Machbarkeitsstudie: LVOOP basiertes Agenten-System

Dr. Holger Brand, Dr. Dietrich Beck, Frederik Berck
GSI Helmholtzzentrum für Schwerionenforschung GmbH, Darmstadt

Kurzfassung

LabVIEW verfolgt seit Version 8.2 ein eigenes Konzept für objekt-orientiertes Programmieren, das konsequent dem Datenfluss-Paradigma folgt [1,2]. Die meisten objektorientierten Entwurfsmuster [3] müssen deshalb mit Bezug auf den Datenfluss neu erfunden werden. Die in diesem Beitrag vorgestellte LabVIEW Klassenbibliothek [4,5] ist auf den Aspekt fokussiert, mit LabVIEW Objekten als Entitäten oder Agenten in parallelen, ereignisgesteuerten und verteilten Applikationen umzugehen. Sie enthält (Basis-) Klassen, die folgende Entwurfmuster implementieren: *Fabrik* (*Factory*), *Referenz*, *Besucher* (*Visitor*), *Thread-Pool* und *Ereignis* (*Event*).

Abstract

Since version 8.2, LabVIEW provides its own approach to object oriented programming consequently following the dataflow paradigm [1,2]. Therefore most well known object oriented design patterns [3] have to be reinvented with respect to dataflow. The LabVIEW class library [4,5] described in this report is focussed to deal with LabVIEW objects as entities or agents in multithreaded, event driven and distributed applications. It provides classes implementing following design patterns: *Factory*, *Reference*, *Visitor*, *Thread-Pool* and *Event*s.

Einführung

Im Gegensatz zu konventionellen objektorientierten Sprachen, wie C++ oder Java, werden Objekte in LabVIEW nicht als Entitäten behandelt, sondern, genauso wie alle anderen elementaren LabVIEW Datentypen, als passive Datenobjekte, die dem Datenflusskonzept folgen [1,2]. Sie fließen von einer Datenquelle zu Datensenken. An einem Drahtabzweig wird ein Klon eines Objektes erzeugt und weitergeleitet. Der Datenflussansatz erhält zwar die intrinsische *multi-threading* Fähigkeit von LabVIEW, birgt aber Probleme für Klassen, die mit Attributen versehen sind, die auf eindeutige Ressourcen verweisen, die nicht automatisch zusammen mit dem Objekt geklont werden, z. B. Dateien, Netzwerkverbindungen oder

physikalische Geräte. Anders als z. B. in Java können LabVIEW Objekte deshalb auch nicht aktiv sein, also keine eigenen Threads starten, weil es sich auch bei Threads um Ressourcen handelt, die nicht einfach geklont werden können.
Da LabVIEW Objekte konsequent dem Datenfluss-Paradigma folgen, müssen alle wesentlichen objektorientierten Entwurfsmuster [3], mit Bezug auf den Datenfluss neu gedacht und implementiert werden. Insbesondere sind Möglichkeiten zu schaffen, die es auf einfache Weise erlauben, LabVIEW Objekte als Entitäten zu behandeln und in parallelen Threads zu verwenden. LabVIEW Objekte können nicht direkt referenziert werden, sondern nur indirekt, durch den Einsatz des *Referenz*-Musters, im Wesentlichen Warteschlangen der Länge 1, vgl. auch LabVIEW Beispiel *ReferenceObject.lvproj*. Das Entwurfsmuster *Fabrik* erlaubt das generische Erzeugen von initialisierten Objekten von Klassen, die nur mit Namen angegeben werden. Initialisierungsparameter werden als *Variant*-Attribute übergeben. Dieses Muster kann als Ersatz für fehlende Konstruktoren eingesetzt werden. In einer früheren Studie wurden diese Entwurfsmuster zu einem Objekt-Manager kombiniert und damit erfolgreich ein Geräte-Server implementiert [7].
Darüber hinaus soll ein **Agenten**-Objekt zusätzlich autonom agieren und kommunizieren können und möglicherweise mobil sein. Die in diesem Beitrag vorgestellte LabVIEW HGF Klassenbibliothek [5,6] ist auf diese Aspekte fokussiert und enthält (Basis-) Klassen, die folgende Entwurfmuster implementieren: *Fabrik (Factory)*, *Referenz*, *Besucher (Visitor)*, *Thread-Pool* und *Ereignisse (Event)*. Diese Muster erlauben die generische Aktivierung von passiven Datenobjekten, die als Agenten behandelt werden sollen.

