Prototyp
für ein
mobiles Agentensystem
in NI LabVIEW

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbeit
FACHBEREICH ELEKTROTECHNIK
UND INFORMATIONSTECHNIK

DIPLOMARBEIT

Thema: **Prototyp**

für ein mobiles Agentensystem

in NI LabVIEW

Bearbeiter/in: **Frederik Berck**

Referent/in: Prof. Dr.-Ing. Michael Kuhn

Korreferent/in: Prof. Dr.-Ing. Ulrich Schultheiß

Abgabe am: **06.04.2010**

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbeit
FACHBEREICH ELEKTROTECHNIK
UND INFORMATIONSTECHNIK

BEARBEITER/IN:

Name: Berck Vorname: Frederik

Geburtstag: 28.09.1982 Matrikel-Nr.: 690096

Adresse: Wilhelmstraße 31, 63225 Langen

Fachgebiet: Elektrotechnik - Telekommunikation und Informationstechnik

Referent/in: Prof. Dr.-Ing. Michael Kuhn

Korreferent/in: Prof. Dr.-Ing. Ulrich Schultheiß

Thema: Prototyp für ein mobiles Agentensystem in NI LabVIEW

Kurzreferat: (max 10 Zeilen)

In LabVIEW werden Objekte nicht als Entitäten behandelt, sondern als passive Datenobjekte die dem Datenflusskonzept folgen. Die HGF Basisklassenbibliothek (Dr. Holger Brand, GSI) stellt Klassen bereit, die wichtige Entwurfsmuster datenflusskonform implementieren und die Behandlung von Objekten als Entitäten erlauben. Um die Möglichkeiten und Restriktionen der datenflussbasierten objektorientierten Programmierung im Hinblick auf zukünftige Projekte tiefergehend zu analysieren soll ein mobiles Agentensystem auf Grundlage der HGF Basisklassen implementiert werden.
Ein Prototyp für ein mobiles Agentensystem wurde im Rahmen der Diplomarbeit erstellt.

Die Arbeit wurde mit/ohne Kooperationspartner (extern) durchgeführt:

Institution: GSI Helmholtzzentrum für Schwerionenforschung GmbH

Anschrift: GSI Helmholtzzentrum für Schwerionenforschung GmbH
Planckstr. 1
64291 Darmstadt

Die Arbeit ist gesperrt: ☐ ja ☒ nein

Unterschrift des/r Referenten/in:

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbeit
FACHBEREICH ELEKTROTECHNIK
UND INFORMATIONSTECHNIK

Bearbeiter:

Name: Berck Vorname: Frederik

Referent/in: Prof. Dr.-Ing. Michael Kuhn Korreferent: Prof. Dr.-Ing. Ulrich Schultheiß

ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Soweit ich auf fremde Materialien, Texte oder Gedankengänge zurückgegriffen habe, enthalten meine Ausführungen vollständige und eindeutige Verweise auf die Urheber und Quellen.
Alle weiteren Inhalte der vorgelegten Arbeit stammen von mir im urheberrechtlichen Sinn, soweit keine Verweise und Zitate erfolgen.
Mir ist bekannt, dass ein Täuschungsversuch vorliegt, wenn die vorstehende Erklärung sich als unrichtig erweist.

Darmstadt, den 06.04.2010 .....
(Unterschrift)

Information zur Aufbewahrung der Abschlussarbeit

Ich bin darüber informiert, dass meine Abschlussarbeit vom* 06.04.2010

– im Fachbereich EIT nach der HImmaVO, in der jeweils gültigen Fassung, aufbewahrt wird,
zur Zeit gilt:

§ 23, Abs.3, S. 2 Fünf Jahre aufzubewahren sind die übrigen Prüfungsunterlagen von Hochschulprüfungen, soweit sie nicht zurückgegeben werden.
§ 23, Abs. 4 Die Aufbewahrungsfristen für die Prüfungsunterlagen beginnen mit Ablauf des Kalenderjahres, in dem dem Prüfling das endgültige Ergebnis der jeweiligen Prüfung mitgeteilt worden ist.

– in Kooperation mit Firmen eine längere Aufbewahrungsfrist haben kann.

– nach Ablauf der oben genannten Aufbewahrungsfrist im Fachbereich vernichtet wird.

– in weiteren Exemplaren bei dem/der Referenten/in verbleibt.

– Diese Information gilt auch für die mit der Abschlussarbeit abgegebenen Fotos oder Datenträger.

Darmstadt, den 06.04.2010 Unterschrift: .....

