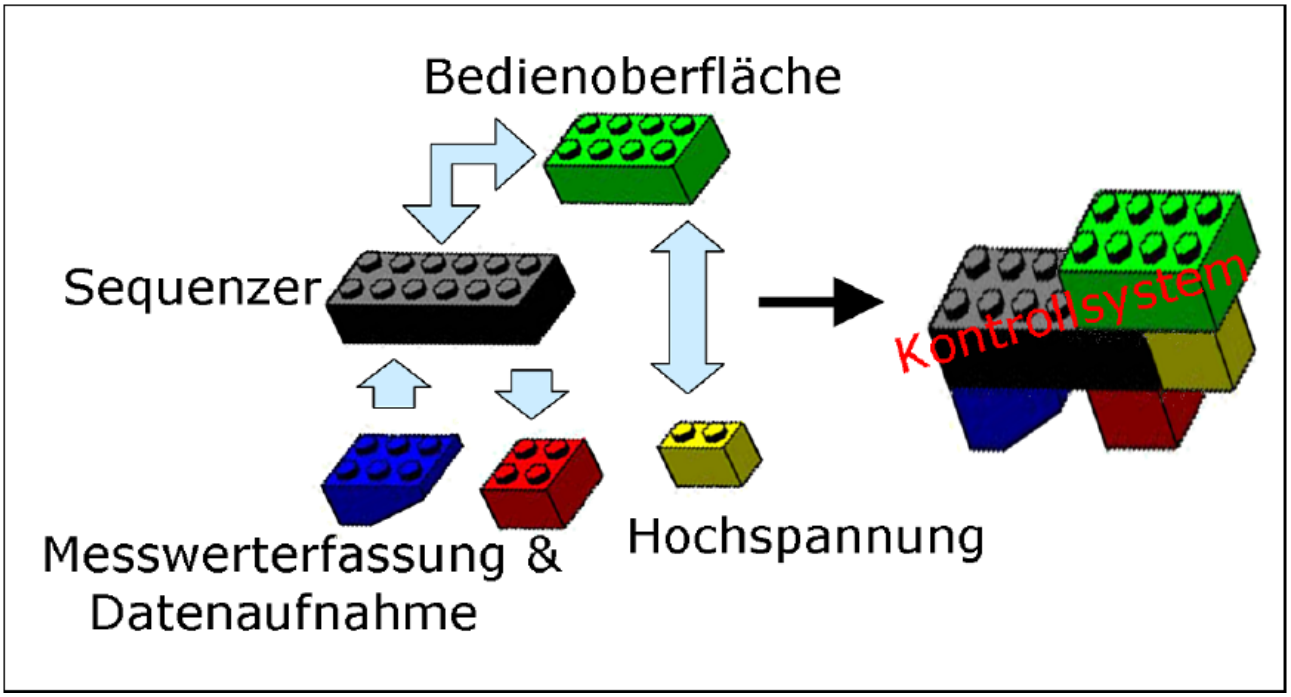

Abbildung 1: *CS-Legobaukastensystem*

Die Wartung, Dokumentation und Weiterentwicklung des Frameworks ist Aufgabe der Kontrollsystemgruppe, die die Experimente bei der Planung und Implementierung ihrer Kontrollsysteme beratend unterstützt. Die Anpassung des Frameworks an ein Kontrollsystem übernehmen die Experimentatoren selbst. Hierfür stellt das

Framework bereits Entwurfsmuster zur Lösung bestimmter Aufgaben, wie beispielsweise das Implementieren gerätespezifischer Software oder grafischer Bedienoberflächen, bereit (siehe auch Abbildung 2).
Diese Zusammenarbeit zwischen Anwendungsentwicklern und Kontrollsystemgruppe gewährleistet, dass Erweiterungen des Frameworks sich an den Anforderungen der Experimente orientieren, ohne deren Wiederverwendbarkeit einzuschränken. Darüber hinaus können Entwicklungen der Experimente direkt in das Framework integriert werden, wie zum Beispiel gerätespezifische Software, die bei anderen Experimenten wieder verwendet werden kann.
Zusammenfassend kann man sagen: Ziel des *CS*-Frameworks ist die Verbesserung der Wiederverwendbarkeit, Wartbarkeit und Dokumentation von Software durch eine einheitliche Software-Architektur und die damit verbundene Ersparnis von Arbeitsaufwand.

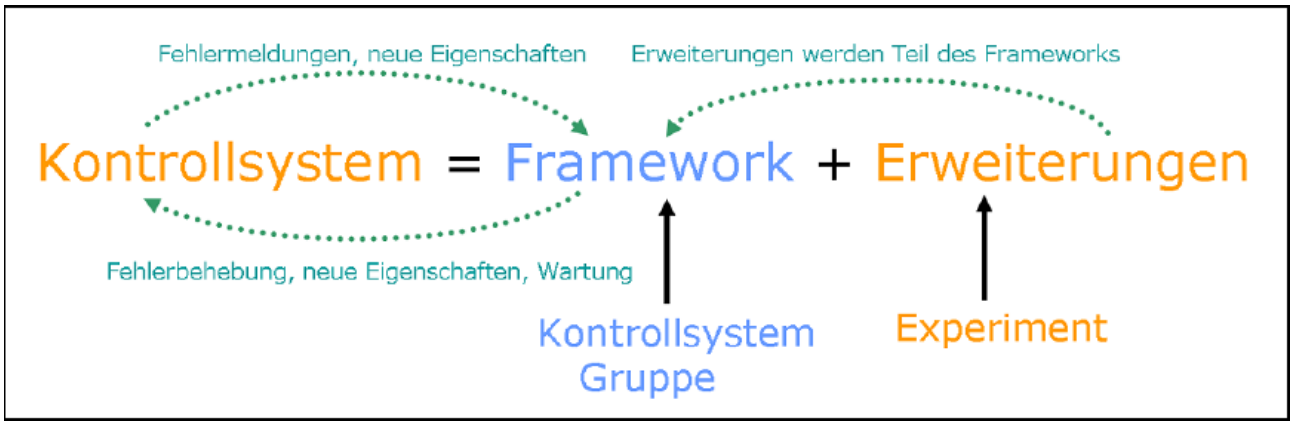

Abbildung 2: *Idee des CS-Framework*

2.1.2 Anforderungen

Im Vergleich zu einer industriellen Produktionsanlage, die eher als statisch zu bezeichnen ist, befindet sich ein Experimentaufbau im ständigen Wandel (siehe [Beck2003a]). Die verwendeten Geräte und deren Konfiguration ändern sich je nach Ziel des Experiments. Oftmals kommt eine Vielzahl an unterschiedlichen Gerätetypen zum Einsatz, die über verschiedene Schnittstellen und Protokolle angesprochen werden. Auch die Anzahl eines Gerätetyps kann mit der Ausbaustufe des Experiments variieren. Zunächst wird getestet, ob ein bestimmtes Gerät den Anforderungen des Experiments gerecht wird, erst anschließend wird die eigentlich benötigte Stückzahl angeschafft. Daher muss die zur Hardware passende Software möglichst frei skalierbar sein.
Ähnlich dynamisch wie der Experimentaufbau verhält es sich auch mit der Kontinuität der Entwickler, bei denen es sich meist um Studenten oder Doktoranden handelt, die dem Experiment maximal für einen Zeitraum von einigen Jahren zur Verfügung stehen. Daher muss der Umgang mit dem Framework bzw. mit dessen Entwicklungsumgebung leicht erlernbar sein, um die Einarbeitungszeit in das Framework möglichst gering zu halten. In diesem Zusammenhang ist es von besonderer Wichtigkeit, dass eine zentrale Gruppe das Wissen und die Dokumentation der Software-Entwicklungen verwaltet, da ansonsten mit den studentischen Entwicklern auch das Wissen über die von ihnen entwickelte Software verschwindet (siehe [Beck2003b]).
Eine weitere allgemeine Anforderung ist die Archivierung von Mess- und Konfigurationsdaten, damit der Experimentator nach Durchführung des Experiments zu Analysezwecken auf diese zurückgreifen kann. Die Überwachung von Sensordaten mittels Alarmfunktionalitäten ist ein elementarer Bestandteil eines Kontrollsystems. Diese so genannten SCADA[4]-Funktionalitäten müssen daher im Kern des Frameworks durch entsprechende Module bereitgestellt werden.
Aus Gründen der Sicherheit ist eine Fernsteuerung der Hardware in der Regel von Nöten. Während der Durchführung eines Experiments wird aufgrund der auftretenden ionisierenden Strahlung der Zugang zum Experimentaufbau gesperrt. Dies hat zur Folge, dass die Bedienung des Kontrollsystems und der direkte Zugriff auf die Hardware, örtlich voneinander getrennt werden müssen. Daher ist die Vernetzung der Rechner, die diese Aufgaben übernehmen, unerlässlich (siehe [Beck2003a]).
Man erhöht mit diesem Vorgehen auch die Skalierbarkeit des Systems, da man die Module einer Anwendung frei auf verschiedene Rechner verteilen kann. Somit wird die Leistungsfähigkeit eines einzelnen Rechners nicht zum limitierenden Faktor für die Skalierbarkeit eines Kontrollsystems. In diesem Zusammenhang spielt die

[4]Supervisory Control and Data Aquisition

Plattformunabhängigkeit der Entwicklungsumgebung eine große Rolle, da auf den Rechnern der Experimente oftmals verschiedene Betriebssysteme, hauptsächlich Windows und Linux, zum Einsatz kommen.

2.1.3 Lösung

Das *CS*-Framework basiert bis zur aktuellen Version 3.10 vollständig auf LabVIEW 8.2, da dieses viele der oben beschriebenen Anforderungen bereits erfüllt. LabVIEW dient hauptsächlich zur Programmierung von Anwendungen im Bereich der Mess- und Automatisierungstechnik und stellt daher standardmäßig eine Vielzahl von Treiberpaketen zur Ansteuerung diverser Hardware-Schnittstellen wie z.B. Profibus[5], GPIB[6] und RS232[7] zur Verfügung. Ebenso bietet National Instruments das DSC[8] Modul an, welches den Umfang von LabVIEW um SCADA-Funktionalitäten erweitert und den Zugriff auf OPC[9]-Serverapplikationen ermöglicht (siehe [Beck2003a]).
Ein weiterer Vorteil von LabVIEW ist dessen grafische Programmierung, die einen intuitiven Umgang mit der Programmiersprache zulässt. Verglichen mit einer textuellen Programmiersprache, fällt der Einstieg in LabVIEW sehr leicht, da der Programmierer sich nicht mit der Einhaltung von befehlsbedingter Syntax auseinandersetzen muss. LabVIEW folgt dem Prinzip des Datenflusskonzeptes. Analog zu einem Stromlaufplan lassen sich parallele Strukturen dadurch einfach verwirklichen.

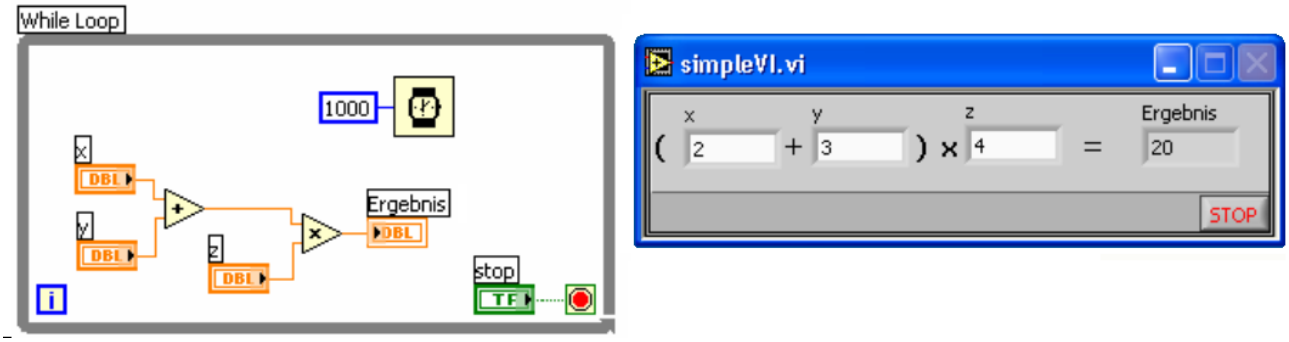

Abbildung 3: *Front Panel und Block Diagramm in LabVIEW*

Die Implementierung des Programmablaufs erfolgt in LabVIEW durch Erstellen eines Blockdiagramms, in das der Programmierer Funktionen in Form von Blöcken, die in LabVIEW als VIs[10] bezeichnet werden, setzt und diese miteinander verbindet. Zu jedem Blockdiagramm existiert ein Front Panel (siehe Abbildung 3), welches Elemente zur Dialoggestaltung bereitstellt. Die Implementierung von grafischen Bedienoberflächen wird dem Entwickler in LabVIEW daher extrem erleichtert.
Zudem unterstützt LabVIEW Multithreading[11] und verfügt über Mechanismen zur Ereignissteuerung und Synchronisierung, wie Warteschlangen und Semaphoren[12]. In Kombination mit den DIM[13]-Funktionen wurde damit ein Kommunikationsmechanismus geschaffen, der es ermöglicht, dass CS-Objekte rechnerübergreifend miteinander kommunizieren können. Darüber hinaus wird die CPU-Last aufgrund der Ereignissteuerung solange auf ein Minimum reduziert, wie keine Ereignisse im System auftreten.
Zum besseren Verständnis der Inhalte dieser Arbeit folgt ab Seite 8 eine Einführung in das *CS*-Framework. Die Objektorientierung und der Ereignismechanismus stehen im Vordergrund der Betrachtung, da daraus maßgebliche Rahmenbedingungen für den Entwurf der Klassenbibliothek resultieren.

[5] Process Field Bus ist ein Standard für die Feldbus-Kommunikation in der Automatisierungstechnik; siehe http://de.wikipedia.org/wiki/Profibus
[6] General Purpose Interface Bus; siehe http://de.wikipedia.org/wiki/IEC-625-Bus
[7] Standard für eine serielle Schnittstelle; siehe http://de.wikipedia.org/wiki/RS232
[8] Datalogging and Supervisory Control
[9] Openness, Productivity, Collaboration
[10] Virtual Instrument; siehe Glossar
[11] thread, siehe Glossar
[12] siehe Glossar
[13] siehe Glossar

2.1.4 Status

In der aktuellen Version 3.10 beinhalten die Basisklassen des *CS*-Framework die Semaphor geschützte Verwaltung der Objektattribute und die Einbettung des rechnerübergreifenden DIM-Ereignismechanismuses. Darüber hinaus existieren viele weitere Funktionalitäten, die das *CS*-Framework zu einem variabel anwendbaren Werkzeug machen.
Bis heute entstanden etwa 60 Geräteklassen für verschiedenste Gerätetypen von unterschiedlichen Herstellern. Entwickelt wurde das *CS*-Framework unter Windows. Allerdings konnte es erfolgreich auf Linux portiert werden (siehe [Beck2003b]).
Die Kontrollsysteme, die auf der Grundlage des *CS*-Framework bisher entstanden sind, beschränken sich nicht auf die GSI, sondern verteilen sich bereits über mehrere Forschungsinstitute. Im Einzelnen handelt es sich dabei um folgende Experimente, deren Kontrollsysteme mit dem *CS*-Framework entwickelt wurden bzw. sich momentan im Aufbau befinden (Stand 2005):

GSI (Deutschland)	**CERN** (Schweiz)	**MSU**[T1] (USA)
PHELIX	REXTRAP[T2]	LEBIT[T3]
SHIPTRAP	ISOLTRAP[T4]	
Motion CaveA[T5]		
RISING[T6]		
FOPI[T7]		

[T1] Michigan State University; Mehr Informationen http://www.msu.edu/.
[T2] Mehr Informationen http://rextrap.web.cern.ch/rextrap/.
[T3] Steht für Low Energy Beam and Ion Trap. Mehr Informationen http://www.nscl.msu.edu/tech/devices/lebit/.
[T4] Mehr Informationen http://rextrap.web.cern.ch/rextrap/.
[T5] Schrittmotorsteuerung für fahrbare Blenden und Leuchtschirme. Mehr Information siehe [Beck2005].
[T6] Mehr Informationen http://www-aix.gsi.de/~wolle/EB_at_GSI/main.html.
[T7] Steht für 4 Pi und spielt auf die Fähigkeit des gleichnamigen Detektors an, den kompletten Raumwinkel erfassen zu können. Mehr Informationen http://www.gsi.de/forschung/kp/kp1/experimente/fopi/index.html/.

Tabelle 1: *Verbreitung des CS-Frameworks*

2.2 Objektorientierung

Da LabVIEW keine Objektorientierung unterstützt[14], wurde eine objektorientierte Entwicklungsumgebung für das *CS*-Framework mit den Funktionen, die LabVIEW zur Verfügung stellt, realisiert. Die folgenden Abschnitte erläutern, wie die grundlegenden Prinzipien der Objektorientierung wie Instanzierung, Attributverwaltung und Vererbung im Rahmen des *CS*-Frameworks umgesetzt wurden. Dabei ist allerdings zu beachten, dass es sich um keine Objektorientierung im Sinne einer Programmiersprache wie Java oder C++ handelt. Viel mehr spricht man im Falle des *CS*-Frameworks von einem objektorientierten Ansatz. Dies äußert sich beispielsweise in dem komplexen Vererbungsprozess, dessen manuelle Ausführung einerseits einige Zeit in Anspruch nimmt, andererseits ein entsprechendes Hintergrundwissen über das *CS*-Framework voraussetzt. Darüber hinaus gibt es viele objektorientierte Prinzipien wie z.B. die Sichtbarkeit von Methoden und Attributen oder abstrakte Klassen, die im *CS*-Framework nur per Konvention existieren. Der LabVIEW-Compiler weiß nichts von der Objektorientierung des *CS*-Framework und kontrolliert daher nur die Einhaltung der LabVIEW Syntax. An dieser Stelle ist die Disziplin des Entwicklers gefordert, sich beim Implementieren an die *CS*-Konventionen zu halten. Hierfür bietet das *CS*-Framework zahlreiche Entwurfsvorlagen, anhand derer ein Entwickler seine Klassen aufbauen kann.
Bei Instanzen einer *CS*-Klasse handelt es sich im Allgemeinen nicht um passive Datencontainer, die Methoden zur Datenmanipulation bereitstellen, sondern viel mehr um aktive Prozesse, die miteinander interagieren und unabhängig von Benutzerinteraktionen aktiv werden können. Hierzu zählt beispielsweise das Versenden von Ereignissen zwischen Objekten oder der Übergang zwischen zwei Zuständen als Reaktion auf äußere Veränderungen. Aktive *CS*-Objekte sind threads[15], deren generisches Verhalten in Form von Basisklassen vom Framework bereitgestellt wird.

[14] Erst seit Version 8.2 ist Objektorientiertes programmieren auch direkt mit LabVIEW möglich
[15] Siehe Glossar

2.2.1 Klassen und Instanziierung

Eine *CS*-Klasse besteht, wie es in der Objektorientierung üblich ist, aus Attributen, Konstruktor, Destruktor und weiteren Methoden, die auf den Attributen operieren. Zusätzlich besitzt jede aktive *CS*-Basisklasse ein VI Template[16]. Dieses enthält den aktiven Code einer Klasse.
Bei der Instanzierung eines Objektes wird der Konstruktor der entsprechenden Klasse aufgerufen, welcher zur Laufzeit eine Instanz des VI Templates in den Speicher lädt und eine dazugehörige, eindeutige Referenz erzeugt (siehe [Brand2005] Seite 75). Hierfür verwendet das *CS*-Framework die VI Server-Funktionen von LabVIEW. Abbildung 4 und 5 zeigen den dynamischen Aufruf von Methoden und VI Templates. Die Referenz

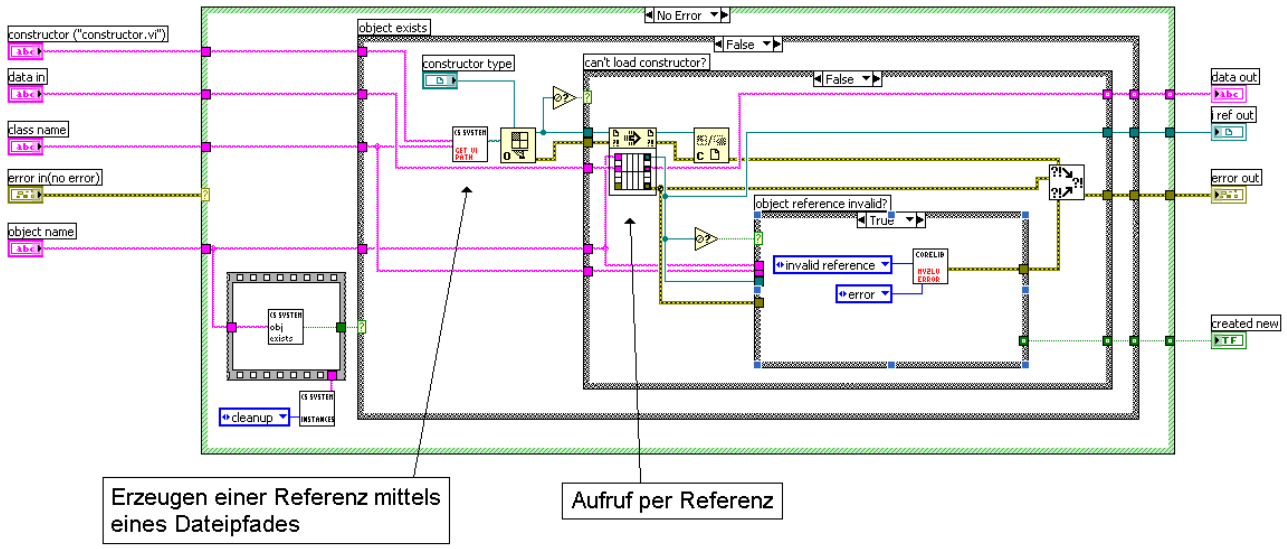

Abbildung 4: *LabVIEW-Blockdiagramm: CSSystem._new.vi dient dem dynamischen Aufruf des Konstruktors einer Klasse*

auf die Instanz des VI Templates speichert die Klasse in ihren Attributen, um diese im Destruktor schließen zu können und den allokierten Platz im Speicher wieder frei zu geben. Unter einer Objektreferenz versteht man im *CS*-Framework die Referenz auf die Instanz der Klasse CSObj. Sie dient als Oberklasse aller CS-Klassen.
Ein VI Template einer Klasse besteht aus einem oder mehreren threads[17], die die klassenspezifischen Aufgaben ausführen. Eine genaue Beschreibung der VI Templates der CS-Basisklassen befindet sich unter Abschnitt 2.3.2.
Für alle Methoden einer Klasse, die in Form von VIs abgespeichert werden, existiert die Namenskonvention "CLASSNAME.METHODNAME.vi". Zusammen mit der Verzeichnisstruktur der CS-Klassenbibliothek verwaltet das Framework die Dateipfade der VI's, die zum Aufruf entsprechender Methoden einer Klasse, wie z.B. Konstruktor oder Destruktor, benötigt werden (siehe [Wiki1]).
Das *CS*-Framework bietet dem Entwickler eine Applikation, die die Vererbung von Klassen vereinfacht, da das manuelle Vererben im *CS*-Framework einen relativ großen Zeitaufwand erfordert, der durch diese Anwendung automatisiert und beschleunigt wird.
Für Debugging- und Testzwecke existiert eine Anwendung, die grundlegende Klassenfunktionalitäten wie den Zugriff auf die Attribute oder die Instanzierung einzelner Templates überprüft.