Agenten: Entwurfsmuster und Klassendiagramme

Da LabVIEW Objekte immer passiv sind, müssen sie durch die Applikation aktiviert werden. Um einem generischen und skalierbaren Ansatz gerecht zu werden, sollen auch die Algorithmen, *Tasks*, in Klassen gefasst werden. Das Entwurfsmuster *ThreadPool*, linkes Klassen-diagramm in Bild 1, ist die Lösung für diese Problemstellung. Die Klasse *HGF_ThreadPool* startet Threads, *HGF_ThreadWorker*. Diese rufen ein *dynamic dispatch* VI der Klasse *HGF_ThreadTask* auf, deren Kindklassen die eigentliche Aktivität ereignisgesteuert implementieren. Ein LabVIEW Objekt, das als Entität behandelt werden soll, kann in einem solchen Task als Attribut leben und sein Draht ist von außen nicht mehr erreichbar. Dadurch werden unbeabsichtigte Drahtabzweige effektiv verhindert. Die Kommunikation mit solchen nicht per Draht erreichbaren Entitäten kann mit Hilfe des Entwurfsmusters *Besucher* gelöst werden, das im rechten Diagramm von Bild 1 illustriert ist. Die *NInstrSim*-Klasse, die den Gerätesimulator [8,9] repräsentiert, ist von *HGF_Visitable* und *HGF_DeviceBase* abgeleitet. Die Geräte-Basisklasse definiert *dynamic*

dispatch VI's, die von den Kindklassen überschrieben werden können, um die gerätespezifischen Aktivitäten zu implementieren. *HGF_Visitable*-VI's werden in diesem Muster nicht direkt von der Applikation aufgerufen, sondern von Besucher-Klassen, die von *HGF_Visitor* geerbt haben, in diesem Beispiel *NInstrSim_ReadWaveform*. Der Besucher hält die Eingangs- und Ausgangsparameter für den SubVI-Aufruf der Geräteklasse in seinen Attributen. Ein Besucher wird von der Applikation mittels Ereignis an den beherbergenden Task gesendet. Beide Klassendiagramme enthalten Assoziationen zu Netzwerkereignissen, *HGF_NetworkEvent* und Prozessvariablen, *HGF_PVBase*, z.B. *SharedVariablen*. Ein *Netzwerk-Visitor* kann alternativ auch von einem vorkonfigurierten Ereignis, das durch Änderung einer Prozessvariablen ausgelöst wird, an einen Task, gesendet werden, der diesen erwartet.
In dieser Studie wird ein **Agent** als Kombination eines passiven Objekts, *HGF_Visitable*, und eines aktiven Tasks, *HGF_ThreadTask*, aufgefasst. Das Verhalten eines **Geräteagenten** wird in diesem Beispiel von der Zustandsmaschine *HGF_DeviceFSM* abstrakt definiert, vgl. Bild 2 Mitte links, und von *HGF_TaskDFSM* getrieben.

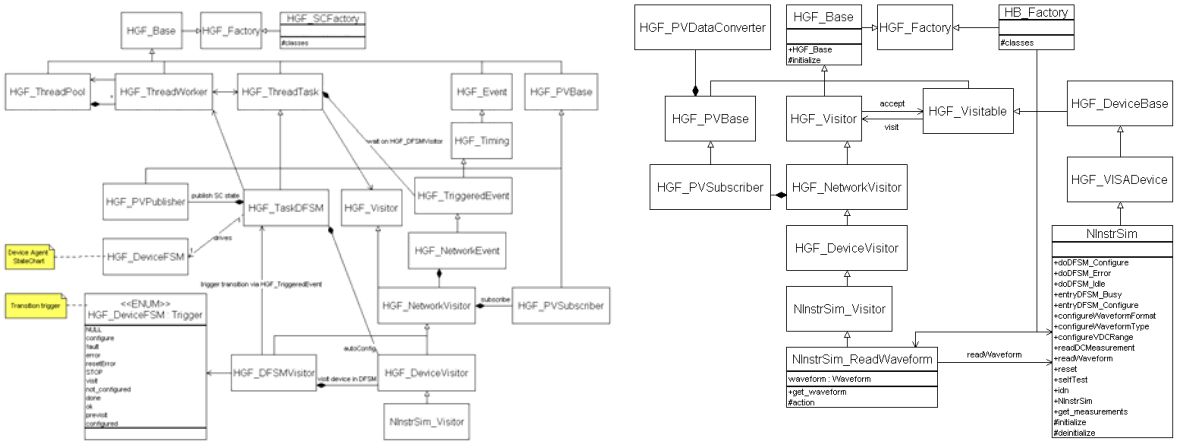

Bild 1: UML-Klassendiagramme illustrieren das Besucher Entwurfsmuster und Assoziationen zwischen den Klassen, die zum Thread-Pool gehören, um die Gerätezustandsmaschine HGF_DeviceDFSM zu treiben. Die Diagramme enthalten auch die Assoziationen zu Netzwerkprozess-variabel-Klassen.