2.2.2 Objekt und Attributverwaltung

Die Objektreferenzen sowie Referenzen auf Template-Instanzen werden innerhalb eines *CS*-Systems zusammen mit dem dazugehörigen Objekt- und Klassennamen in einem nicht initialisierten Shiftregister gespeichert (siehe Abbildung 28 im Anhang). Der Registerinhalt wird aktualisiert, sobald eine neue Instanz erzeugt oder eine bereits vorhandene zerstört wird.
Derselbe Mechanismus wird zur Attributverwaltung verwendet. Zu jeder Klasse existiert ein VI (siehe Abbildung 29 im Anhang) welches, ebenfalls in einem Shiftregister, die Referenzen und die Attribute aller Instanzen einer Klasse verwaltet. Im Gegensatz zu dem vorher verwendeten ObjektVIEW, welches die Attribute in den Anzeigeelementen des Front Panels speichert, ist der Zugriff auf ein Shiftregister um etwa den

[16] Virtual Instrument Template; siehe Glossar
[17] siehe Glossar

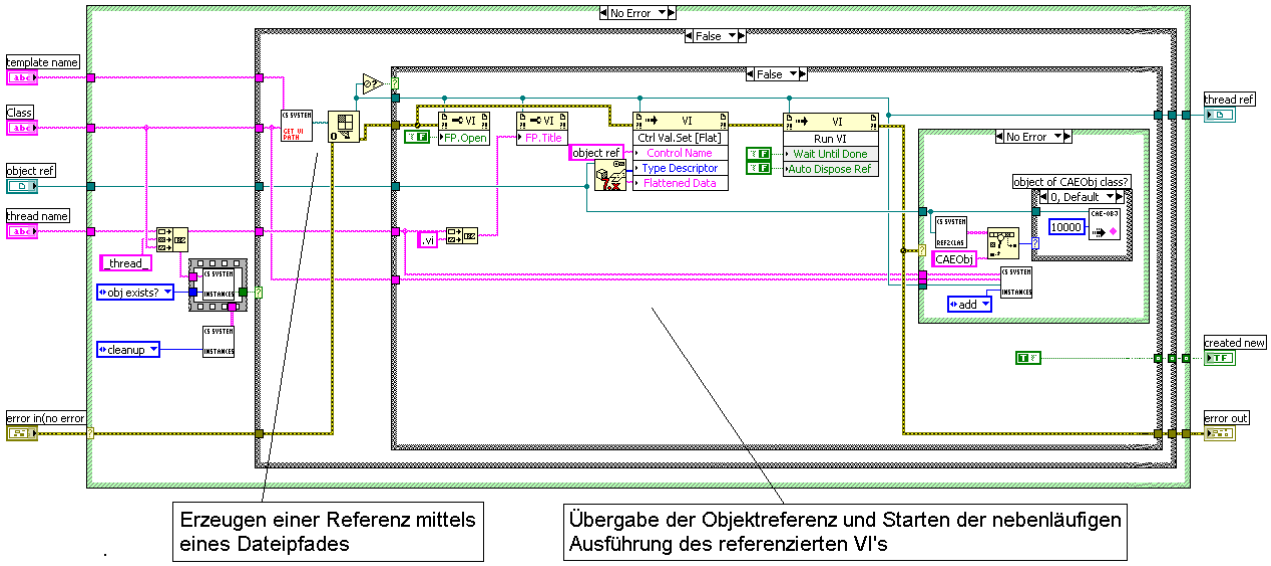

Abbildung 5: *LabVIEW-Blockdiagramm: CSSystem._startThread.vi startet eine Instanz eines Klassen-Templates*

Faktor 100 schneller, da ein Thread-Wechsel zwischen dem Blockdiagramm und dem Front Panel vermieden wird (siehe [Brand2005] Seite 75).
Die grundlegenden Eigenschaften eines *CS*-Objektes sind in der Basisklasse CSObj zusammengefasst. Sie beinhaltet die Zugriffsmethoden auf die Attribute eines Objektes. Da im *CS*-Framework ein Objekt aus mehreren threads bestehen kann, die asynchron nebeneinander laufen, benötigt man eine Möglichkeit, den parallelen Zugriff auf die Attribute eines Objekts zu verhindern, um so deren Konsistenz sicherzustellen. Ein solcher Ausschlussmechanismus ist mit Hilfe von Semaphoren, die von LabVIEW zur Verfügung gestellt werden, realisiert, und kann bei jedem Attributzugriff optional verwendet werden.

2.2.3 Vererbung

Innerhalb des *CS*-Frameworks spiegeln sich Vererbungsbeziehungen in der Schachtelung von Konstruktoren wieder. Im Falle der einfachen Vererbung wird der Konstruktor der Oberklasse im Konstruktor der Unterklasse aufgerufen. Mehrfachvererbung ist ebenfalls möglich, indem mehrere Konstruktoraufrufe innerhalb eines Konstruktors stattfinden. Kommt es dadurch zum mehrfachen Aufruf des Konstruktors einer Oberklasse, wird diese immer nur einmal vererbt.
Zu beachten ist in diesem Zusammenhang auch, dass ein Konstruktoraufruf außerhalb eines Konstruktors eine neue Instanz der Klasse erzeugt und nicht der Vererbung dient. Soll innerhalb eines Konstruktors eine neue Instanz erzeugt werden, geschieht dies mittels des NEW-Operators (siehe Abbildung 4) (siehe [Brand2005] Seite 76). Auf diese Weise können assoziative Klassenbeziehungen wie Kompositionen oder Aggregationen implementiert werden.

Grundsätzlich gillt per Konvention:
Erzeugen neuer Objekte → Verwendung des NEW-Operators;
Vernichten existierender Objekte → Verwendung des DELETE-Operators;
Realisieren einer Vererbung → Konstruktor-Aufruf innerhalb eines Konstruktors.

Abbildung 30 im Anhang zeigt die Schachtelung der Konstruktoren anhand der Vererbungshierarchie des BaseProcess Konstruktors.

2.3 Klassenbibliothek

2.3.1 Drei-Schichten-Architektur

Um die Modularität und Flexibilität des Frameworks zu gewährleisten, wird eine Drei-Schichten-Architektur verwendet, wie sie häufig in der Informatik anzutreffen ist (siehe [Balzert2005] Seite 448). Durch einheitliche Schnittstellen und Kapselung zusammengehöriger Aufgaben in Klassen, können Komponenten einer Schicht ohne großen Aufwand ausgetauscht werden, ohne den kompletten Quellcode einer Anwendung überarbeiten zu müssen. Dies setzt allerdings eine strikte Kategorisierung der *CS*-Klassen in Geräteschicht, Anwendungsschicht und Operationsschicht voraus.
Gemäß dem Schichtenmodell, dessen Grundgedanke häufig Ausgangspunkt für den Entwurf von modularen Informationssystemen darstellt, benutzen weiter oben liegende Schichten die Funktionalitäten der unteren Schichten. Das wohl bekannteste Beispiel hierfür ist das OSI-Modell[18]. Die Schichten des *CS*-Frameworks und deren Aufgaben werden im Folgenden näher erläutert:

Geräteschicht Die unterste der drei Schichten stellt die Schnittstelle zur Hardware her. Dazu kapseln Geräteklassen die Treibersoftware, um eine einheitliche Schnittstelle für den Zugriff auf Geräte innerhalb des *CS*-Framework zu ermöglichen. Die Instanz einer Geräteklasse repräsentiert in der Regel eine Hardwarekomponente. Zur Laufzeit kann man durch Instanzierung weiterer Objekte ein Kontrollsystem dynamisch an die tatsächliche Anzahl von Geräten und Gerätetypen anpassen.

Anwendungsschicht Klassen der Anwendungsschicht verwenden Instanzen der Geräteschicht, um die Gerätefunktionalitäten in einen für den Anwender sinnvollen Zusammenhang zu bringen. Typische Aufgaben der Anwendungsschicht sind die Bereitstellung von SCADA-Funktionalität, Sequenzern oder eine Benutzerverwaltung.

Operationsschicht Auf dieser Ebene wird typischerweise die Schnittstelle zum Benutzer in Form von grafischen Bedienoberflächen hergestellt. Durch die Aufteilung der Klassen in Schichten können Bedienoberflächen auf eine spezifische Anwendung angepasst werden ohne die Prozesse im Hintergrund verändern zu müssen.

2.3.2 Systemklassen

Das *CS*-Framework stellt einige abstrakte[19] Basisklassen bereit. In Abbildung 6 sind diese dunkel unterlegt. Mittels Vererbung kann der Entwickler darauf aufbauend seine eigenen Klassen entwerfen. Einen wichtigen Mechanismus stellen in diesem Zusammenhang virtuelle Methoden[20] der Basisklassen dar, deren Schnittstelle bereits in den Basisklasse definiert ist. Die Methode selbst wird jedoch erst in der Unterklasse implementiert.

CSObj Die Basisklasse aller *CS*-Klassen erzeugt in ihrem Konstruktor eine Referenz auf die Instanz ihres VI Templates. Diese wird an die Konstruktoren der abgeleiteten Klassen zurückgegeben und dient als Objektreferenz, die eindeutig das Objekt identifiziert. Mit ihrer Hilfe können sowohl Informationen wie Instanz-, Klassenname und die Namen der Oberklassen abgefragt, als auch auf die Objektattribute zugegriffen werden.
Zusätzlich verwaltet die Klasse einen Semaphor, der den wechselseitigen Zugriff auf gemeinsam genutzte Ressourcen durch verschiedene threads eines Objektes verhindert. Dieser Mechanismus wird vor allem für den konsistenten Schreib- und Lesezugriff auf die Objektattribute verwendet.
Es handelt sich nicht um eine aktive Klasse, da das dazugehörige VI Template nur der Erzeugung der Objektreferenz dient, selbst jedoch keine threads oder sonstigen Code beinhaltet.

CAEObj Die Klasse stellt die Methoden für die Handhabung des Ereignismechanismus bereits. Dazu gehört das Erzeugen, Versenden und Empfangen von Ereignissen. Die Klasse besitzt selbst keinen thread zur Verarbeitung eintreffender Ereignisse, sondern stellt nur die Methoden bereit, die eine solche Klasse benötigt. Zusätzlich bietet die Klasse die Möglichkeit den CS-Reservierungsmechnaismus zu nutzen.
Der Ereignismechanismus dieser Klasse benutzt C-Bibliotheken, welche den DIM-Ereignismechanismus kapseln. Mehr dazu in Abschnitt 2.4.1.

[18] Open Systems Interconnection; Schichtenmodell für die Kommunikation offener, Informationsverarbeitender Systeme. siehe http://de.wikipedia.org/wiki/OSI-Schichtenmodell
[19] Abstrakte Klassen dienen im Gegensatz zu konkreten Klassen nur der Vererbung
[20] siehe Glossar

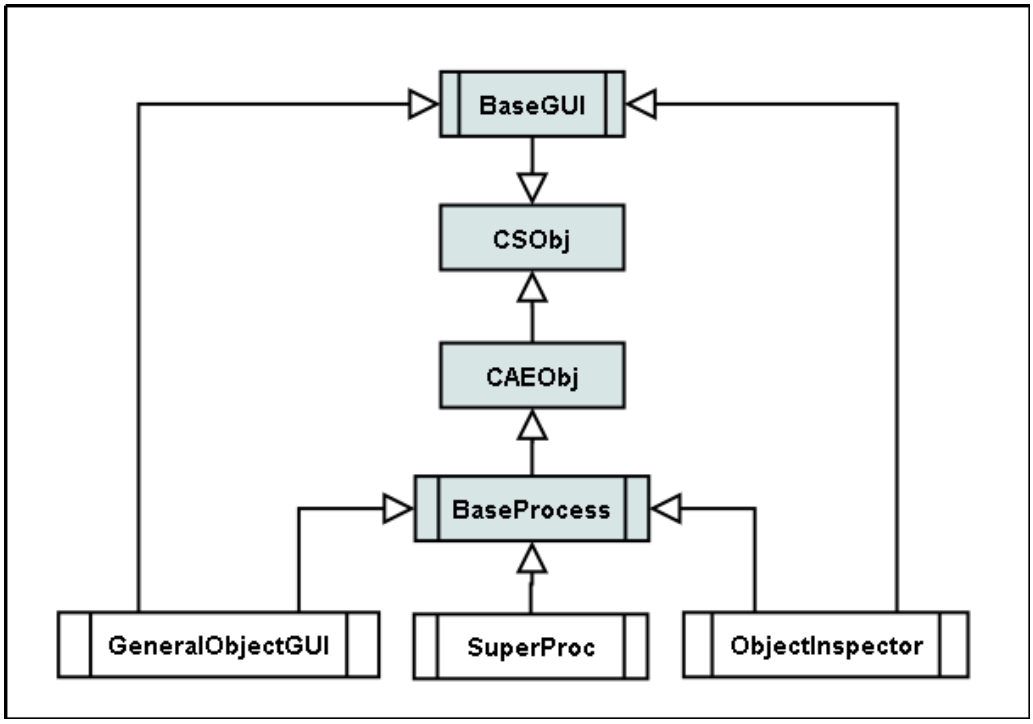

Abbildung 6: *UML-Klassendiagramm: Basisklassen des CS-Framework*

BaseGUI Diese Klasse dient als Vorlage für die Implementierung von Benutzeroberflächen. Neben einem thread für das GUI[21], beinhaltet die Klasse BaseGUI Methoden, um auf Parameter des GUI's programmatisch zugreifen zu können. Dazu zählen beispielsweise die Größe und Position des Front Panels und deren Bedienelemente oder dessen Sichtbarkeit.

BaseProcess Die BaseProcess Klasse besteht hauptsächlich aus zwei threads: die Ereignisschleife und die periodische Schleife, welche beide über eine Watchdog[22]-Funktionalität verfügen (siehe [Beck2003b]). Die periodische Schleife dient der Durchführung von zyklischen Aktionen, was z.B. im Falle mancher Geräte benötigt wird, um diese periodisch auszulesen oder abzufragen.
Die wichtigere Funktionalität besteht in der Ereignis-Schleife, die die Kommunikation zwischen Objekten ermöglicht. Als Unterklasse der Klasse CAEObj verfügt die Klasse BaseProcess über Methoden zur Ereignissteuerung. Jede Instanz einer Unterklasse der BaseProcess-Klasse besitzt mit der Ereignis-Schleife einen thread, der auf eintreffende Ereignisse wartet. Sobald ein Ereignis eintrifft, wird die Ereignis-Schleife aktiv und ruft über ihre virtuelle Methode die entsprechenden Methode der Unterklasse auf. Zu diesem Zweck besitzt jede Klasse, die von BaseProcess geerbt hat, eine Methode namens "CLASSNAME.ProcCases.vi", in die der Entwickler seine Ereignisbehandlungsmethoden implementiert. Ebenso müssen in einer weiteren Methode namens "CLASSNAME.ProcEvents.vi" die Namen der Ereignisse, die ein Objekt der Klasse empfangen kann, definiert werden. Dasselbe Prinzip wird auch bei der periodischen Schleife verwendet. Die Schnittstelle ist in der Basisklasse BaseProcess implementiert, die aufgerufene Methode in der Unterklasse. Die entsprechende Methode für die periodische Schleife heisst "CLASSNAME.ProcPeriodic.vi".
Eine weitere Fähigkeit des BaseProcess besteht in dem Auslesen einer Datenbank, die Datensätze für Prozesse enthält (siehe [Brand2005]). Da die Klasse BaseProcess meist als Oberklasse für Geräteklassen verwendet wird, enthält die Datenbank in diesen Fällen meist Daten zur Konfiguration der Hardware, wie zum Beispiel Busadressen.

[21] Graphical User Interface; deutsche Übersetzung: grafische Benutzeroberfläche
[22] siehe Glossar

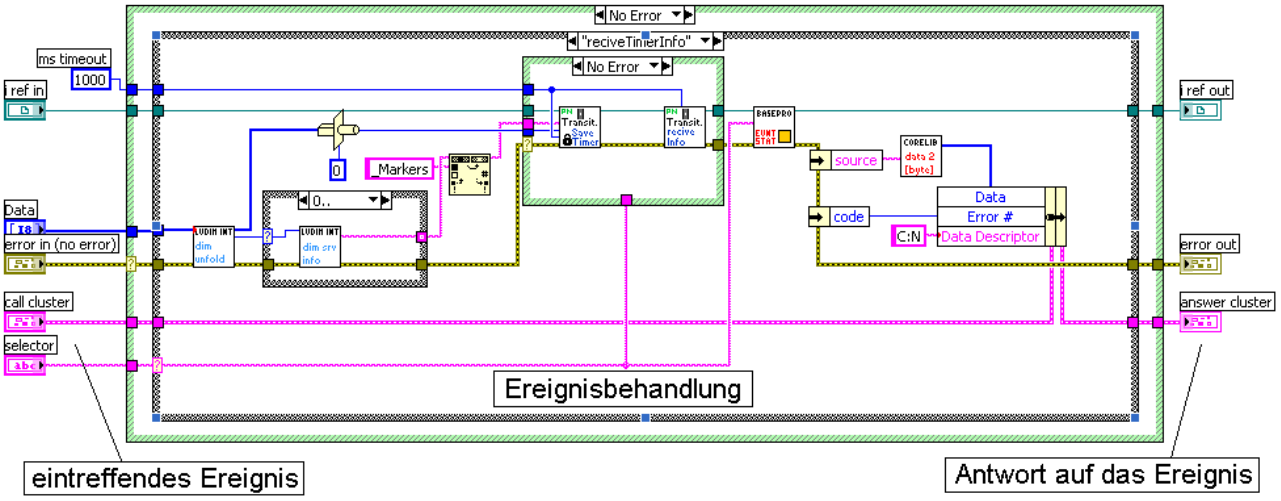

Abbildung 7: *LabVIEW-Blockdiagramm: Beispiel eines ProcCases.vi für die Ereignisbehandlungsmethode eines Prozesses*

Neben den Basisklassen, die dem Entwickler als Grundlage für eigene Klassen dienen sollen, beinhaltet das *CS*-Framework noch einige weitere Klassen, die für den Betrieb des Frameworks von großer Wichtigkeit sind:

SuperProc Eine Instanz namens "SystemID\Super" der Klasse SuperProc wird beim Start eines *CS*-Systems instanziert. Sie dient als Container-Objekt für alle *CS*-Objekte und ist per *CS*-Konvention ein Singleton-Objekt[23]. Ihre Aufgabe besteht im Erzeugen, Zerstören und Reservieren von Prozessen. Reservierte Prozesse können, solange noch bestehende Reservierungen existieren, nicht vom SuperProc zerstört werden.

ObjectInspector Die Instanz dieser Klasse dient dem Entwickler als Werkzeug zum Überwachen und Debuggen. Sie zeigt alle *CS*-Instanzen und deren Status an. Je nach Klasse des Objektes können Informationen über die Zustandsmaschine, die Lebenszeit der Instanz, usw. angezeigt werden. Ebenfalls ermöglicht sie dieselben Informationen von anderen *CS*-Systemen über das Netzwerk abzufragen und anzuzeigen.

GeneralObjectGUI Ähnlich wie der Object Inspector stellt diese Klasse dem Entwickler eine GUI zur Verfügung, mit deren Hilfe Objekte instanziert und zerstört werden können. Darüber hinaus können Ereignisse an Objekte versandt werden, um während der Entwicklungsphase deren Verhalten zu testen oder im Betrieb Einstellungen vorzunehmen. All diese Funktionalitäten sind optional auch automatisierbar, sodass beim Start oder auf Wunsch des Entwicklers eine Liste von Objekten instanziert wird und diesen bestimmte Ereignisse zugeschickt werden, um ein System in einen Anfangszustand zu versetzen oder verschiedene Einstellungen zu laden.

2.4 Ereignismechanismus

Sobald eine Klasse von der CSObj Klasse erbt, hat diese die Möglichkeit Ereignisse durch den *CS*-Ereignismechanismus zu erhalten. Eine vordefinierte Struktur zum Empfangen und Reagieren auf einzelne Ereignise erhält man durch Vererbung von der BaseProcess Klasse.

2.4.1 DIM

Vor *CS*-Version 3.0 existierte ein Ereignismechanismus, welcher allein auf LabVIEW basierte. Wegen zu schlechter Performance und einer zu umfangreichen Kontrollstruktur, welche aus QueueClient, QueueServer und QueueListener bestand, war es nötig, einen neuen Ereignismechanismus zu implementieren. Das vom CERN verwendete DIM-Protokoll[24] erfüllte alle Voraussetzungen, die das *CS*-Framework benötigt und wird nun seit *CS*-Version 3.0 verwendet.
Durch eine Wrapper-Bibliothek kann DIM auch in LabVIEW und damit im *CS*-Framework genutzt werden. DIM basiert auf einem Publisher-Subscriber Mechanismus und gewährleistet durch seinen einfachen Aufbau

[23] Es existiert nur eine Instanz der Klasse auf jedem *CS*-System. Siehe [Brand2005] Seite 89
[24] siehe Glossar

eine schnelle und unkomplizierte rechnerübergreifende Kommunikation. Detalierte Informationen zum Aufbau des DIM-Protokolls befinden sich in den Abhandlungen [Gaspar1993] und [Gaspar2000](siehe Literaturverzeichnis). Abbildung 8 zeigt die grundlegenden Zusammenhänge der DIM-Kommunikationsstrukturen.

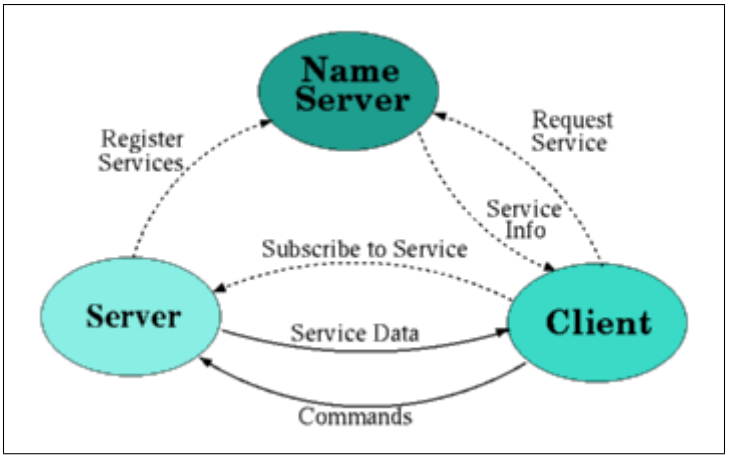

Abbildung 8: *Aufbau der DIM-Kommunikationsstruktur*

2.4.2 Ereignistypen

Service Die Instanz, die einen Dienst[engl. Service] bereit stellen will (des weiteren Server genannt), registriert all ihre Dienste beim sogenannten "DIM-Nameserver". Will nun eine andere Instanz (auch Client genannt) bereitgestellte Dienste abonnieren, so erhält sie die Namen der Dienste und deren Server vom DIM-Nameserver. Der Client kann sich nun direkt beim Server für den jeweiligen Dienst eintragen, und erhält automatisch ein Ereignis, sobald sich die Daten dieses Dienstes ändern (siehe auch Abbildung 8).

Command Des weiteren besteht in DIM die Möglichkeit auf sogennante "Kommandos"[engl. command] zu reagieren. Genauso wie beim Registrieren der Dienste, werden auch Kommandos beim DIM-Nameserver registriert. Will eine Instanz ein Kommando absenden, so befinden sich alle dazu benötigten Informationen beim DIM-Nameserver.

2.4.3 Leistungsfähigkeit

Anders als mit dem vorherigen CS-Ereignismechanismus, hat DIM die Möglichkeit die volle Bandbreite einer Netzwerkverbindung zu nutzen. Die so erreichte höhere Performance wird nun nicht mehr durch das Protokoll limitiert, sondern einzig durch die Leistungsfähigkeit der verwendeten Verbindung. Auch benötigt DIM keine eindeutigen Computernamen mehr, wie es im alten Ereignisprotokoll der Fall war. Daraus resultierend besteht nun die Möglichkeit, mehr als eine *CS*-Instanz pro Computer zu erzeugen.

3 Definition

3.1 Motivation

ObjectVIEW bot mit Hilfe vorhandener Basisklassen die Möglichkeit, Objekt- und Petri-Netze zu benutzen. Da ObjectVIEW an der GSI nicht mehr zum Einsatz kommt, ist es sehr wünschenswert, auch für das *CS*-Framework Objekt- und Petri-Netze zur Verfügung zu haben. Da beide Netzformen als ausführbare Designdokumente dienen, helfen sie sehr beim späteren Verständnis von Programmcode und bei der Prozesssteuerung selbst.
Die Objekt-Netze im *CS*-Framework haben alle Vorteile, die auch Objekt-Netze in ObjectVIEW boten. Darüber hinaus können CS-Objekt-Netze ein verteiltes Netzwerk kontrollieren. Sie unterscheiden sich in vielen Punkten mehr oder minder stark vom Vorgänger.
Ähnlich verhält es sich bei den Petri-Netzen, welche als Alternative zu den *CS*-Sequenzer-Klassen eingesetzt werden können. Gerade bei der Steuerung verteilter Systeme braucht man oft Sequenzer, um logische und zeitliche Verknüpfungen zwischen verschiedenen Geräten zu realisieren. Die CS-Petri-Netze stellen dem Benutzer ein einfaches und elegantes Werkzeug zur Verfügung, auch komplexe Sequenzer schnell und unkompliziert implementieren zu können.

3.2 Funktionsweise

3.2.1 Objekt-Netze

Ein *CS*-Objekt-Netz ist ein Kontrollkonstrukt, welches es dem Benutzer erlaubt, festgelegte Strukturen von *CS*-Objekten in einem verteiltem System zu starten. Im Objekt-Netz kann man die Ports festlegen, über welche die *CS*-Objekte miteinander kommunizieren. Die Daten, die durch diese Ports gesendet werden, lassen sich im Objekt-Netz ohne Einschränkung des Datentypes visualisieren. Bei der Benutzung eines Nicht-Standard-Datentyps kann der Anwender einfach und problemlos ein eigenes VI zum Anzeigen der dazugehörigen Daten schreiben. Abbildung 9 zeigt ein Beispiel-Netz, welches die oben beschriebenen Zusammenhänge verdeutlicht. Die Abbildung beschreibt den Aufbau einer Heizungssteuerung, welche im Abschnitt 4.4 noch einmal näher erläutert wird.

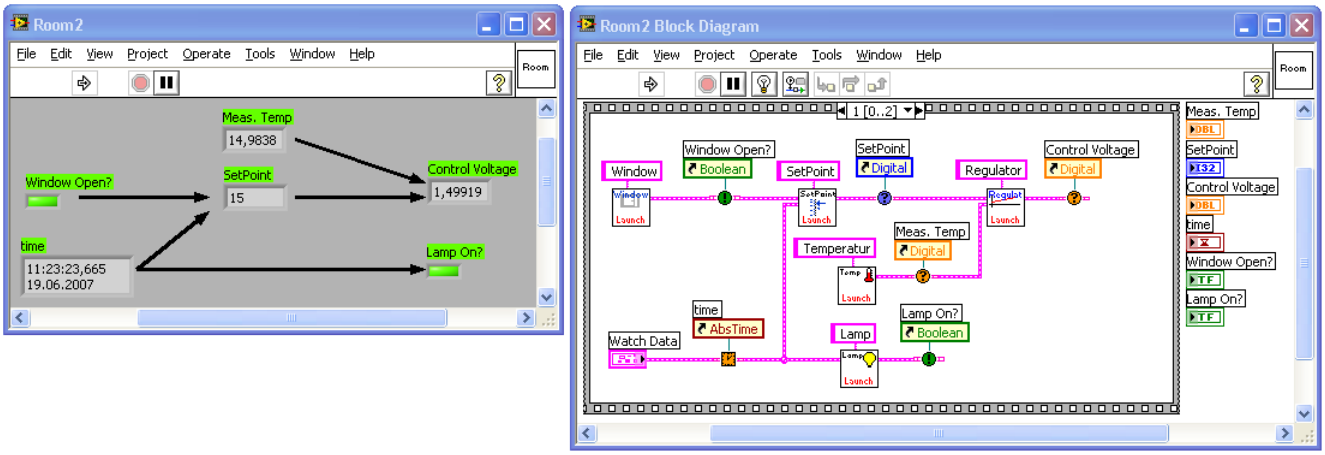

Abbildung 9: *Unternetz des Heizungssteuerung-Beispiels*

Generell werden alle Objekte im Objekt-Netz automatisch überwacht. Fällt eines der Objekte aus oder hat schwerwiegende Fehler, so wird dieses umgehend neu gestartet und mit den richigen Port-Informationen initialisiert.

3.2.2 Definition der Petri-Netze

Ein Petri-Netz ist ein mathematisches Modell von nebenläufigen[25] Systemen. Es stellt eine formale Methode der Modellierung verschiedenster Abläufe dar. Petri-Netze wurden durch Carl Adam Petri in den 1960er Jahren definiert. Sie verallgemeinern wegen der Fähigkeit, nebenläufige Ereignisse darzustellen, die Automatentheorie[26].
Heute existieren mehrere verschiedene Typen von Petri-Netzen, welche sich im Allgemeinen nicht viel von einander unterscheiden. Hier wird nur auf die von Bernd Baumgarten vorgestellten Netze [Baumgarten2006] eingegangen, alle anderen Formen weisen meist nur unbedeutende Modifikationen auf.

Definition Netzgraphen

Ein Netz oder Netzgraph ist ein Tripel N = (S,T,F) derart, dass

$S \cap T = \emptyset$ und

$F \subseteq (S \times T) \cup (T \times S)$.

Die Elemente von S nennt man Stellen, sie werden dargestellt durch Kreise;

die Elemente von T nennt man Transitionen, sie werden dargestellt durch Balken oder Rechtecke;

die Elemente von F nennt man Kanten, sie werden dargestellt durch Pfeile[27].

Vorbereich einer Stelle (oder einer Transition) x:

$\bullet x := \{y | (y, x) \in F\}$,

die Menge aller Input-Transitionen bzw. Input-Stellen.

Nachbereich einer Stelle (oder einer Transition) x:

$x\bullet := \{y | (x, y) \in F\}$,

die Menge aller Output-Transitionen bzw. Output-Stellen.

Definition ST-System

Ein 5-Tupel Sys = (S,T,F,W,M_0) heist ein **Stellen-Transitions-System** oder **ST-System**, wenn:

- (S,T,F) ein Netz ist,
- $W : F \to \mathbb{N}$ (Kantengewichte), und
- $M_0 : S \to \mathbb{N}_0$ gilt (Anfangsmarkierung des ST-Systems).

Jede Abbildung $M : S \to \mathbb{N}_0$ heißt Markierung.

[25] Nebenläufigkeit liegt vor, wenn mehrere Ereignisse in keiner kausalen Beziehung zueinander stehen, sich also nicht beeinflussen. Ereignisse sind nebenläufig, wenn keines eine Ursache des anderen ist. Oder anders ausgedrückt: Aktionen können nebenläufig ausgeführt werden (sie sind parallelisierbar), wenn keine das Resultat der anderen benötigt.(Wikipedia)

[26] Neuronale Netze, Zelluläre Automaten und Petri-Netze haben die Fähigkeit nebenläufige Ereignisse darzustellen. Einfache Automaten, wie Kellerautomaten, Registermaschinen oder gewöhnliche Automaten besitzen diese Fähigkeit nicht.(Wikipedia)

[27] siehe auch Abbildung 11

Für jede Markierung M wird definiert: Eine Transition t ist **aktiviert** unter M, geschrieben als $M[t\rangle$, wenn $\forall s \in \bullet t : M(s) \geq W(s,t)$.
In diesem Falle kann[28] t **schalten** (oder **feuern**), was M in die Folgemarkierung Mt, geschrieben als $M[t\rangle Mt$, überführt:

$$Mt(s) = \begin{cases} M(s) - W(s,t) & wenn\ s \in \bullet t \setminus t\bullet \\ M(s) + W(t,s) & wenn\ s \in t\bullet \setminus \bullet t \\ M(s) + W(t,s) - W(s,t) & wenn\ s \in \bullet t \cap t\bullet \\ M(s) & sonst. \end{cases}$$

Graphisch werden Kantengewichte durch Zahlen dargestellt, welche man an den Kanten notiert (Voreinstellung:1). Stellenmarkierungen M(s) werden durch eine entsprechende Anzahl von Punkten innerhalb eines Platzes visualisiert.

Inzidenzmatrix

Sei Sys = (S,T,F,W,M_0) ein ST-System, und seien S = $\{s_1, s_2, ..., s_m\}$ und T = $\{t_1, t_2, ..., t_n\}$. Dann sind die Elemente der $m \times n$ Inzidenzmatrix C = Inc(Sys) definiert durch:

$\forall 1 \leq i \leq m, 1 \leq k \leq n :$

$$C_{i,k} = \begin{cases} W(t_k, s_i) & wenn\ (s_i, t_k) \notin F, (t_k, s_i) \in F \\ -W(s_i, t_k) & wenn\ (s_i, t_k) \in F, (t_k, s_i) \notin F \\ W(t_k, s_i) - W(s_i, t_k) & wenn\ (s_i, t_k) \in F, (t_k, s_i) \in F \\ 0 & sonst. \end{cases}$$

Fakt: Wenn das ST-System Sys schleifenfrei[29] ist, dann ist (S,T,F,W) bis auf "Isomorphie" eindeutig durch Inc(Sys) bestimmt.

Da $C_{i,k}$ angibt, wie sich die Markenzahl auf der Stelle s_i ändert, wenn die Transition t_k schaltet, zeigt $C_{\bullet,k}$ (die k-te Spalte von C) die Veränderung der ganzen Markierung in diesem Falle, d.h. $Mt_k = M + C_{\bullet,k}$ (wenn man s_i und i "identifiziert").

Abbildung 10 zeigt die Inzidenzmatrix zum Netzgraphen aus Abbildung 11. Die so erzeugten Matrix kann nun in einer sog. linearen Analyse verwendet werden, um strukturelle Beschränktheit, eine S-Invariante und andere Eigenschaften des Systems zu bestimmen. Auf die lineare Analyse wird hier nicht weiter eingegangen. Detailierte Informationen sind ab Seite 20 in [Baumgarten2006] zu finden.

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 2 & -2 & -2 \end{pmatrix}$$

Abbildung 10: *Inzidenzmatrix zum Netzgraphen aus Abbildung 11*

[28] Wie üblich, gehen wir in dieser Definition der ST-Systeme nicht auf Konfliktsituationen ein, in denen mehrere Transitionen ("gleichzeitig") schalten können. In den ausprogrammierten CS-Petri-Netzen **muss** eine Transition schalten, sobald dies möglich ist. Bei einer Gleichberechtigung mehrerer Transitionen entscheidet dort der Zufall.

[29] Ein ST-System ist schleifenfrei, wenn gilt: $\forall t \in T, \forall s \in S : s \notin \bullet t \cap t\bullet$

3.2.3 Verwendung der Petri-Netze

Die Hauptkomponenten eines Netzgraphen (nach Bernd Baumgarten [Baumgarten2006]) sind Plätze (Stellen) und Transitionen. Man stellt einen Platz im Diagramm als Kreis, und eine Transition als Rechteck dar (siehe auch Abbildung 11). Plätze und Transitionen verbindet man nun untereinander. Hierbei achtet man darauf, keinen Platz mit einem Platz verbinden und keine Transition mit einer Transition. Das dadurch entstehende Netz ist ein bipartiter Graph (siehe Abbildung 11). Den Kanten gibt man nun eine Richtung und versieht diese bei Bedarf mit Gewichtungen (ungewichtete Kanten haben per Konvention ein Gewicht von eins).
Ein Platz dient als Container für Marken. Marken definieren die verschiedenen Zustände[30], die ein Netz einnehmen kann. Ist die Zahl der Zustände endlich, so bezeichnet man das Netz als beschränkt.
Vor dem "Start" eines Petri-Netzes gibt man eine bestimmte Markenzahl auf jedem Platz vor. Sobald nun eine Transition "schaltet" ändert sich die Markenzahl und die Verteilung der Marken im Netz. Eine Transition kann schalten, sobald jeder Vorgänger-Platz mindestens so viele Marken hat, wie das zur Transition gehörige Kantengewicht angibt.
Schaltet die Transition, vernichtet sie die benötigten Marken aller Vorgänger-Plätze und erzeugt neue Marken auf allen Nachfolger-Plätzen. Durch die sogenannte Erreichbarkeitsanalyse (siehe Tabelle 2) kann man nun die

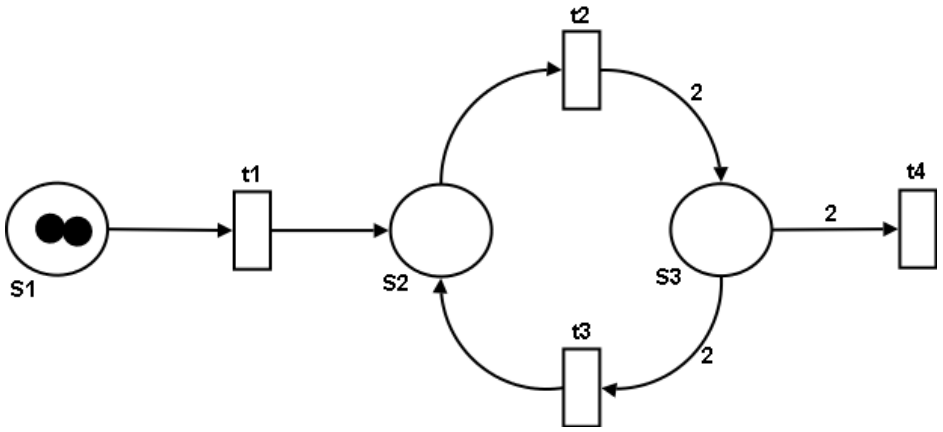

Abbildung 11: *Beispiel eines Petri-Netzes*

Menge der Zustände festellen, die ein Netz annehmen kann, sofern dieses beschränkt ist. Das Ergebnis der Erreichbarkeitsanalyse dient hierbei nicht als Darstellungsform eines Netzes, denn nicht jedes Netz lässt sich durch sie darstellen. Es zeigt die Menge der Zustände, die ein bestimmtes Petri-Netz annehmen kann. In jedem Zustand besitzt das Netz nun eine, oder mehrere schaltbereite Transitionen, die einen Wechsel in einen anderen Zustand ($M[t\rangle Mt$) auslösen können.
Des weiteren existiert die Methode der Überdeckungsanalyse, mit der sich die Beschränktheit eines Netzes feststellen lässt. Auf diese wird hier aber nicht weiter eingegangen.

Zustand	S1	S2	S3	Schaltungen
M_0	2	0	0	$t1 \rightarrow M_1$
M_1	1	1	0	$t1 \rightarrow M_2, t2 \rightarrow M_3$
M_2	0	2	0	$t2 \rightarrow M_4$
M_3	1	0	2	$t1 \rightarrow M_4, t3 \rightarrow M_1, t4 \rightarrow M_5$
M_4	0	1	2	$t3 \rightarrow M_2, t2 \rightarrow M_6, t4 \rightarrow M_7$
M_5	1	0	0	$t1 \rightarrow M_7$
M_6	0	0	4	$t3 \rightarrow M_4, t4 \rightarrow M_8$
M_7	0	1	0	$t2 \rightarrow M_8$
M_8	0	0	2	$t3 \rightarrow M_7, t4 \rightarrow M_9$
M_9	0	0	0	

Tabelle 2: *Ergebnis der Erreichbarkeitsanalyse zum Petri-Netzes in Abbildung 11*

[30] Ein Zustand des Petri-Netzes ist hier eine aus M_0 durch schalten erzeugte Vertilung M der Marken im Netz. In den Timer-Netzen besitzt ein Zustand zudem einen Zeitstempel.

Eine leicht veränderte Variante des Petri-Netz-Modells, die auch im *CS*-Framework zum Einsatz kommt, sind die Kapazitätsnetze. Ein Platz innerhalb eines *CS*-Petri-Netzes hat nun eine gewisse Markenkapazität. Ist diese erreicht, kann eine vorgehende Transition nicht mehr schalten, sondern muss warten bis wieder genügend Freiraum für neue Marken verfügbar ist. Kapazitätsnetze sind auch als gewöhnliche Petri-Netze darstellbar.

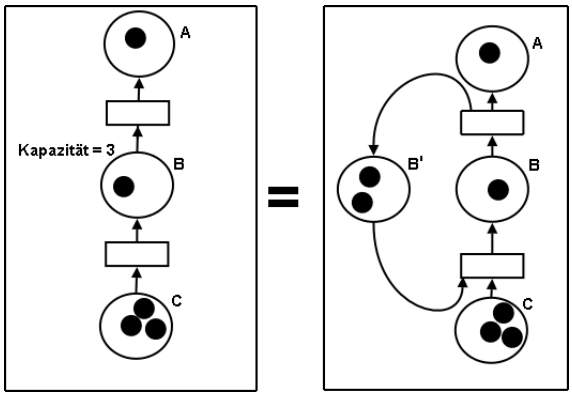

Abbildung 12: *Kapazitätsnetze, eine Unterform gewöhnlicher Petri-Netze*

Abbildung 12 zeigt anhand eines Beispiels, wie man ein Kapazitätsnetz als gewöhnliches Petri-Netz darstellen kann. Für jeden Platz (z.B. Platz B) mit einer Kapazität (z.B.: Kapazität von B = 3) wird ein zusätzlicher Platz (im Beispiel: B') eingeführt.
Die Anzahl der Marken auf B' wird für die Startverteilung M_0 so gewählt, dass
$M_0(B)+M_0(B')$ = Kapazität von B ist.
Für jede Transition $x \in \bullet B$ wird eine Kante (B', x) eingeführt mit W(B', x)=W(x, B), und für jede Transition $y \in B\bullet$ wird eine Kante (y, B') eingeführt mit W(y, B')=W(B, y).
Ist B' leer, so können die Vorbereichs-Transitionen nicht mehr schalten und somit keine Marken mehr zu B gelangen. Sobald eine Nachbereichs-Transition von B schaltet, entstehen erneut Marken auf B' und somit Platz für neue Marken auf B.