Beispiel: Erzeugung und Verhalten eines stationären Geräteagenten

Die obere Sequenz in Bild 2 oben zeigt die Erzeugung der notwendigen Objekte für die Netzwerkereignisbehandlung. Ein Gerätebesucher wird mit einer *SharedVariablen* assoziiert und als Parameter einem Taskbesucher mitgegeben, der bei Änderung der *SharedVariablen* via *HGF_Event* an den Geräteagenten gesendet wird. In der unteren Sequenz wird das *HGF_TaskDFSM*-Objekt mit den Initialisierungsparametern für das Geräteobjekt und dem *SharedVariable*-Ereignisobjekt an den *ThreadPool*

gesendet und im Hintergrund von einem *Worker* zum Leben erweckt. Die Schleife illustriert die Änderung der *SharedVariablen*, die im Hintergrund den Besuch des Geräteagenten durch den Gerätebesucher bewirkt.
Die möglichen Aktionen bei Zustandsübergängen und periodischen Funktionen, *Entry-*, *Exit-* und *Do*-Aktionen, werden von der Klasse *HGF_DeviceBase* für alle Zustände virtuell definiert und können von Kindklassen überschrieben werden. Das Geräteobjekt wird in der Transition */createDevice* mit Hilfe einer *Fabrik* erzeugt, initialisiert und als Attribut in dem *StateData*-Cluster intern gespeichert, vgl. Bild 2 Mitte rechts. Bei Terminierung der Zustandmaschine, *STOP/destroyDevice*, wird das Geräteobjekt wieder vernichtet. Während des Lebenszyklus im *Alive*-Zustand, wechselt der Agent von *NotConfigured* über *Configure* zu *Configured.Idle.OK*. Im *Idle*-Zustand überwacht das Geräteobjekt optional periodisch das physikalische Gerät und wechselt bei leichten temporären Fehlern zwischen *OK* und *Fault*. Während sich der Agent im *Idle*-Zustand befindet kann er von *HGF_DeviceVisitor*-Objekten besucht werden. Damit ist ein temporärer Wechsel in den Zustand *Busy* verbunden, in dem die gewünschten Aktionen ausgeführt, also SubVI's der Geräteklasse vom Besucherobjekt aufgerufen werden, *accept.vi* in Bild 2 unten. Bei schweren Fehlern wechselt der Agent in den Zustand *Error*, den er nur durch den expliziten Trigger *resetError* verlassen kann, nachdem der Fehler durch ein periodisch ausgeführtes *dynamic dispatch* VI behoben wurde.

Zusammenfassung und Ausblick

In diesem Artikel wurde gezeigt, dass sich Agenten abstrakt durch die Kombination von passivem Datenobjekt und Task definieren lassen und mittels Thread-Pool und Besuchern aktiviert werden können. Kindklassen implementieren die konkreten Aktivitäten. Für die Realisierung mobiler Agenten ist zusätzlich eine weitere Task-Klasse notwendig, die LabVIEW Klassen dynamisch laden kann, notwendige Authentisierungsverfahren implementiert, Sicherheitsaspekte berücksichtigt und Introspektion erlaubt. Diese Aufgabe ist Gegenstand der Diplomarbeit von Herrn Berck.