Die jetzige Variante der Petri-Netze erlaubt es zwar, klare Kausalitäten aufzustellen, garantiert jedoch nicht, dass Netze in endlicher Zeit durchlaufen werden. Die Netzform ist zu einfach aufgebaut, um im *CS*-Framework zum Einsatz zu kommen. Um ein reales System in Echtzeit von einem Petri-Netz steuern zu lassen, bedarf es zeitlicher Strukturen und Komponenten, welche wir z.B. in den Timer-Netzen vorfinden.

3.2.4 Timer-Netze

Um auch zeitliche Zusammenhänge darzustellen, entwickelte man das Modell der sogenannten Timer-Netze. Wie auch bei den Petri-Netzen gibt es hier viele verschiedene Verwirklichungen. In der Arbeit werden nur spezielle Teile der von Bernd Baumgarten definierten Timer-Netze verwendet (siehe [Baumgarten2006]).
In einem Timer-Netz kann man jede Transition mit zeitlichen Bedingungen verknüpfen. Man definiert bestimmte Timer, die beim Netzstart loslaufen und spricht diese dann über die Transitionen an. Zum Beispiel meint man mit $[t_1 > 10]$: diese Transition schaltet erst, wenn auf Timer t_1 mehr als 10 Sekunden vergangen sind. Auch kann man Timern mit Befehlen wie $(t_1 = 5)$ bestimmte Werte zuweisen.
Als zusätzliche Kontrollstruktur existieren hier drei verschiedene Transitions-Typen, MAY, MUST und ASAP (as soon as possible).
Die MAY-Transition bezeichnet die herkömmliche Petri-Netz-Transition, welche im ersten Abschnitt vorgestellt wurde. MUST-Transitionen schalten spätestens zum angegebenen Zeitpunkt, sofern alle Marken-Kriterien erfüllt sind. ASAP-Transitionen hingegen schalten direkt, sobald alle erforderlichen Marken- und Zeit-Kriterien erfüllt sind.

Timer-Netze stellen genügend Optionen zur Modellierung reale Systeme zur Verfügung, daher sind diese die erste Wahl zur Verwirklichung im *CS*-Framework. Es existieren zwar noch höhere Netzformen, wie z.B. farbige Netze, und Netze, welche mit beliebigen Datentypen arbeiten, doch die Beschreibung und Implementierung dieser höheren Netzformen würde den Rahmen dieser Diplomarbeit sprengen.
Die Timer-Netze, die im *CS*-Framework zum Einsatz kommen, benötigen nur einen Transitions-Typ, die ASAP-Transition. Beliebige zeitliche Verzögerungen, wie bei MAY und MUST sind im realen System eher störend als nützlich, und tragen höchstenz zur Verwirrung des Benutzers bei.

3.3 Anforderungen

Folgende Anforderungen gelten sowohl für die Objekt-Netze als auch für die Petri-Netze:

Möglichkeit der hierachischen Strukturierung: Das Aufrufen von Unternetzen, welche als einfache sub-VI's dargestellt werden, sollte ähnlich wie in LabVIEW selbst möglich sein.

Mehrmaliges Instanziieren eines Netzes: Das Netz selbst sollte als Klasse existieren. So wäre die Instanziierung weiterer Netze desselben Types kein Problem mehr. Weiterhin muss hier auf die Namensgebung des Netzes und aller Unter-Objekte geachtet werden, da im *CS*-Framework jeder Objektname eindeutig ist. Eine strikte Konvention ist hier zu schaffen.

Programmatische Ansteuerung der Netze: Die Kontrollstrukturen der Objekt-, sowie der Petri-Netze sollte man nicht nur von Hand bedienen können. Jeder Knopf der Netzkontrolle ist so zu gestalten, dass er auch durch einen programmatischen Befehl angesprochen werden kann.

Benutzerfreundlichkeit: Der Benutzer eines Objekt- oder Petri-Netzes soll nicht gezwungen sein, sich breites Vorwissen über die Netze anzueignen. Die Gestaltung eigener Anwendungen sollte möglichst intuitiv und ohne großen programmatischen Aufwand möglich sein. Zu diesem Zweck erscheint es sinnvoll, einige Beispiele zu implementieren, an denen sich der Benutzer orientieren kann.

Design: Sowohl Objekt-, wie auch Petri-Netze sind ausführbare Designdokumente. Der User hat somit die Möglichkeit, auf einen Blick die Funktionsweise eines Systems zu erfassen und kann diese gleichzeitig an seine Bedürfnis anpassen. Das heisst wiederum, alle Komponenten der Objekt- und Petri-Netze sollten leicht verständlch und intuitiv in der Benutzung sein.

Kontrollstrukturen: In jedem Netz muss eine Kontrollstruktur existieren, welche es dem Benutzer erlaubt, das Netz zu starten, es für Modifikationen anzuhalten, oder es herunter zu fahren. Es erscheint als sinnvoll diese Strukturen ins Haupt-Netz einzubetten, da sie damit automatisch in jedem Netz vorhanden wären.

3.3.1 Spezielle Anforderungen der Objekt-Netze

Wiederverwendbarkeit: Objekte, welche einmal in einem Objekt-Netz verwendet wurden, sollten ohne große Umstände erneut in einem anderen Objekt-Netz funktionieren. Eine klare Schnittstelle ist zu schaffen, die für jedes Objekt eindeutig ist, und an welcher alle ein- und ausgehenden Ports direkt abgelesen werden können.

Das Einbinden existierender Klassen: Es sollte eine Möglichkeit geben, schon existierende Klassen im Objektnetz aufzunehmen. Die Mehrfach-Vererbung im CS- Kontrollsystem bietet hier einen brauchbaren Ansatz, welchen man testen und weiter verfolgen sollte. Jede Klasse, die im Objekt-Netz verwendet wird, sollte man nach wie vor auch normal starten können.

Überwachung der Netzobjekte: Der *CS*-Watchdog bietet die Möglichkeit den Status aller *CS*-Objekte auf einem System programmatisch abzufragen. Nun wird eine Instanz benötigt, welche auf diese Informationen reagiert. Diese Instanz sollte Objekte beim Netzstart initialisieren, sie im Falle einer Störung neu starten und sie beim Beenden der Anwendung ordnungsgemäß vernichten.

Flexibilität der Port-Datentypen: Das Objekt-Netz sollte standardmäßig alle DIM-Datentypen unterstützen, und für diese VI's zum Entpacken bereit stellen. Des weiteren sollte der Benutzer aber auch die Möglichkeit haben, beliebige Datentypen zu benutzen. Die Visualisierung dieser exotischen Datenformate geschieht dann in einem VI, welches der Benutzer selbst schreibt.

3.3.2 Spezielle Anforderungen der Petri-Netze

Auslösen von Aktionen beim Schalten: Petri-Netze allein sind ein rein theoretisches Konstrukt, sie haben keine festen Beziehungen zur realen Welt. Genau das wird aber hier gefordert. Um Kontakt mit der realen Welt herzustellen, benötigt man Aktionen, die durch das Netz ausgelöst werden. Es wäre z.B. sinnvoll, jedem Platz eine Eintritts-, eine Austritts-Aktion und eine periodische Aktion zuzuweisen. In diesen Aktionen kann dann die Markenzahl des Platzes vom Benutzer variabel verändert werden, um diese an äussere Gegebenheiten anzupassen. Des weiteren könnte man beim Schalten einer Transition ähnliche Aktionen auslösen.

Plätze und Transitionen als Klassen: Um sich einiges an Arbeit zu sparen, erscheint es zweckdienlich, Plätze sowie Transitionen als CS-BaseProcess Klassen zu erstellen. So könnten diese den schon implementierten Ereignismechanismus, sowie auch viele andere Vorteile nutzen.
Der Aufbau neuer Programmier-Konstrukte für Plätze und Transitionen würde Arbeitschritte verlangen, die im *CS*-Framework schon lange abgehandelt wurden, und somit unnötig Entwicklungszeit verschwenden.

Kapazitätsgrenze von Plätzen: Wie in Abschnitt 3.2.3 schon erwähnt, benötigt man in der programmatischen Modellierung der Plätze eine maximale Markenkapazität. Diese wirkt sich nicht nur auf Plätze, sondern noch entscheidender auf Transitionen aus. Eine Transition muss nicht nur prüfen, ob alle Vorgänger-Plätze genug Marken haben, sie ist jetzt auch dazu gezwungen die Nachfolger-Plätze zu betrachten. Ist dort nicht genügend Platz zum Lagern der Marken, kann nicht geschaltet werden.

Reservierungsmechanismus: Eine Transition muss sich erst vergewissern, dass alle Kriterien, die sie zum Schalten benötigt, erfüllt sind. Danach kann sie den Schaltvorgang selbst einleiten. Um jedoch sicher zu gehen dass zwischen dem Überprüfen der Schaltkriterien und dem Schaltvorgang selbst keine andere Transition die benötigten Marken verbraucht, ist es erforderlich alle zugehörigen Plätze exklusiv für die Dauer der Schaltaktion zu reservieren.

Nutzung existierender Klassen: Wie auch bei den Objekt-Netzen soll es möglich sein, schon existierende Klassen ins Netz einzubinden. Hier erscheint es sinnvoll, das Problem mit Hilfe der Vererbung anzugehen. Erbt eine existierende Klasse von der Platzklasse, so kann diese umgehend im Petri-Netz genutzt werden.

Überwachung der Netzobjekte: Ähnlich wie beim Objekt-Netz sollte es beim Petri-Netz eine Instanz geben, welche die Netzobjekte über den CS-Watchdog startet, überwacht und zerstört. Allerdings muss hier darauf geachtet werden, nur Transitionen im Störungsfall neu zu starten. Das Neu-Starten von Plätzen könnte zu einer unkontrollierten Neuverteilung der Marken führen, und somit ungewollte Ereignisse auslösen.

Wiederverwendbarkeit: Plätze sowie Transitionen und andere Netzkomponenten sind so zu erstellen, dass eine erneute Verwendung in anderen Netzen problemlos erfolgen kann. Durch automatische Initialisierung der Objekte, ohne Benutzung der CS-Access Datenbank sollte dies einfach zu erreichen sein.

Zeitliche Komponenten: Wie schon in Abschnitt 3.2.4 angedeutet, benötigt die praxisorientierte Umsetzung der Petri-Netze eine zeitliche Komponente. Eine Timer-Klasse ist zu implementieren, welche das Zählen von Sekunden übernimmt. Ergänzend sollten Transitionen nun einen Eingang besitzen, an dem man Timer-Kriterien festlegen kann. Nun erscheint es sinnvoll, der Kontrollstruktur des Petri-Netzes einen Pause-Knopf hinzuzufügen, der das Netz für unbestimmte Zeit anhält.

Kommunikation zwischen Plätzen und Transitionen Sobald sich die Markenzahl eines Platzes ändert, sollte der Platz die Änderung allen angeschlossenen Transitionen mitteilen. Die Nutzung von DIM-Services erleichter dies sehr. So genügt es, den Transitionen anfangs die richtigen Plätze zuzuweisen. Sobald nun ein Platz seine Markenzahl veröffentlicht, erfahren alle Transitionen, die den DIM-Service dieses Platzes empfangen, die neue Markenzahl automatisch und können direkt auf die neue Situation reagieren.
Des weiteren steht die Kommunikation über DIM-Commands zur Verfügung. DIM-Commands könnten beispielsweise genutzt werden, um gezielt Plätze anzusprechen. Ein sauberes Zusammenspiel von Kommunikation und Reservierungs-Mechanismus ist hier erforderlich.

4 Entwurf und Realisierung

4.1 Objekt-Netze

Die hier beschriebene Version der Objekt-Netze ging aus einigen älternen Versionen hervor. Sie kam durch zahlreiche Tests und Rücksprachen mit Holger Brand und Dietrich Beck zustande, welche an vielen Stellen ihre Ideen mit einbrachten.

4.1.1 Klassenstruktur

Wie auf Abbildung 13 zu erkennen ist, besitzt das *CS*-Framework eine relativ flache Klassenhierarchie. Alle weiss hinterlegten Rechtecke zeigen *CS*-Klassen, welche standardmäßig im CS existieren. Die grau hinterlegten Klassen gehören zu den eigentlichen Objekt-Netzen, welche im Rahmen dieser Arbeit implementiert wurden. Grüne (dunkelgraue) Rechtecke zeigen, welche Klassen der Benutzer der Netze zusätzlich erstellen kann.

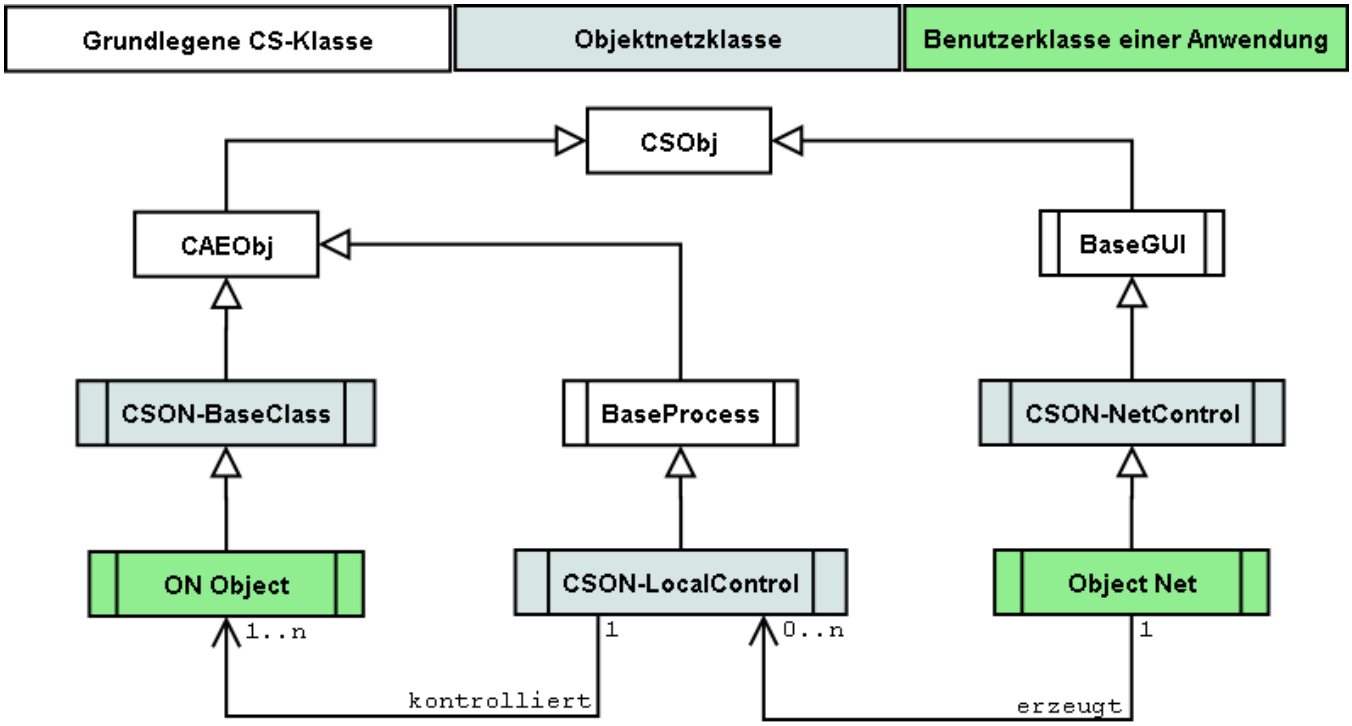

Abbildung 13: *Vererbungshierarchie der CS-Objekt-Netze*

In der Klassentabelle (Tabelle 3) sind die Funktionen der einzelnen Klassen im Detail erläutert.

Klasse	Verwendung
CSON-NetControl	Will man ein eigenes Objekt-Netz konstruieren, so erzeugt man dieses durch Vererbung von der CSON-NetControl Klasse. Man hat nun die Möglichkeit, das eigene Netz in der Methode "CLASS.MainObjectNet.vit" zu verwirklichen und dieses über die bereitgestellte Kontrollstruktur zu steuern.
CSON-BaseClass	Nur Objekte, welche von dieser Klasse geerbt haben, können im Objekt-Netz gestartet werden. Objekte dieses Typs tragen im Attribut die Portnamen und Datentypen ihrer Kommunikationspartner. Werden die so bereitgestellten Ports vom Objekt genutzt, so kann man den Datenfluss im Objekt-Netz visualisieren.
CSON-LocalControl	Sobald ein Objekt-Netz gestartet wird, wird auf jedem benötigten CS-System eine Instanz der CSON-LocalControl Klasse erstellt. Der CSON-LocalControler startet und überwacht nun alle zu diesem System gehörigen ON-Objekte. Er dient als lokale Kontrollinstanz, welche auch funktioniert, wenn die Netzwerkverbindung verloren geht.

Tabelle 3: *Verwendung der Objekt-Netz Klassen*

4.1.2 Benutzung der Objekt-Netze

Ein Objekt-Netz besteht hauptsächlich aus Methoden, welche Objekte instanziieren (diese werden im weiteren "Launch.vi´s" genannt) und Methoden, welche den Datenfluss zwischen Objekten grafisch darstellen (sog. "Observer.vi´s"). Will man ein *CS*-Objekt in einem Objekt-Netz verwenden, so lässt man dessen Klasse von der Objekt-Netz Basisklasse erben. Somit erhält dieses Objekt ein eigenes Launch.vi. Jedes Launch.vi hat eingehende und ausgehende Ports, welche man nun an die spezifische Klasse anpasst. Durch eine verständliche Namensvergabe der Ports und durch richtige Zuweisung der Datentypen erhöht man die Wiederverwendbarkeit der Klasse und stellt gleichzeitig ein festes Interface zur Verwendung innerhalb der Objekt-Netze her.

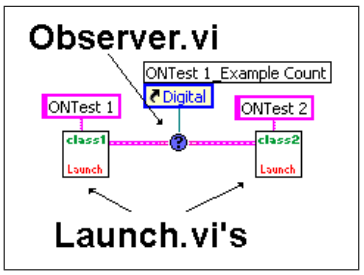

Abbildung 14: *Lauch.vi's und Observer*

Um im Objekt-Netz nun ein konkretes Objekt einer Klasse auf einem bestimmten *CS*-System zu erzeugen, platziert man das Launch.vi ins Netz und verbindet dessen Eingänge "Object name" und "SystemID" mit den gewünschten Namen.
Man hat nun die Möglichkeit, die Ports der Launch.vi´s untereinander zu verbinden (siehe Abbildung links). Die Verbindung gibt an, dass die beiden Klassen miteinander kommunizieren. Dabei stehen ausgehende Ports für Datenquellen, und eingehende für Datensenken. Will man den Datenfluss visualisieren, so platziert man ein Observer.vi zwischen die beiden Ports der Launch.vi's. Auch hier ist auf die Wahl des passenden Datentyp zu achten.

Um ein neues Netz zu konstruieren, erzeugt der Benutzer eine Klasse durch Vererbung von der CSON-NetControl Klasse. In die durch den Vererbungsprozess erzeugte Methode "MainObjectNet.vit" kann man nach Belieben ein Objekt-Netz einfügen. Erstellt man jetzt ein Objekt dieser Klasse, wird automatisch das dort konstruierte Objekt-Netz ausgeführt, welches immer den gleichen Namen wie das CS-Objekt selbst trägt.
Jedes dieser so erzeugten Objekt-Netze besitzt eine Kontrollstruktur (Abbildung 15), die es dem Benutzer ermöglicht, das Netz zu steuern. Um unabsichtliches Auslösen der Kontrollen zu vermeiden, erscheint nach jedem Knopfdruck ein Dialog, in der die Aktion noch einmal zu bestätigen ist. Das in Abbildung 15 gezeigte Beispiel gehört zu einer Heizungssteuerung und wird im Abschnitt 4.4 näher erläutert. Die einzelnen Kontrollen lassen sich auch programmatisch ansteuern. Ausführliche Informationen zur programmatischen Ansteuerung befinden sich im Abschnitt 4.3.4.

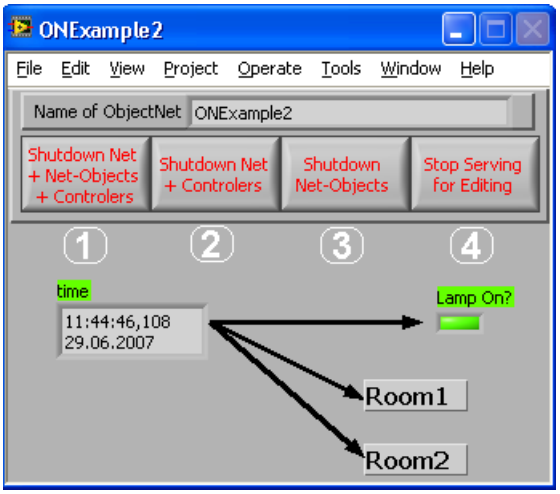

Abbildung 15: *Objekt-Netz Kontrollstruktur*

Bedeutung der einzelnen Knöpfe:

1. Drückt man Knopf 1, so werden alle ON-Objekte, alle ON-LocalControler und das Netz selbst zerstört.
2. Durch Knopf 2 zerstört man nur die Kontrollstrukturen, also das Netz selbst und die ON-LocalControler.
3. Knopf 3 verursacht eine Zerstörung aller ON-Objekte. Die Kontrollstrukturen bleiben erhalten.
4. Drückt man Knopf 4, so werden keine Objekte zerstört. Einzig das Objektnetz selbst wird angehalten, um dem Benutzer die Möglichkeit zu geben, dieses während der Laufzeit zu modifizieren.

4.1.3 Zuweisen von Ports

Um zu gewährleisten, dass nur zueinander passende Datenquellen und Senken miteinander verbunden werden können, besteht die Möglichkeit, den ein- und ausgehenden Ports verschiedene Datentypen zuzuweisen. Versucht man zwei ungleiche Ports zu verknüpfen, so erzeugt man einen Fehler in LabVIEW, das Netz ist somit nicht mehr ausführbar.

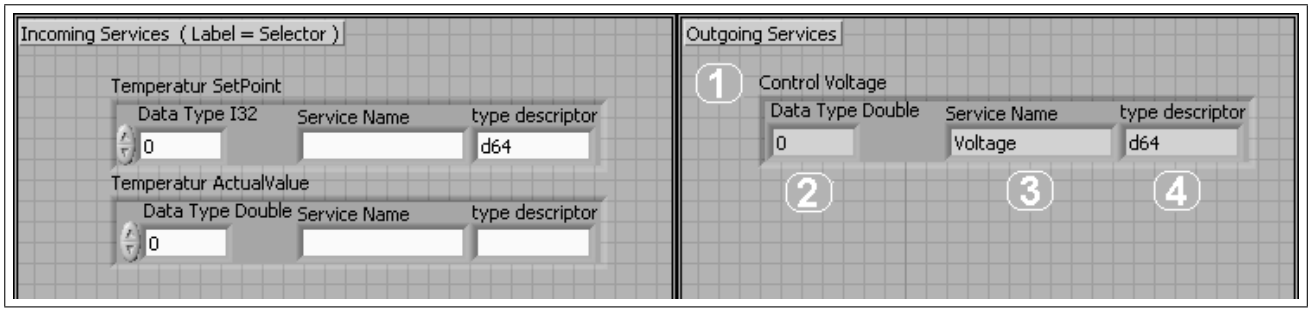

Abbildung 16: *Portbelegung eines Launch.vi´s*

Abbildung 16 zeigt einige präparierte Ports. Wählbare Parameter sind hierbei:

1. Name: Der Name des Kontrollelements bei einem eingehenden Service (Incoming Services) ist nach dem dazu passenden Selector in den "ProcCases" der Klasse zu wählen.
2. Datentyp: Der hier gewählte Datentyp ist an den Service-Datentyp anzupassen. So können nur Methoden vom gleichen Datentyp mit diesen Ein- und Ausgängen verbunden werden.
3. Service Name: Bei Ausgängen ist hier der Name des Services einzutragen. Eingänge beziehen diese Information automatisch.
4. Type descriptor: Der Datentyp, als Zeichenkette im DIM-Format, kann hier angegeben werden.

4.1.4 Grundlegende Netzstruktur

Die in der Abbildung 13 modellierten Beziehungen zeigen die grundlegene Kontrollhierarchie im Netz. Das Netz selbst (Object Net) erzeugt auf jedem benötigten *CS*-System ein CSON-LocalControl Objekt. Dieses startet und kontrolliert nun alle zu diesem System gehörigen ON-Objekte. Die Systemstruktur enstand aus dem Gedanken heraus, die Kontrolle über Netzobjekte auch dann zu behalten, wenn das Netzwerk teilweise oder ganz ausfällt. Außerdem ist bei dieser Methode der Objekt-Zugriff direkt über Referenz möglich, wodurch sich perfomantere Methoden zum Zerstören und Erzeugen von Objekten eröffnen.
Beim Starten des Netzes werden alle relevanten Informationen, Verbindungen zwischen Objekten, Namen, Subnetze etc. einmal eingelesen und im Objektattribut des Netzes selbst gespeichert. Diese Daten werden dann über einen DIM-Service veröffentlicht, auf den alle CSON-LocalControler reagieren. Die CSON-LocalControler, welche auch vom Netz selbst gestartet werden, reagieren, sobald sie die Netzdaten erhalten haben. Sie extrahieren daraus alle für ihr CS-System relevanten Datensätze und starten die erforderlichen Objekte. Abbildung 31 im Anhang zeigt ein Flussdiagramm, welches die Zusammenhänge noch einmal verdeutlicht.

4.1.5 Datenentschlüsselung und Visualisierung

Zum Entpacken der empfangenen Datenpakete dient ein etwas ungewöhnlicher Mechanismus. Beim Start des Netzes schreibt jedes Observer.vi die jeweils relevanten Daten[31] ins Attribut des Netzes und ein einziger DIM-Listener wird erzeugt. Dieser benutzt nun die Daten im Attribut des Netzes und abonniert alle dort befindlichen Services .
Wird ein DIM-Service aktualisiert ruft der DIM-Listener dynamisch das passende Observer.vi auf (von dem aus er gestartet wurde), um die eingehenden Daten zu entpacken und diese auf den angeschlossenen Indikator zu schreiben. Die Entpack-Routine und das Schreiben der Daten auf das Frontpanel-Anzeigeelement sind dabei im Observer.vi gespeichert(Siehe auch Abbildung 17).

[31] Ein Datensatz aus diesem Array besteht aus Servicename, Indikatorreferenz und Dateipfad.

Beim Auftreten eines Fehlers wird der Hintergrund des Frontpanel-Anzeigeelements farbig markiert, um Informationen über die Qualität der Daten zu erhalten. Die Bedeutung der einzelnen Farben ist wie folgt festgelegt:

rot:	Kein DIM-Listener vorhanden/es ist kein Service definiert.
blau:	Keine Verbindung zum abonnierten Service.
gelb:	Die Daten des Dienstes sind alt/korrupt.
grün:	Es sind keine Fehler aufgetreten.

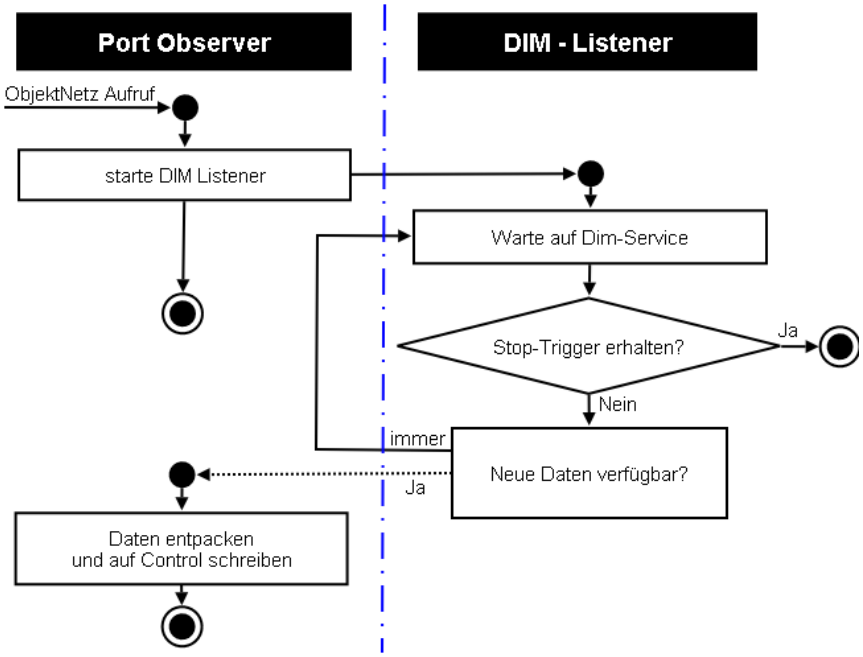

Abbildung 17: *Der ON-Data Observer Mechanismus*

Das Observer.vi wird also mehrmals aufgerufen, und dies für zwei völlig unterschiedliche Aufgaben. Das mag anfangs merkwürdig erscheinen, macht jedoch insofern Sinn, als die Zuordnung des Port-Datentyps und das Entpacken eingehender Daten im selben VI, und damit zentral passiert. Da der Datentyp beliebig wählbar sein soll, ist ohnehin eine Methode notwendig, welche die Daten entpacken muss.
Würde man die Entpack-Methode unabhängig vom Observer.vi speichern, so müsste man im Objekt-Netz den Pfad zu dieser Methode mit angeben. Ein solches Vorgehen entspräche allerdings nicht der Philosophie der Objekt-Netze. Die Nutzbarkeit als Designdokument würde beträchtlich darunter leiden und die erforderlichen Vorkenntnisse zur Implementierung von Objekt-Netzen steigern.
Aus den hier aufgeführten Gründen wuchs der Entschluss, das Observer.vi zweifach zu nutzen. Auch wenn dies nicht der üblichen LabVIEW-Programmierung entspricht, so überwiegen doch die Vorteile, die man dadurch gewinnt.

4.1.6 Der Watchdog-Mechanismus

Benutzt man im Objekt-Netz ein Objekt, welches von der BaseProcess Klasse geerbt hat, so wird dieses automatisch durch den *CS*-Watchdog überwacht. Die CSON-LocalControl Klasse nutzt dies aus, mit Hilfe des CS-Watchdog-Mechanismuses erfährt er den Zustand aller BaseProcess-Objekte. Wird nun bei einem ON-Objekt ein fehlerhafter Zustand angezeigt, so versucht der CSON-LocalControler das betreffende Objekt mit der “unload”-Funktion zu löschen. Sollte das Objekt darauf nicht reagieren, wird als letzte Möglichkeit die “kill Object”-Funktion angewendet. Sobald nun das ON-Objekt erfolgreich zerstört wurde, erstellt der CSON-LocalControl dieses neu, mit Hilfe von “Create Object”. Damit kann die Anwendung des Benutzers reibungsfrei weiter agieren.

4.1.7 Wiederverwendung bestehender Objekte

Objekte, welche einmal in einem Objekt-Netz verwendet wurden, können dank der einheitlichen Schnittstelle, des Launch.vi's, sofort auch in anderen Objekt-Netzen eingesetzt werden. Ein korrekt implementiertes Launch.vi verfügt über verschiedene Ein- und Ausgangsports, welche der jeweiligen Funktion entsprechende Namen tragen. Eine Festlegung des Datentyps auf jedem Port, verhindert die falsche Nutzung und beugt Missverständnissen vor. Wie beim Lego-Steck Prinzip kann man nun die existierenden Klassen zusammenfügen und wiederverwenden.

4.2 Petri-Netze

4.2.1 Klassenstruktur

Die Klassenstruktur der Petri-Netze (siehe Abbildung 18) ähnelt der Strukur der Objekt-Netze. Das Netz selbst und der CSON-LocalControler wurden zu großen Teilen beibehalten. Nicht so die restlichen Netzobjekte. War man bei den Objekt-Netze gezwungen, von einer ON-Base Klasse zu erben, kann man hier die schon zur Verfügung stehenden Klassen benutzen. Weiterhin kann man bei Bedarf durch Vererbung von der CSPN-Place Klasse neue PN-Place Klassen erzeugen, und somit individuelle Netzobjekte kreieren.

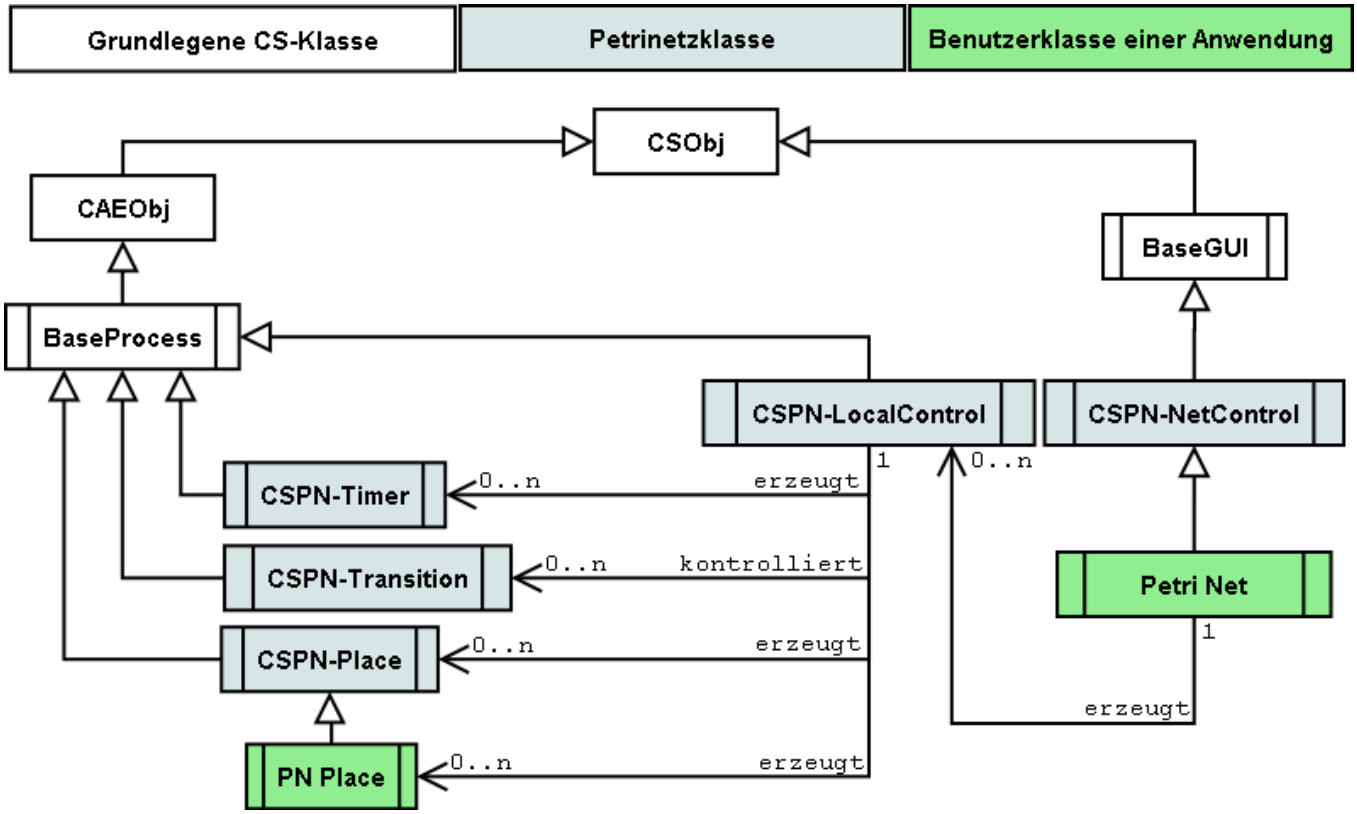

Abbildung 18: *Vererbungshierarchie der CS-Petri-Netze*

In der Klassentabelle (Tabelle 4) sind die Funktionen der einzelnen Klassen im Detail erläutert.

4.2.2 Grundlegene Unterschiede zu den Objekt-Netzen

Die Aufgaben eines *CS*-Petri-Netzes sind völlig anderer Natur, als die eines *CS*-Objekt-Netzes. Dort ging es darum, beliebige Objekte zu starten, zu kontrollieren und deren Kommunikation zu observieren. Im *CS*-Petri-Netz hingegen ist die Kontrolle und die Kommunikaton der Objekte nur zweitrangig. Zwar werden Transitionen im Falle einer Störung neu gestartet, das Hauptaugenmerk liegt hier jedoch auf der Deterministik des Petri-Netzes.
Des weiteren werden keine beliebigen Datentypen, sondern nur Petri-Netz Marken visualisiert, die Verknüpfung des Netzes mit der Außenwelt geschieht über Aktions-Aufrufe an den Plätzen und Transitionen. Vom Neustarten der Plätze wird abgesehen, da damit die Markenzahl im Netz gravierend verfälscht werden könnte, was unvorhersehbare Folgen hätte.

Klasse	Verwendung
CSPN-NetControl	Will man ein eigenes Petri-Netz konstruieren, so erzeugt man dieses durch Vererbung von der CSPN-NetControl Klasse. Man hat nun die Möglichkeit, das eigene Netz in der Methode "CLASS.MainPetriNet.vit" zu verwirklichen und dieses über die bereitgestellte Kontrollstruktur zu steuern.
CSPN-LocalControl	Sobald ein Petri-Netz gestartet wird, wird auf jedem benötigten *CS*-System eine Instanz der CSPN-LocalControl Klasse erstellt. Der CSPN-LocalControler startet und überwacht nun alle zu diesem System gehörigen Petri-Netz Objekte. Er dient als lokale Kontrollinstanz, welche auch funktioniert, wenn die Netzwerkverbindung verloren geht.
CSPN-Place	Die zur Konstruktion eines *CS*-Petri-Netzes benötigten Plätze werden von dieser Klasse bereitgestellt. Um einen vollwertigen Platz in einem Petri-Netz zu erhalten, genügt es, das "CSPN-Place.Launch.vi" in diesem Netz zu platzieren. Will man einen Platz mit zusätzlicher Funktionalität, besteht auch die Möglichkeit von CSPN-Place zu erben, und somit eine eigene Platz-Klasse zu schaffen.
CSPN-Transition	Die CSPN-Transition Klasse ermöglicht das Nutzen von Transitionen im *CS*-Petri-Netz . Ähnlich wie bei der CSPN-Place Klasse genügt es, das "CSPN-Transition.Launch.vi" in das eigene Petri-Netz zu platzieren. Die Möglichkeit, von der CSPN-Transition Klasse zu erben, ist nicht vorgesehen.
CSPN-Timer	Die CSPN-Timer Klasse stellt dem *CS*-Petri-Netz zeitliche Komponenten zur Verfügung. Platziert man ein "CSPN-Timer.Launch.vi" im *CS*-Petri-Netz , so erhält man einen Sekundenzähler (lässt sich später auf kleinere Zeitintervalle optimieren), welcher zusammen mit dem Netz startet. Der Zähler hält an, sobald das Netz in den Pause-Modus wechselt. Er ermöglicht den Transitionen, unter temporalen Bedingungen zu schalten. Auch hier ist keine Vererbung vorgesehen

Tabelle 4: *Verwendung der Petri-Netz Klassen*

Genauso wie bei den Objekt-Netzen besitzen die Petri-Netze eine grundlegene Benutzer-Kontrollstruktur (Abbildung 19). Wieder erscheint nach jedem Knopfdruck ein Dialog, in der die Aktion noch einmal zu bestätigen ist. Das in Abbildung 19 gezeigte Beispiel gehört zum Petri-Netz "Dinner der Philosophen" und wird im Abschnitt 4.4 näher erläutert.

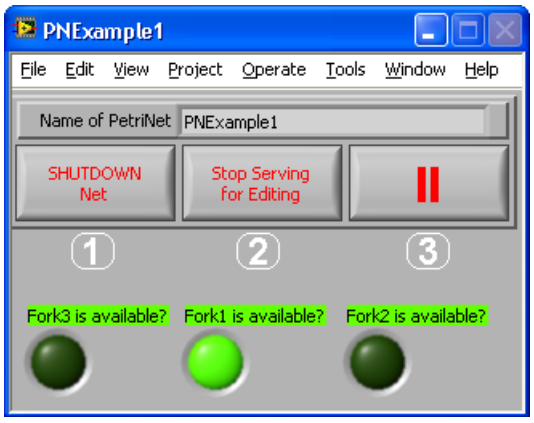

Abbildung 19: *Petri-Netz Kontrollstruktur*

Bedeutung der einzelnen Knöpfe:

1. Drückt man Knopf 1, so werden alle Netzobjekte (Plätze, Transitionen und Timer), alle PN-LocalControler und das Netz selbst zerstört.
2. Knopf 2 verursacht eine Zerstörung aller Netzobjekte, die Kontrollstrukturen bleiben dabei jedoch erhalten.
3. Durch Knopf 3 löst man den Pause-Modus des Netzes aus. Dadurch werden alle Plätze vom Netz selbst reserviert und alle Timer erhalten einen Pause-Befehl. Erneutes Betätigen des Knopfes beendet die Pause.

4.2.3 Schaltbedingungen der Transition

Ob eine Transition schalten kann oder nicht, hängt von zahlreichen Kriterien ab, die alle erfüllt sein müssen. Sobald die Transition eine Zeit- oder Platzaktualisierung via DIM empfängt, kann sie diese Kriterien prüfen. Es wird nun ermittelt, ob genügend Marken auf den Vorgänger-Plätzen liegen (Kantengewichtungen), ob genügend freier Platz auf den Nachfolger-Plätzen zur Verfügung steht (Kantengewichtungen) und ob die zeitlichen Kriterien, soweit angeschlossen, erfüllt sind. Abbildung 21 zeigt die Verwirklichung der eben beschriebenen Logik im dazugehörigen Blockdiagramm.

Des weiteren wird ein Spezialfall, die "kleinste Schleife" geprüft, und, falls notwendig, behandelt. Wie in Abbildung 20 zu sehen ist, geht es bei der "kleinsten Schleife" um einen Platz, der sowohl Vorgänger, wie auch Nachfolger einer Transition ist. Ob nun genug Platz für die Nachfolger-Marken zur Verfügung steht, hängt hier auch davon ab, wieviele Marken der Platz als Vorgänger fortgibt. Die Behandlung dieser Spezialfalles wird in einem seperaten VI vorgenommen, welches ein Sub-VI der eigentlichen Schaltprüfung ist.
In den gewöhnlichen Petri-Netzen bezeichnet man diese Schaltregel als Schleifen-tolerante starke Schaltregel. Abbildung 20 zeigt die Verwendung einer "kleinsten Schleife" als Schaltvorgangs-Zähler.

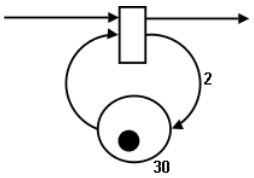

Abbildung 20: *kleinste Schleife*

4.2.4 Der Platzreservierungs-Mechanismus

Wenn zwei Transitionen gleichzeitig versuchen, auf einen Platz zuzugreifen, welcher genug Marken für eine der beiden Transitionen besitzt, kann dies zu fehlerhaften Schaltungen führen.
Daher erscheint es sinnvoll, einen Mechanismus zu implementieren, welcher dafür sorgt, dass ein Platz immer nur einer Transition zum Schalten bereit steht. Des weiteren wird diese Technik auch als Reservierungsmechanismus, Lock-Mechanismus oder Semaphore[32] bezeichnet.
Erfüllt eine Transition nun alle Schaltbedingungen, kann sie die erforderlichen Plätze reservieren, indem sie das entsprechende Kommando an diese sendet. Sofern keiner der Plätze schon anderweitig reserviert war, kann die Schaltprozedur fortgesetzt werden. Die so reservierten Plätze nehmen jetzt nur noch Befehle dieser Transition entgegen, bis die Transition sich entscheidet, die Plätze wieder frei zu geben. In dieser Zeit kann die Transition nun ihren Schaltvorgang erledigen, ohne von anderen Transitionen dabei gestört zu werden.