[1] http://zone.ni.com/devzone/cda/tut/p/id/3573
[2] http://zone.ni.com/devzone/cda/tut/p/id/3574
[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994
[4] H. Brand D.Beck, "The HGF Base Class Library based on LVOOP", GSI Scientific Report 2008, Seite 260, ISSN: 0174-0814, http://www.gsi.de/library/GSI-Report-2009-1
[5] http://wiki.gsi.de/cgi-bin/view/NIUser/HGFBaseClassLibrary
[6] HGF: Helmholtz, GSI, FAIR
[7] D. Beck and H. Brand, "Control System Design Using LabVIEW

Object Oriented Programming", Proceedings of ICALEPCS07, Knoxville, Tennessee, USA
[8] http://sine.ni.com/nips/cds/view/p/lang/de/nid/10763
[9] http://zone.ni.com/devzone/cda/epd/p/id/1540

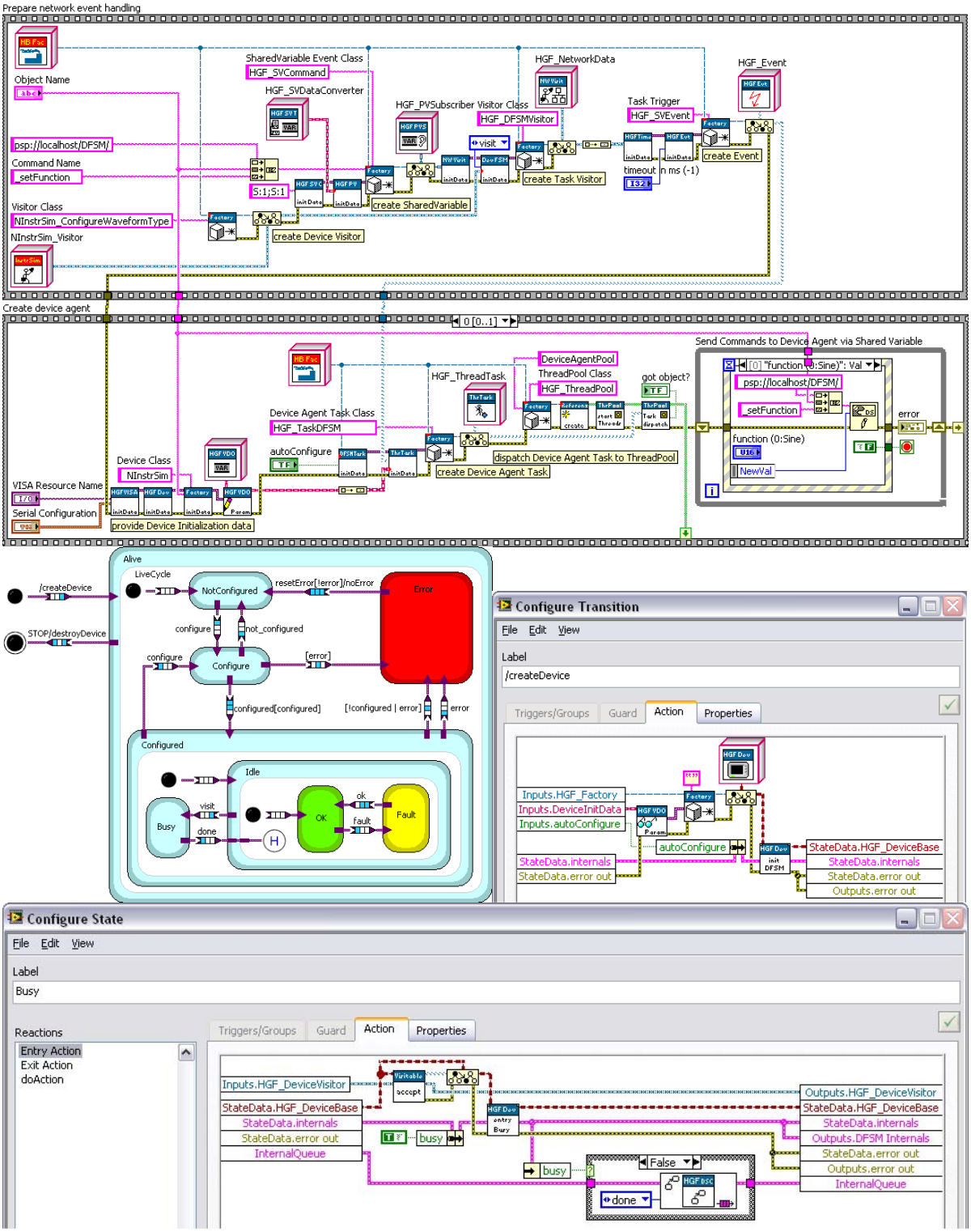

Bild 2: Lebenszyklus eines Geräteagenten. Oben: Erzeugung und Aktivierung des Geräteagenten-Tasks. Mitte: Zustandsmaschine und

Erzeugen des Geräteobjekts. Unten: Ausführung von Besucher-Aktionen, die durch Änderung von Netzwerkvariablen ausgelöst werden.