4.2.5 Die Funktion der Timer-Klasse

Die Timer-Klasse gibt dem Benutzer die Möglichkeit zeitliche Steuerelemente ins Netz einzufügen. Ein Objekt der Timer-Klasse zählt die Zeit seit dem Netzstart in Sekunden. Das Timer-Objekt merkt sich den Zeitpunkt des Starts, vergleicht diesen jede Sekunde mit der Jetzt-Zeit und veröffentlicht sein Ergebnis über einen DIM-Service. Drückt der Benutzer des Petri-Netzes den Pause-Knopf, so erhalten auch alle Timer-Objekte den Pause-Befehl. Die Zeit-Veröffentlichung wird gestoppt, bis ein "continue"-Befehl eingeht. Nun wird der neue Start-Zeitpunkt errechnet[33], und im Attribut des Timer-Objektes gespeichert.
Transitionen können mit Hilfe von speziellen Bedingungen auf Timer-Objekte reagieren. Zur Verfügung stehen die sechs Operationen: $<, >, =, \leq, \geq$ und "set to". Wobei letztere zum Setzen einer Zeit, und nicht zur Abfrage dient. Die Timer-Bedingung besteht aus drei Komponenten, dem Timer-Objektnamen, der Operation, und dem Wert (z.B. t1 $>$ 30). Beliebig viele solcher Bedingungen können nun an jede Transition im Netz angeschlossen werden. Zusätzlich zu den Marken-Kriterien müssen alle Timer-Kriterien einer Transition erfüllt sein, damit diese schalten kann.
Anders als die Vergleichsoperatoren arbeitet "set to" als Befehl. Schreibt man z.B. "t3 set to 40" in eine Timer-Bedingung, so setzt die betreffende Transition beim Schalten den Timer t3 auf 40 Sekunden.

4.2.6 Einbindung bestehender Klassen

Das generische Erzeugen von Marken, ausgehend von externen Objekten wird im *CS*-Petri-Netz durch einen Platz verwirklicht. Die Entscheidung fiel nicht auf eine Transition, da Transitionen im Petri-Netz Schaltvorgänge und Aktivitäten beschreiben. Ein Objekt, welches konstant existiert, lässt sich jedoch leichter als Platz beschreiben. Will man einen Platz generieren, welcher mehr Fähigkeiten besitzt als die CSPN-Place Klasse bietet, so hat man zusätzlich die Möglichkeit, eine eigene Klasse zu erstellen, die von der CSPN-Place Klasse erbt. Benutzt man das vom CS-Kontrollsystem bereitgestellte Vererbungswerkzeug, so erhält man eine fertige Klasse, einschließlich des notwendigen Launch.vi's. Will der Benutzer hingegen eine bestehende Klasse zur Petri-Netz Klasse aufrüsten, so ist er gezwungen von Hand den Konstruktor und Destruktor der CSPN-Place Klasse in die eigene zu platzieren, ein Launch.vi zu kopieren und in den eigenen ProcCases die ProcCases der CSPN-Klasse aufzurufen.

[32] siehe Glossar
[33] start = start_old + now – start_pause

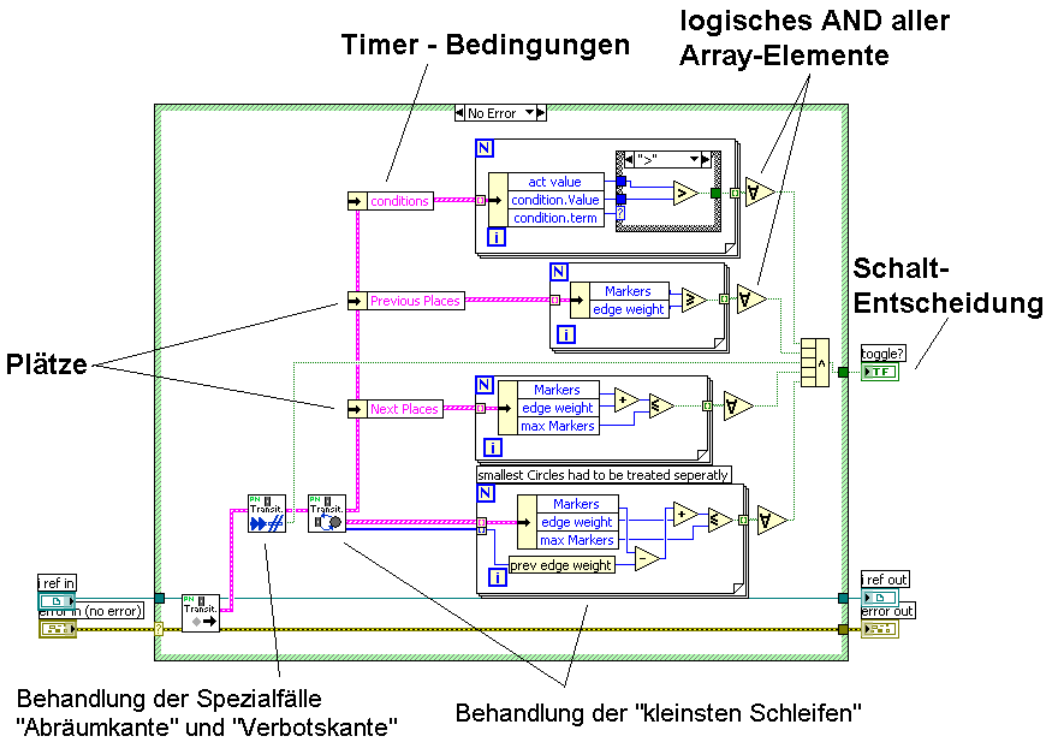

Abbildung 21: *Das Transition.Toggle-Check.vi*

Die so erzeugte Klasse hat alle Eigenschaften, die auch die CSPN-Place Klasse bietet. Eine Entry- und eine Exit-Action werden dem Benutzer standardmäßig bereitgestellt (siehe Abschnitt 4.2.9). Zudem hat dieser nun die Möglichkeit der Klasse beliebige eigene Programmstrukturen hinzuzufügen.

4.2.7 Der gemeinsame DIM-Listener

In den Objekt- und Petri-Netzen existiert ein einziger DIM-Listener, welcher alle benötigten DIM-Services abonniert. Diese Technik wurde eingesetzt, um Systemresourcen zu sparen und mögliche Wartezeiten zu minimieren. Der DIM-Listener wird vom CSPN-NetControl Objekt gestartet, nachdem das komplette Netz eingelesen wurde. Er meldet sich zu Beginn an allen benötigten DIM-Services an und erzeugt eine Warteschlange[34], auf die eingehende Daten später geschrieben werden. Sobald ein Datenpaket ankommt, wird anhand der “callBackID” die zugehörige Control-Referenz ausfindig gemacht. Je nach Datentyp werden die Infomationen nun auf dieses Kontrollelement geschrieben.
Wie auch bei den Objekt-Netzen wird, beim Auftreten eines Fehlers, der Hintergrund des Frontpanel-Anzeigeelements farbig markiert, um Informationen über die Qualität der Daten zu erhalten. Die Bedeutung der einzelnen Farben ist wie folgt festgelegt:

rot:	Kein DIM-Listener vorhanden/es ist kein Service definiert.
blau:	Keine Verbindung zum abonnierten Service.
gelb:	Die Daten des Dienstes sind alt/korrupt.
grün:	Es sind keine Fehler aufgetreten.

Der DIM-Listener arbeitet so lange, bis entweder ein Stopp-Signal im Attribut der Netzklasse gesetzt wird, oder ein Fehler beim Lesen des Attributes oder der Warteschlange entsteht. Danach löscht er die Warteschlange, meldet sich von allen DIM-Services ab und beendet sich selbst.

[34] In der Informatik bezeichnet eine Warteschlange (engl. Queue [kju]) eine häufig eingesetzte spezielle Datenstruktur. Eine Warteschlange kann eine beliebige Menge von Objekten aufnehmen und diese bei Bedarf wieder zurückgeben.

4.2.8 Generisch veränderbare Markenzahlen

Im *CS*-Petri-Netz besteht die Möglichkeit die Markenzahl von Plätzen beliebig programmatisch zu verändern. Die Methoden "CSPN-Place.Add Markers" und "CSPN-Place.Remove Markers" ermöglichen innerhalb der CSPN-Place Klasse einen gesicherten, einfachen Zugriff auf die Markenzahl eines Platzes. Hierbei wird die Markenzahl direkt verändert, falls der Reservierungsmechanismus inaktiv ist. Ist dieser aktiviert, so wird der benötigte Befehl zur Markenänderung in einer Warteschlange gespeichert, welche beim Beenden der Reservierung abgearbeitet wird. Sollten nicht mehr genug Marken für die Befehle in der Warteschlange vorhanden sein, oder ist die Kapazität des Platzes ausgelasted, so wird der jeweilige Befehl verworfen.

4.2.9 Einbettung des Netzes in die reale Umgebung

Wie schon erwähnt, existiert eine Schnittstelle, an der eine Kommunikation des Petri-Netzes mit der Aussenwelt möglich ist. Diese besteht hauptsächlich aus den drei VI's Entry-, Periodic-, und Exit-Action, soweit sie vom Benutzer zur Verfügung gestellt werden. Die Entry-Action wird aufgerufen, sobald auf einem Platz Marken erzeugt werden, die Exit-Action, wenn der Platz Marken verliert und die Periodic-Action zu einer festen Intervallzeit, unabhängig davon, ob der Platz Marken besitzt, oder nicht. Will der Anwender, dass eine periodische Aktion in Abhängigkeit der Markenzahl ausgeführt wird, so kann er die Markenzahl mit der passende Funktion innerhalb der ProcPeriodic abfragen.
Erbt man von der CSPN-Place Klasse, so bekommt man Entry- und Exit-Action automatisch, und besitzt eine eigene ProcPeriodic vom BaseProcess, in der man periodische Aktivitäten abhandeln kann. Benutzt man allerdings die schon vorgegebenen Platz-Klasse, so muss man die Pfade zu den jeweiligen Aktions-VI's direkt im Netz angeben.
Schon im Konstruktor versucht das Platz-Objekt eine Referenz auf die drei Methoden zu öffnen, soweit dem Objekt Pfade auf die benötigten Methoden bereit stehen. Beim späteren Ausführen der Methoden erfolgt dann ein Zugriff über die schon offene Referenz, was wiederum die Performanz steigert. Wird der Destruktor des Platzes aufgerufen, so werden alle offenen Referenzen geschlossen.

4.2.10 Spezielle Kanten

Um einem Petri-Netz die Mächtigkeit einer Turingmaschine[35] zu verleihen, bedarf es nur einer zusätzlichen Kante, der Verbotskante (siehe [Baumgarten2006]). Diese Verbotskante erlaubt einer Transition zu schalten, falls auf einem Platz keine Marken sind. Im Gegensatz zur gewöhnlichen, gewichteten Kante beginnt die Verbotskante immer an einem Platz und endet an einer Transition. Eine Veränderung der Markenzahl des angeschlossenen Platzes erfolgt hierbei nicht.

Eine weitere "exotische" Kante ist die Abräumkante. Wie auch die Verbotskante kann sie nur von einem Platz zu einer Transition zeigen. Die Abräumkante stellt keine Schaltkriterien auf, sie wird aktiv, sobald die angeschlossene Transition schaltet. Wie der Name vermuten lässt, räumt sie alle Marken ab, die auf dem Platz liegen, welcher mit ihr verbunden ist.
Abbildung 22 zeigt die grafische Notation der beiden Kanten. Die Abräumkante wird durch einen Doppelpfeil dargestellt, die Verbotskante durch einen doppelten Querstrich.

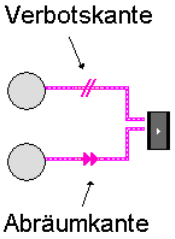

Abbildung 22: *Sonderkanten*

Die Verbotskante, sowie die Abräumkante werden auch im CS-Petri-Netz als zusätzliche Optionen zur Verfügung stehen. Sie werden dort als negative Kantengewichtungen erfasst[36] und gesondert ausgewerted (siehe auch Abbildung 21).

[35] Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing 1936 entwickeltes mathematisches Modell eines Computers. Das Besondere an einer Turingmaschine ist, dass sie mit nur drei Operationen (Lesen, Schreiben und Kopf bewegen) die Probleme lösen kann, die auch von einem Computer gelöst werden könnten. Sämtliche mathematischen Grundfunktionen wie Addition und Multiplikation lassen sich mit diesen drei Operationen simulieren. Darauf aufbauend kann man komplexere Programme erzeugen. Eine Funktion, die so durch eine Turingmaschine berechnet werden kann, nennt man eine turingberechenbare Funktion. (wikipedia)

[36] Abräumkante:(-2), Verbotskante: (-1)

4.3 Gemeinsame Merkmale beider Netzformen

Einige Merkmale sind Objekt- und Petri-Netzen gemein, was nicht zuletzt darin seinen Ursprung hat, dass die Petrinetze auf dem Fundament der Objektnetze erstellt wurden.

4.3.1 Namensgebung und Punkt-Konvention

Sobald man ein Netz (oder ein Subnetz) mehr als einmal verwenden will, stößt man auf ein Problem in der Namensgebung der Objekte. Statisch angelegte Objektnamen würden nun doppelt vorkommen, was dazu führt, das die betreffenden Objekte im *CS*-Framework nicht instanziert werden könnten.
Eine elegante Lösung für dieses Problem bietet die "Punkt Konvention". Die Objektnamen, die man in Objekt- und Petri-Netzen verwendet, werden automatisch um die Kette der aufrufenden Netze und Subnetze erweitert, mit einem Punkt als Trennzeichen. Der so erhaltene Name ist eindeutig und hat eine klare Struktur. So ist es möglich ein und dasselbe Netz beliebig oft zu instanzieren, ohne in die Gefahr zu kommen, doppelte Objektnamen zu generieren. In der aktuellsten Version der Netze wird die Punkt-Konvention erzwungen.

4.3.2 Globale Subnetzvariablen

Ein Subnetz erhält als Eingabeparameter zwar einen Namen, weiss dadurch jedoch nicht, in welchem Hauptnetz es sich befindet. Ebenso weiss ein Lauch.vi nicht von selbst, von welchem Netz aus es aufgerufen wird. Diese Information ist jedoch essenziell für alle Objekte im Netz.
Die triviale Lösung des Problems wäre, man übergibt jedem Launch.vi und jedem Subnetz als Übergabeparameter den Netznamen. Allerdings würden derart konstruierte Netze sehr schnell unübersichtlich werden. Die Möglichkeit, das Netz als Designdokument zu benutzen, ginge hiermit verloren.
Eine etwas komplexere, aber durchaus elegante Lösung, stellt hier die Benutzung funktionaler globaler Variablen[37] dar. Jedes Netz und jedes Subnetz besitzt einen eindeutigen Template- oder Instanznamen. Trägt sich nun ein solches Netz mit diesem Namen in eine funktionale globale Variable ein, braucht das VI, welches in diesem Netz aufgerufen wird, nur noch die Kette der Aufrufe (Call Chain) zurück zu verfolgen, den richtigen Template- oder Instanznamen dort finden, und den dort gespeicherten Netznamen auslesen. So hat jedes untergeordnete VI die Möglichkeit, programmatisch auf den Netznamen, in dem es verwendet wird, zuzugreifen.

4.3.3 Ablaufinvariante[38] Subnetze

Sobald in den Objekt- und Petri-Netzen Subnetze benutzt werden, benötigen diese Subnetze die Eigenschaft der Ablauf-Invarianz. Da ein Subnetz durchaus auch mehr als einmal aufgerufen werden kann, wird so sichergestellt, das zwei verschiede Subnetze auch immer verschiedene Addressen im Arbeitsspeicher haben. Zwar besteht noch die Möglichkeit an dieser Stelle VI-Templates (Siehe VIT im Glossar) zu nutzen. Jedoch bieten Ablauf-Invariante VI's zusätzlich den Vorteil, beim Öffnen direkt die richtige Instanz darzustellen[39], und sind damit bedeutend besser geeignet als die VI-Templates.

4.3.4 Reaktion auf externe Kommandos

Sowohl das Objekt- wie auch das Petri-Netz kann man mit programmatischen Kommandos steuern. Die Möglichkeit von BaseProcess zu erben, wurde hier bewusst nicht genutzt, um die Netzklasse nicht mit unnötigem Ballast zu überladen. Hier arbeitet parallel zur Kontrollereignissstruktur eine while-Schleife, welche auf eingehende Kommandos lauscht. Sobald ein Befehlskommando eintrifft, werden die gleichen Strukturen durchlaufen, die auch ein Tastendruck auslösen würde.

4.4 Beispiele

Anhand von Beispielen kann dem Benutzer einfach und nachvollziehbar die Funktionsweise der Objekt- und Petri-Netze klar gemacht werden.

[37] In Form von nichtinitialisierten Shift Registern, die auch im CS-Kontrollsystem zum Speichern der Objekt-Attributdaten genutzt werden
[38] Siehe Glossar
[39] Diese Möglichkeit existiert seit LabVIEW 8

4.4.1 Beispiele für *CS*-Objekt-Netze

“A very basic Object Net”

In diesem Netz (siehe Abbildung 23) werden alle grundlegenden Techniken, die man in Objekt-Netzen verwenden kann, aufgezeigt. Das Objekt ONTest 1 sendet ONTest 2 in diesem Beispiel einen Integerwert. ONTest 2 casted[40] diesen in eine Zeichenkette, die ONTest 1 wiederum, über das Shift Register als Eingangsparameter erreicht. Anders als bei den Petri-Netze besitzen Objekt-Netze keinen bipartiten Netzgraphen. Rückkopplungen müssen hier, wie in LabVIEW vorgesehen, über eine Schleife behandelt werden. Die im Beispiel verwendete while-schleife zeigt eine simple Anwendung dieser Technik. Beim zweiten Durchlauf erhalten alle Lauch.vi’s die für sie relevanten Informationen.
ONTest 3 zeigt, dass es nicht notwendig ist, einen Service, welchen man im Objekt-Netz startet, auch zu empfangen. Die Struktur in der rechten unteren Ecke verdeutlicht die Möglichkeit, auch Dienste zu abonnieren, welche nicht im Netz gestartet wurden, in diesem Fall den Dienst “CS_ Age” des *CS*-Systems.

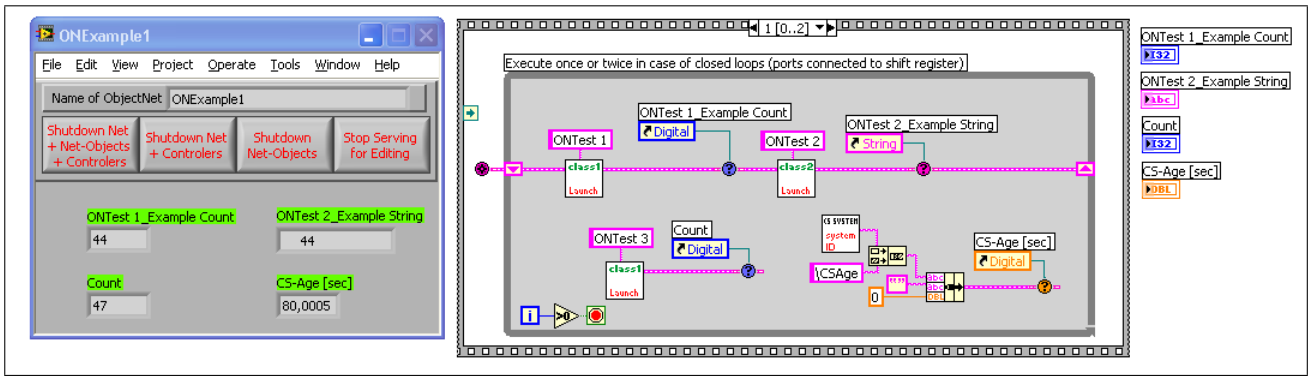

Abbildung 23: *Objekt-Netz Beispiel 1 — A very basic Object Net*

[40] Typumwandlung (engl. type conversion oder cast) bezeichnet in der Informatik die Umwandlung des Wertes eines Datentyps in einen Wert eines anderen Datentyps.(Wikipedia)

“Example of a hierarchic Object Net”

Dieses Netz (siehe Abbildung 24) zeigt die Verwendung von Subnetzen innerhalb der Objekt-Netze. Das ablaufinvariante[41] Subnetz “Room” wird im Hauptnetz zweimal mit verschiedenen Subnetznamen aufgerufen. Die so erzeugten Instanzen bilden wiederum eigene Netze, die völlig unabhängig voneinander operieren. Einzige Verbindung der beiden Netze ist der übergebene “Watch”-Service, welcher stets die aktuelle Zeit publiziert. Verschiedene Komponenten, welche zur Heizungs- und Lichtsteuerung notwendig sind, reagieren nun auf die Zeit, die gemessene Temperatur, den angestrebten SetPoint und den Status des Fensters. In diesem Beispiel ändern sich die simulierten Parameter ständig, um den Schaltvorgang zu verdeutlichen.
Da Blockdiagramme im Executable nicht mehr sichtbar sind, wurden die Zusammenhänge zwischen den einzelnen Parametern auf dem Frontpanel noch einmal mit Pfeilen aufgezeigt.

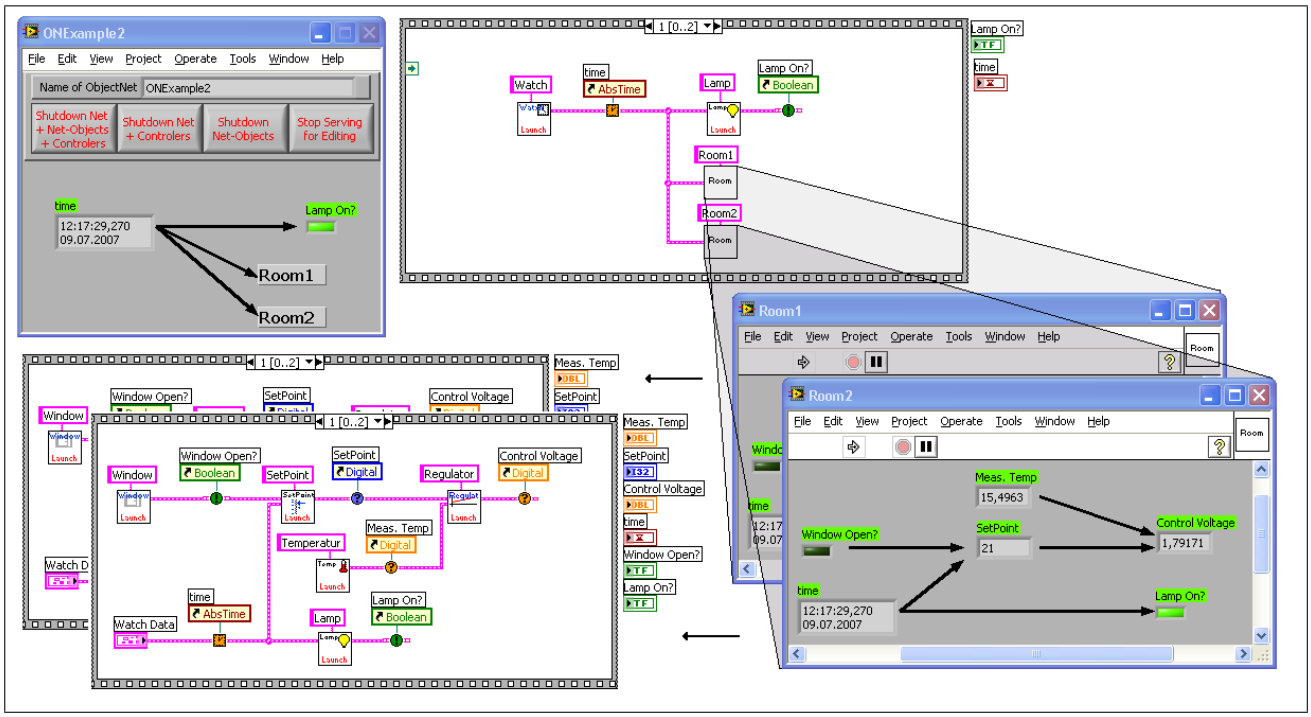

Abbildung 24: *Objekt-Netz Beispiel 2 — Example of a hierarchic Object Net*

[41] siehe Glossar

“Use of an existing deviceclass in the net”

Bei diesem Beispiel handelt es sich um ein Netz-Objekt, ohne zugehöriges Objekt-Netz. Es zeigt, wie man eine schon existierende Klasse im Objekt-Netz verwenden kann, ohne nur das geringste an dieser zu ändern.
Die hier implementierte Netzklasse hat von der CSON-BaseClass und von der SimAFG Klasse geerbt. Sie besitzt die Fähigkeit ein Objekt der SimAFG Klasse im Objekt-Netz darzustellen. Die wichtigste Modifikation der Klasse wurde innerhalb der ProcCases vorgenommen (siehe Abbildung 25). Jedes eingehende ON-Ereignis wird zu einem Ereignis der SimAFG Klasse gemacht und an deren ProcCases weitergeleitet. Ausgehende Ereignisse der SimAFG Klasse erhalten den Namen eines ON-Ereignisses und werden unter diesem Namen veröffentlicht.
Somit ist die SimAFG Klasse vollständig im Objekt-Netz gekapselt.

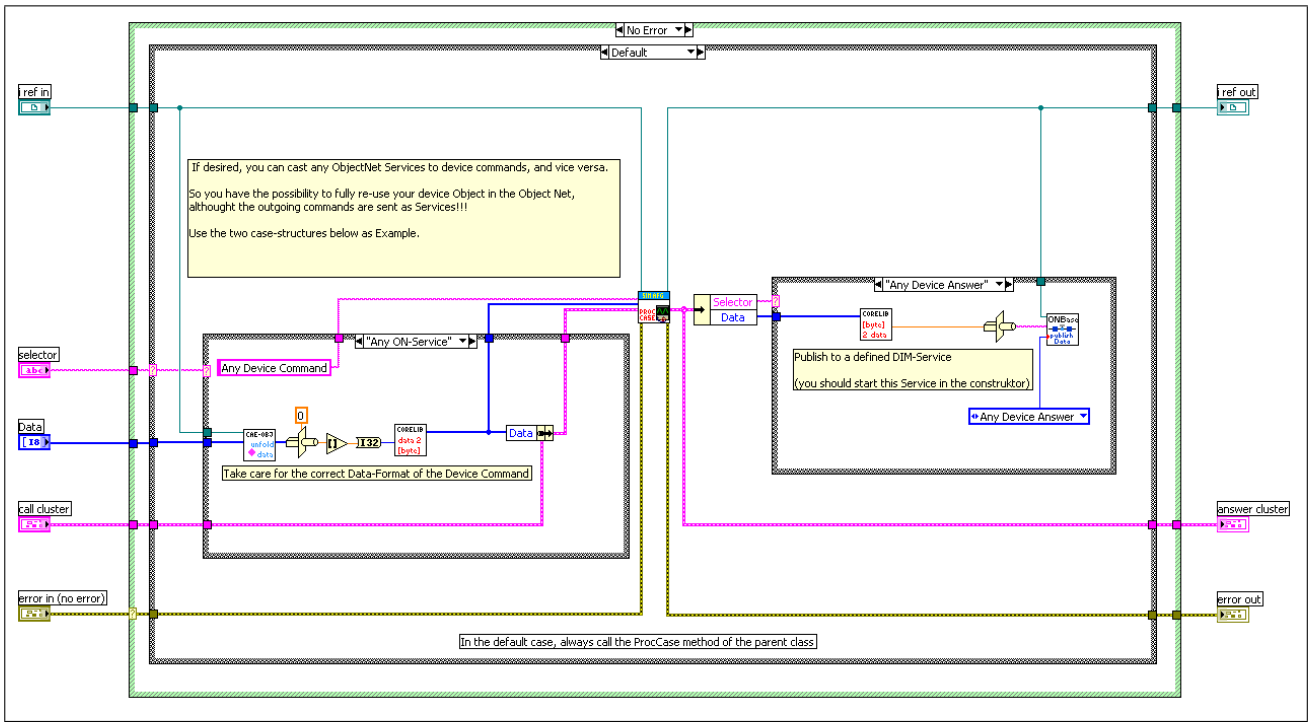

Abbildung 25: *Objekt-Netz Beispiel 3 — Proc Cases der “CSON Example DeviceClass”*

4.4.2 Beispiele für *CS*-Petri-Netze

“A very basic Petri Net”

Das hier gezeigte Petri-Netz (siehe auch Abbildung 26) modelliert die Produktion, die Verpackung und den Abtransport von Eiern. Der Platz “eggs” besitzt eine Periodic-Action, welche die Hühner simuliert, die ständig neue Eier legen. Sobald 12 Eier zur Verfügung stehen, werden diese, zusammen mit einem Karton zu einem vollen Eierkarton verpackt. Nun wird, in Abhängigkeit von der Zeit nachgesehen, ob 20 volle Kartons vorhanden sind. Wenn ja, werden diese zu einer Palette gebündelt und vom Eiertransporter abgeholt.
Dieses Beispiel gibt dem Benutzer einen ersten Eindruck über die Möglichkeiten, welche ein *CS*-Petri-Netz bietet. Es zeigt die Benutzung von Actions, enthält gewöhnliche, sowie boolesche Plätze und benutzt zudem die Timer-Klasse.

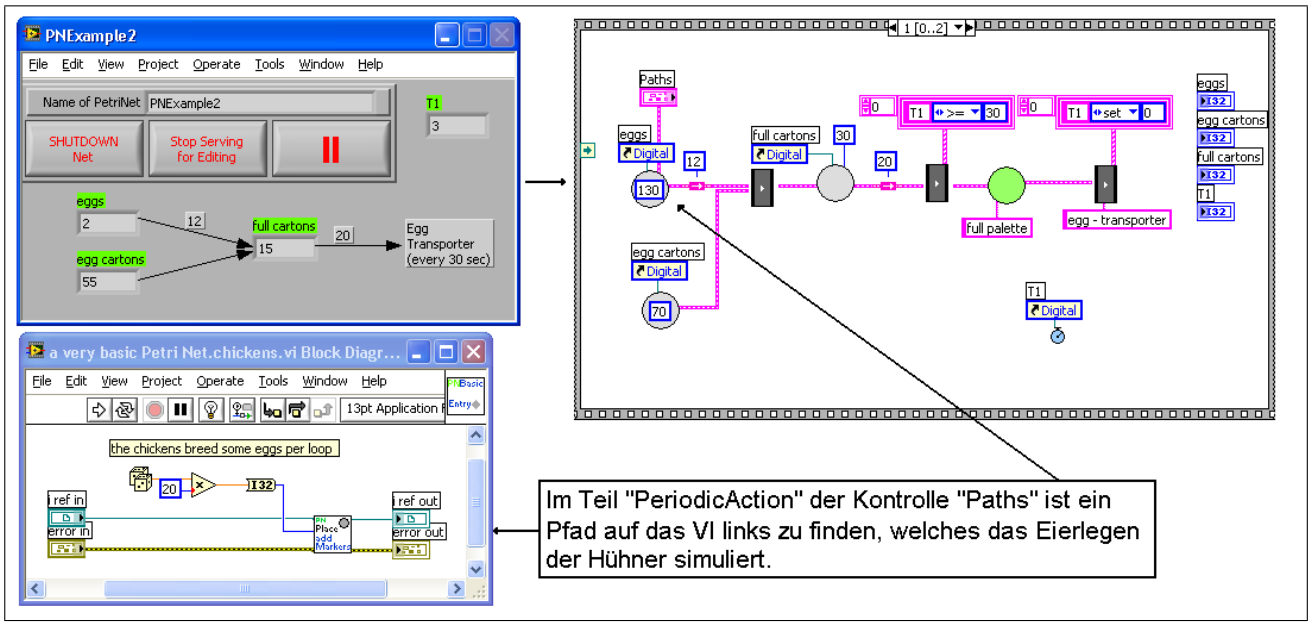

Abbildung 26: *Petri-Netz Beispiel 1 — A very basic Petri Net*

“Dinner of philosophers”

Ein oft benutztes Beispiel zur Demonstration der Fähigkeiten der Petri-Netze ist das “Dinner der Philosophen” (siehe auch Abbildung 27). Drei Philosophen, Aristoteles, Platon und Demokrit, sitzen an einem runden Tisch. Zwischen den Philosophen liegt jeweils eine Gabel. Sobald nun einer von ihnen zwei Gabeln in den Händen hält, kann er anfangen, zu essen, was jedoch auch dazu führt, dass die anderen beiden solange warten müssen. Das Beispiel beschreibt den Schaltkonfklikt von zwei Transitionen, ein typisches Petri-Netz Problem. Beide benötigen eine Marke zum Schalten, aber nur eine kann diese nehmen. Man versucht hier eine gewollte nicht-Deterministik zu modellieren.
Das *CS*-Petri-Netz reagiert angemessen, indem nach dem Schalten einer Transition die Freigabe der Plätze in zufälliger Reihenfolge abgearbeitet wird. Eine ungewollte Abhängigkeit wird so vermieden.
Durch das ablaufinvariante Subnetz “Philosoph” wird die Aktivität jedes einzelnen Philosophen sichtbar. Dieser ist entweder im Status “denken” oder im Status “essen”. Sobald zwei freie Gabeln vorhanden sind, versucht der Philosoph diese zu benutzen. Gelingt es der Transition, beide Gabeln (Plätze) zu reservieren, schaltet diese, und der Philosoph wechselt damit in den Zusatand “essen”. Nachdem er 3 Sekunden lang gegessen hat, legt er die Gabeln zurück, und wechselt in den Zustand “denken”. Sobald er eine Sekunde nachgedacht hat, bietet sich wieder die Möglichkeit, Gabeln zu reservieren und das Spiel beginnt von vorn.
Das Beispiel verdeutlicht zum einen die Nutzung von Subnetzen, und zeigt andererseits auch einfache Techniken in der Benutzung der Petri-Netze auf.

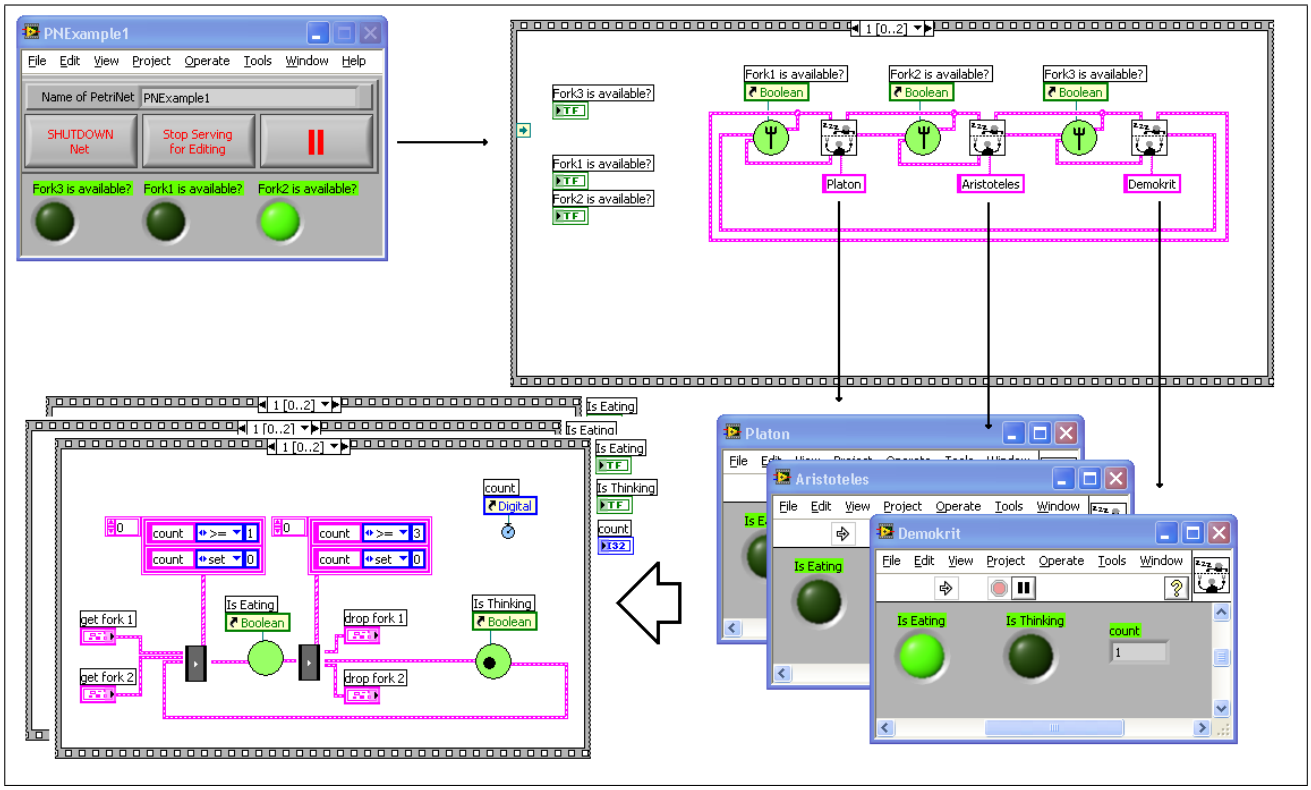

Abbildung 27: *Petri-Netz Beispiel 2 — Dinner of philosophs*

5 Schlussbetrachtungen

5.1 Ergebnis

Im Rahmen dieser Arbeit wurde das fertige Klassenkonstrukt der Objekt- und Petri-Netze vollständig ins *CS*-Framework integriert. Bereits bestehende Klassen können ohne große Probleme zu Netzklassen umgewandelt werden, was dem Entwickler die Möglichkeit gibt, schon funktionierende Strukturen erneut zu nutzen. Ein breit ausgelegtes Spektrum von Beispielen erleichtert dem Benutzer den Einstieg in die neue Technik und zeigt auf, wo eine Benutzung der Objekt- und Petri-Netze sinnvoll ist. Genauso steht es dem Benutzer frei, nur Teile seines Projektes in einem Netz zu verwirklichen. Die Modularität der Netze ermöglicht eine freie Kommunikation zu Nicht-Netz-Objekten und hält somit die Möglichkeit offen, Netz- und nicht Netz-Objekte beliebig zu kombinieren.

5.2 Ausblick

5.2.1 Anwendung

Ein erster Einsatz der Netze ist von Holger Brand für die Kontrolle einer Schrittmotorsteuerung vorgesehen. Es soll eine bereits existierende Lösung, welche sequenziell arbeitet mit Hilfe der Petri-Netze ersetzt werden. Ein genauer Aufbauplan und eine Liste der zu steuernden Parameter existiert noch nicht.
Die Objekt-Netze haben bisher noch keine konkrete Anwendung. Eine Nutzung dieser im HITRAP[42] Kontrollsystem wäre möglich und würde zeigen, ob sich diese auch in der Praxis bewähren.
Ob die beiden Netzformen von den Anwendern des *CS*-Frameworks angenommen werden, wird sich in Zukunft zeigen, sobald erste Rückmeldungen der Benutzer eingehen.

5.2.2 Höhere Petri-Netz -Formen

Oft sind Timer-Netze zum Modellieren komplexer Problemstellungen nicht mächtig genug, oder wird das dafür benötigte Netz zu groß und unübersichtlich. So liegt es nahe, in Zukunft auch Höhere Netzformen im *CS*-Framework zu implementieren.
Höhere Netze beschreiben allgemein Petri-Netze, die nicht mit Marken, sondern mit beliebigen Datenobjekten arbeiten. Transitionen besitzen Schaltbedingungen, welche beliebige logische Verknüpfungen enthalten können und Kanten tragen nicht nur Gewichtungen, sondern haben auch die Möglichkeit, Variablennamen zu tragen.

[42] Heavy Ion Trap. HITRAP(Siehe auch Abbildung 32 im Anhang) ist eine Beschleuniger- und Ionenfallenanlage, welche mit hochgeladenen Ionen aus dem ESR gespeist wird, diese abbremst, akkumuliert und abkühlt.

A Glossar

Ablaufinvarianz
Startet man ein gewöhnliches LabVIEW-VI, so kann dieses so lange es aktiv ist, nicht erneut von anderer Stelle gestartet werden, da im Arbeitsspeicher nur eine Instanz des Quellcodes existiert. Möchte man mehrere Instanzen eines VI's gleichzeitig abarbeiten, so existiert die Option dieses VI Ablauf-Invariant zu machen.

DIM(Distributed Information Management)
Ereignisgesteuerte Kommunikationsschnittstelle für verteilte Systeme, welches 1990 am CERN entwickelt wurde. Es basiert auf der Client-Server Architektur, d.h. ein Server publiziert Dienste, die eine beliebige Anzahl an Clients abonnieren können.
[siehe http://dim.web.cern.ch/dim/]

Framework
Begriff aus der Softwaretechnik, der insbesondere im Rahmen der objektorientierten Softwareentwicklung sowie bei komponentenbasierten Entwicklungsansätzen verwendet wird. Das Framework definiert die Schnittstellen und den Kontrollfluss der Anwendung. Ziel eines Frameworks ist die Wiederverwendung von Modulen, die durch das Framework definiert sind und dem Anwender als Vorlage dienen sollen.
[siehe http://de.wikipedia.org/wiki/Framework]

LabVIEW (Laboratory Virtual Instrument Engineering Workbench)
Grafische Programmiersprache von National Instruments, die häufig in der Mess- und Automatisierungstechnik zur Datenaufnahme und -verarbeitung verwendet wird. Die grafische Programmierung erfolgt mit der Programmiersprache G, die auf dem Modell des Datenflusses basiert und die Programmierung nebenläufiger Verarbeitung mit geringem Aufwand ermöglicht.
[siehe http://www.ni.com/labview/whatis/]

ObjectVIEW
Eine kommerzielle, objektorientierte Erweiterung für LabVIEW.

Objektnetz
Die Bezeichnung Objektnetz existierte schon in ObjectVIEW und bezeichnete dort ein Netz, welches Objekte starten und Kommunikationskanäle zwischen diesen aufbauen konnte. Im *CS*-Framework bietet das Objektnetz zusätzlich die Möglichkeit Objekte im verteilten Netzwerk zu starten und diese zu kontrollieren.

Petri-Netz
Ein Petri-Netz ist ein mathematisches Modell von nebenläufigen Systemen. Es stellt eine formale Methode der Modellierung von Systemen bzw. Transformationsprozessen dar. Die ursprüngliche Form der Petri-Netze nennt man auch Bedingungs- oder Ereignisnetz. Endliche Automaten und Bedingungs- oder Ereignisnetze sind gleichmächtig. Petri-Netze wurden durch Carl Adam Petri in den 1960er Jahren definiert. Sie verallgemeinern wegen der Fähigkeit, nebenläufige Ereignisse darzustellen, die Automatentheorie.(Wikipedia)

Platz (oder auch Stelle)
Ein Platz im Petrinetz beschreibt einen Container für beliebige Markenarten. Plätze können eine maximale Kapazität besitzen, müssen dies aber nicht.

Prozess

Ein Prozess bezeichnet den Ablauf eines Computerprogramms auf einem Prozessor und kann Threads des Betriebssystems belegen, falls dieses Multithreading unterstützt.
[siehe http://de.wikipedia.org/wiki/Prozess_%28Informatik%29]
Im Rahmes des *CS*-Frameworks ist ein Prozess eine Instanz einer beliebigen Unterklasse der Klasse BaseProc. Die grundlegende Eigenschaft eines Prozesses ist die Fähigkeit der Ereignisbehandlung. D.h. ein Prozess ist in der Lage, Ereignise zu empfangen und auf diese zu reagieren.

Semaphore
In der Informatik wird mit diesem Begriff eine Datenstruktur zur Prozesssynchronisierung bezeichnet (Wikipedia).
Greifen nebenläufige Programmteile auf gemeinsame Daten zu, wird ein Ausschlussmechanismus benötigt, der den Zugriff regelt. Solange Daten von einem Thread beschrieben werden, muss der Zugriff für alle anderen Threads gesperrt sein, ansonsten kommt es zu einem asynchronen Zugriff, der zu einer Inkonsistenz der Daten führen kann.
[siehe http://de.wikipedia.org/wiki/Semaphor(Informatik)]

Transition
Als Transition (lat. Übergang) bezeichnet man in der theoretischen Informatik den Übergang von Zustand q1 in den (nicht notwendig verschiedenen) Zustand q2 einer mathematischen Maschine (Wikipedia).
Im Petrinetz bezeichnen Transitionen Objekte, welche die Eigenschaft haben, unter bestimmten Bedingungen zu "schalten". Schaltet eine Transition, so verändert sie die Markenzahl aller Plätze, die mit ihr verbunden sind.

Thread
Programmaktivitäten, die nebenläufig, d.h. quasi-parallel, ausgeführt werden [Balzert2005] Seite 823.
In der Informatik ein Ausführungsstrang beziehungsweise eine Ausführungsreihenfolge der Abarbeitung der Software.
[siehe http://de.wikipedia.org/wiki/Thread_%28Informatik%29]
Im Kontext von LV bezeichnet der Begriff Teile des Quellcode, die über keine Datenabhängigkeit mit anderen Codefragmenten verbunden sind. Platziert man einfach zwei Schleifen im Blockdiagramm und verbindet diese nicht miteinander, werden beide Schleifen von LabVIEW asynchron ausgeführt.

UML(Unified Modelling Language)
Notation zur grafischen Darstellung Objektorientierter Konzepte [Balzert2005][Seite145]. Die weitgehend programmiersprachenunabhängige Symbolsprache umfasst 13 verschiedene Diagrammtypen, mit denen verschiedene Aspekte eines Softwaresystems beschrieben werden können.
[siehe http://de.wikipedia.org/wiki/UnifiedModelingLanguage]

VI (Virtual Instrument)
Bezeichnung für ein (Unter-)Programm im Rahmen der grafischen Programmiersprache LabVIEW. Ein VI besteht aus einem Front Panel zur Erstellung von Bedienoberflächen und Deklaration der Übergabeparameter und einem Blockdiagramm, das der Implementierung des Quellcodes dient.

VIT(Virtual Instrument Template)
Vorlage oder Muster für ein VI. Zur Laufzeit können von einer Vorlage beliebig viele Instanzen in den Speicher geladen werden, auf die mittels einer eindeutigen Referenz zugegriffen werden kann.

Watchdog
Technischer Begriff für eine Komponente eines Systems, die Überwachungsaufgaben übernimmt. Im Fehlerfall wird diese aktiv, um wieder andere Prozesse auszulösen, die den Fehler beheben können.
[siehe http://de.wikipedia.org/wiki/Watchdog]

B Diagramme

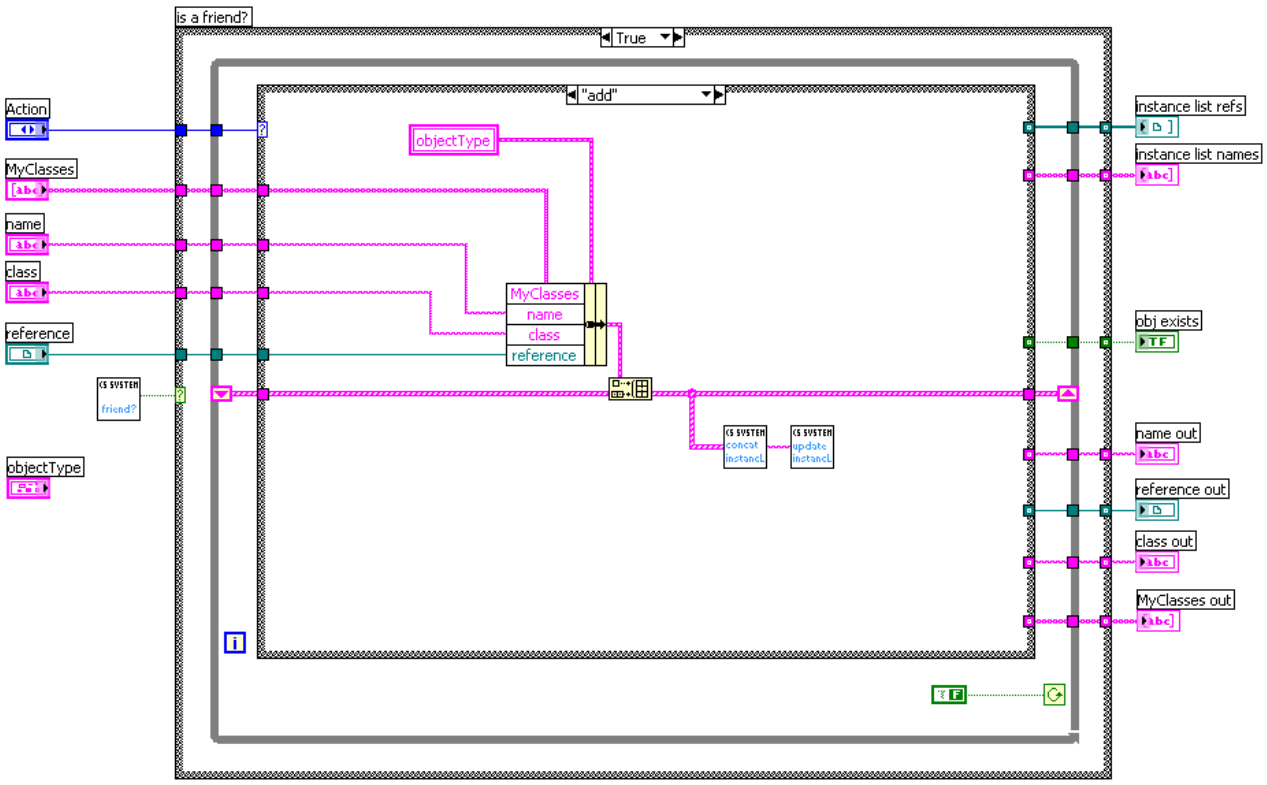

Abbildung 28: LabVIEW-Blockdiagramm: CSSystemLib.instances.vi verwaltet die Objekte und Referenzen innerhalb eines lokalen CS-Systems

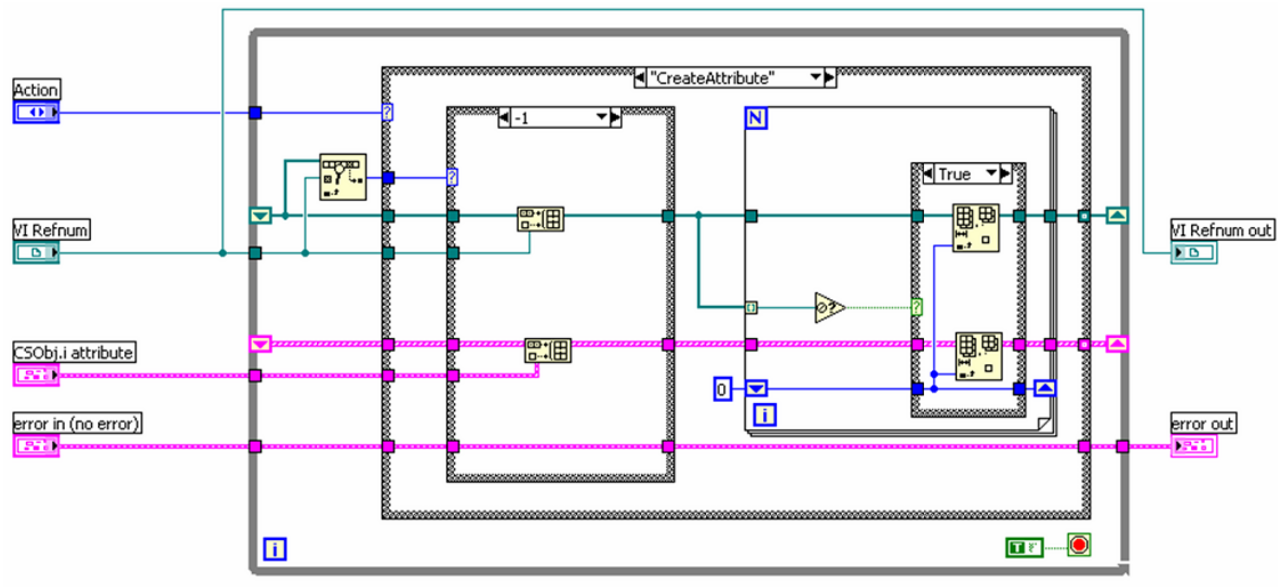

Abbildung 29: LabVIEW-Blockdiagramm: CLASSNAME.i attribute.vi zur Verwaltung der Attribute aller Instanzen einer Klasse

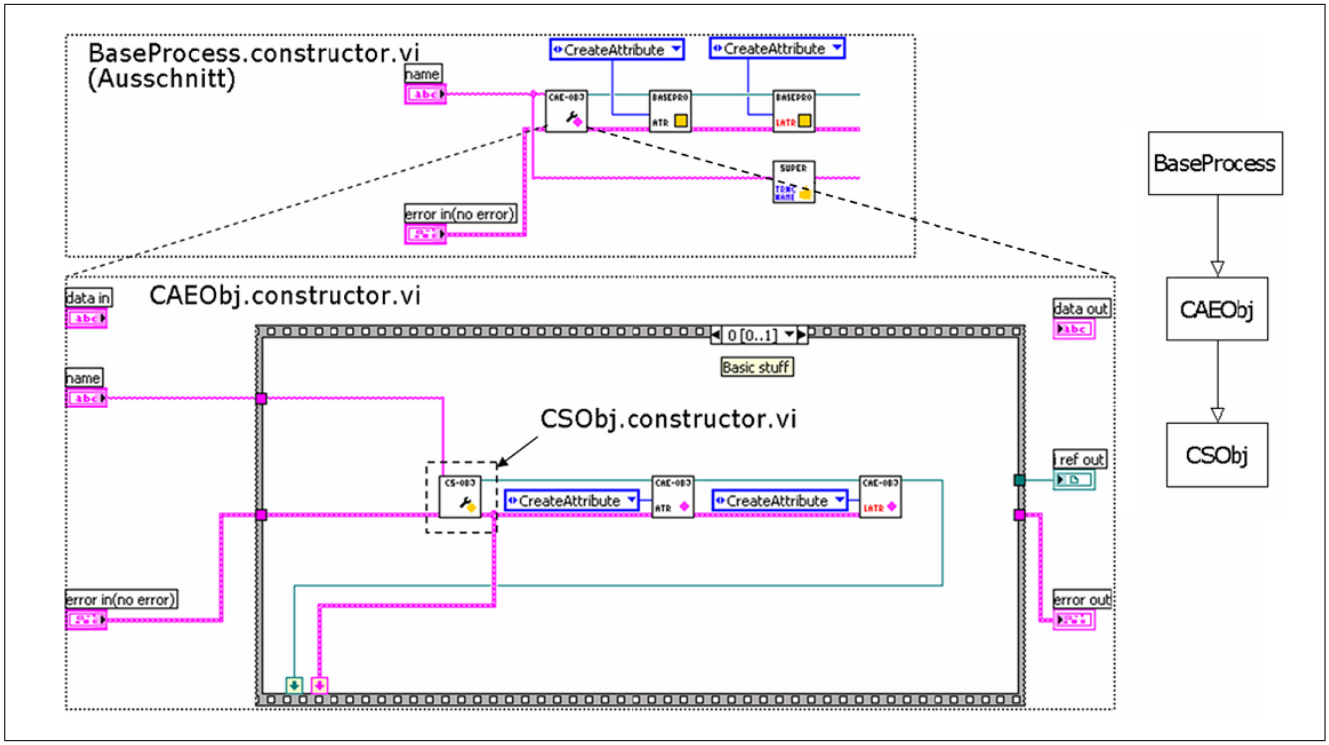

Abbildung 30: LabVIEW-Blockdiagramme: Vererbung durch geschachtelte Konstruktoren

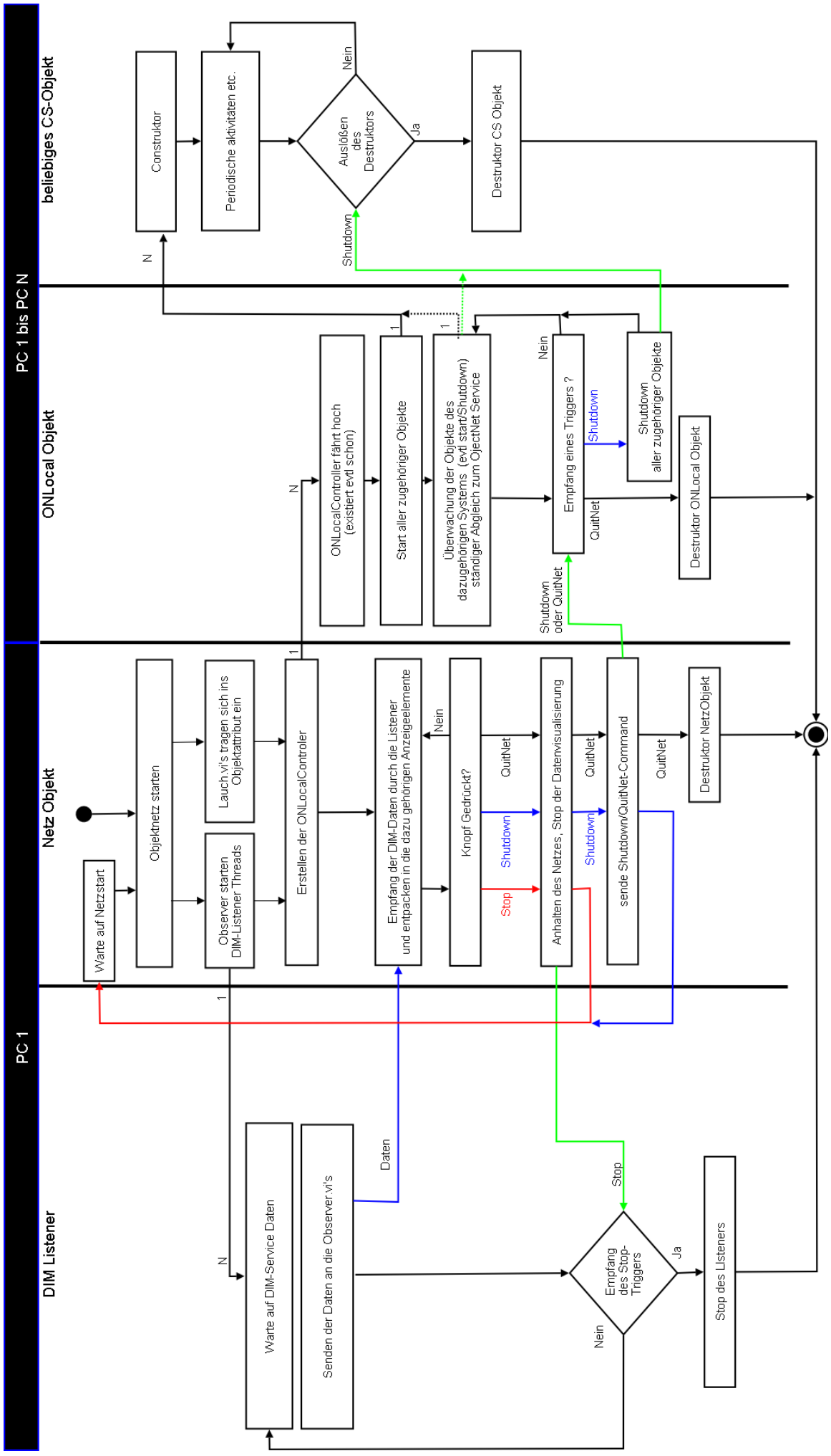

Abbildung 31: Grundlegener Aufbau von Objekt- und Petri-Netzen

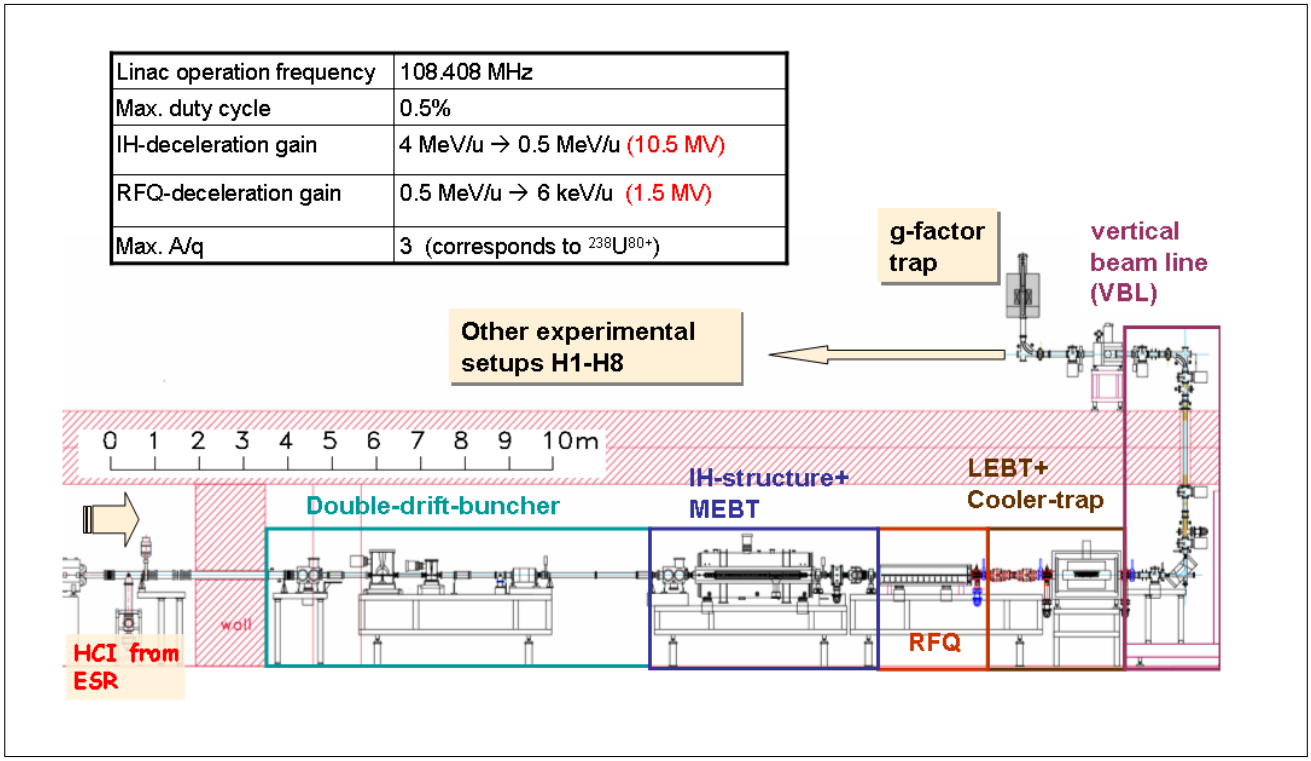

Abbildung 32: Übersicht der HITRAP beam line

C Literaturverzeichnis

Literatur

[Baumgarten2006] Dr. Bernd Baumgarten Script zur Vorlesung “Petri-Netze”
Bernd Baumgarten 2006

[Balzert2005] H. Balzert
Lehrbuch Grundlagen der Informatik
2.Auflage
Spektrum Akademischer Verlag, ISBN 3-8274-1410-5

[Beck2003a] D.Beck, H.Brand
Ein allgemeines Kontrollsystem für Experimenteinrichtungen an der GSI
„Virtuelle Instrumente in der Praxis 2003“
München, Germany, Herausgeber R. Jamal and H. Jaschinski, ISBN 3-7785-2908-0
(2003) 30-34

[Beck2003b] D.Beck, H.Brand, F.Herfurth
CS- A Control System Framework for Experiments (not only) at GSI
„The IX International Conference on Accelerator and Large Experimental Physics Control Systems“
ICALEPCS 2003, Gyeongju, Korea

[Beck2005] D.Beck, H.Brand
Status and Developement of the *CS* Control System Framework
„SEI Frühjahrstagung 2005“
Darmstadt, Germany, Herausgeber F. Wulf

[Beck2005] D. Beck
Anwendung des allgemeinen Kontrollsystems *CS*-Framework auf eine flexible Schrittmotorsteuerung an der GSI
„Virtuelle Instrumente in der Praxis 2004“
Fürstenfeldbruck, Germany, Herausgeber R. Jamal and H. Jaschinski, ISBN 3-7785-2932-3, 250-254

[Brand2005] H.Brand
CS-Kurshandbuch
Version 1.2
Ausgabe November 2005

[Gaspar1993] Gaspar/Dönszelmann
DIM - A Distributed Information Management System for the Delphi experiment at CERN
Vancouver, June 8-11 1993

[Gaspar2000] C. Gaspar1, M. D¨onszelmann1, Ph. Charpentier1
DIM, a Portable, LightWeight Package for Information Publishing, Data Transfer and Inter-process Communication
Padova, Italy, 1-11 February 2000
http://dim.web.cern.ch/dim/papers/CHEP/DIM.PDF

[Kugler06] Diplomarbeit Maximilian Kugler
Hochschule Darmstadt, Fachbereich Elektrotechnik und Telekommunikation
Entwicklung einer Klassenbibliothek zur Erstellung generischer Sequenzen im Rahmen des *CS*-Frameworks .

[Wiki1] http://wiki.gsi.de/cgi-bin/view/CSframework/CSObjectOrientation (Stand:27.06.2006)

[Wiki2] http://wiki.gsi.de/cgi-bin/view/CSframework/CSDocuments (Stand:29.07.2006)