* Abgabedatum

Inhalt

Einleitung

GSI Helmholtzzentrum für Schwerionenforschung GmbH

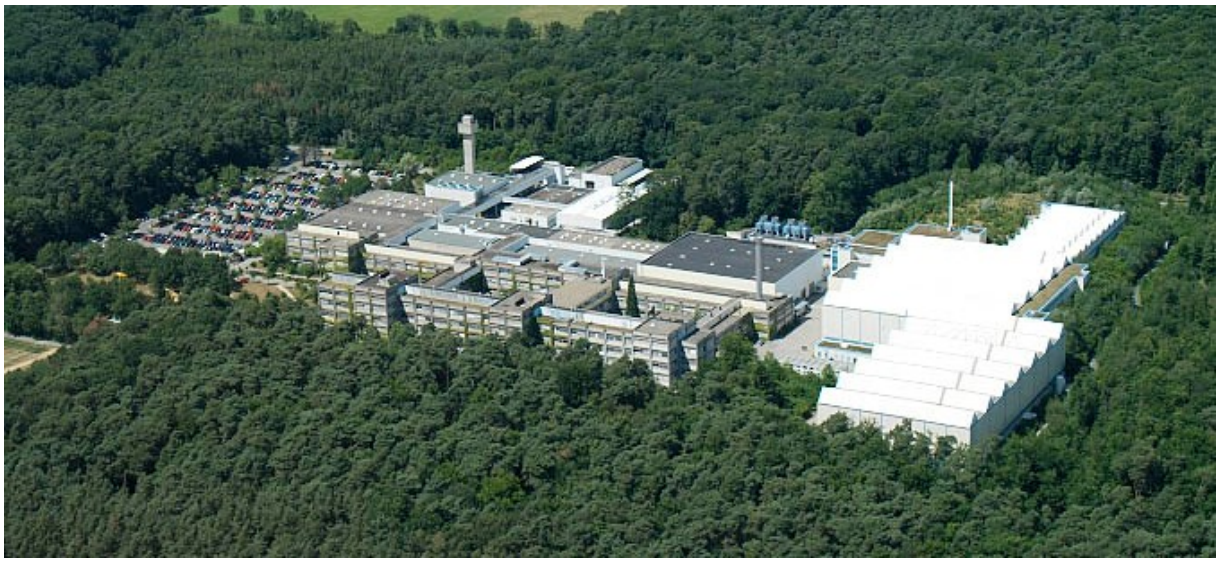

Abbildung 1: Luftaufnahme der GSI

Gegründet wurde das Helmholtzzentrum für Schwerionenforschung im Jahre 1969 unter dem Namen Gesellschaft für Schwerionenforschung (GSI) als Großforschungseinrichtung für Grundlagenforschung im Bereich der Kern- und Atomphysik. Gesellschafter sind das Land Hessen zu 10% und die Bundesrepublik Deutschland zu 90%. Die GSI hat mehr als 1000 Mitarbeiter, von denen etwa 200 Wissenschaftler und Ingenieure sind. Hinzu kommen ca. 1000 Forscher von Hochschulen und Forschungsinstituten aus der ganzen Welt, die die Anlagen der GSI nutzen.

An der GSI existiert eine in ihrem Aufbau weltweit einmalige Beschleunigeranlage für Ionenstrahlen. Diese besteht aus dem UNILAC (Universal Linear Accelerator), dem SIS (Schwerionensynchrotron) und dem ESR (Experimentierspeicherring). Im Linearbeschleuniger UNILAC werden die Ionen auf etwa 20% der Lichtgeschwindigkeit beschleunigt. Anschließend werden die vorbeschleunigten Ionen im ringförmigen SIS weiter beschleunigt und erreichen nach mehreren Hunderttausend Umläufen bis zu 90% der Lichtgeschwindigkeit. Von hier aus können die Ionen dann direkt zu den Experimenten geführt werden. Alternativ können die Ionen auch in den ESR geleitet werden, wo sie bei konstant hoher Geschwindigkeit mehrere Millionen bis Milliarden Umläufe vollführen und so für Experimente zur Verfügung stehen.

Die Forschung an der GSI widmet sich schwerpunktmäßig dem Erzeugen und Erforschen neuer Elemente. Hierzu werden geladene Atome (Ionen) mithilfe von elektromagnetischen Feldern in einen Strahl gebündelt und auf extrem hohe Geschwindigkeiten beschleunigt. Dieser Ionenstrahl trifft anschließend auf eine Materialprobe (Target), wobei die Atomkerne miteinander verschmelzen und zu einem stabilen Element zerfallen können. Bekannt geworden ist die GSI vor allem mit der Entdeckung und Erzeugung der Elemente mit den Ordnungszahlen 107 bis 112 (Bohrium, Hassium, Meitnerium, Darmstadtium, Röntgenium. Element 112, das erst kürzlich von der IUPAC anerkannt wurde, trägt seit dem 19. Februar 2010 den Namen Copernicium.)

Neben der Grundlagenforschung im Bereich der Kern- und Atomphysik wird auch in anderen Bereichen geforscht. Hierzu zählen unter Anderem Materialforschung, Biophysik und Strahlenmedizin. An der GSI wurde eine bahnbrechende neue Krebstherapie entwickelt, bei der Tumore im Kopfbereich mit Ionen bestrahlt werden. In einer Testphase wurden 407 Patienten erfolgreich behandelt. Erkenntnisse aus dieser Forschung flossen in die Konstruktion einer Beschleunigeranlage für die medizinische Strahlentherapie ein, die in Heidelberg gebaut und im November 2009 in Betrieb genommen wurde. Dort sollen bis zu 1000 Patienten pro Jahr behandelt werden.

Weitere Angebote der GSI sind diverse Ausstellungen und Vorträge, oder Bildungseinrichtungen wie ein Schülerlabor, in dem Schülerinnen und Schüler selbstständig an Experimenten arbeiten und dabei die faszinierende Welt der Physik kennenlernen können.

Die Zukunftspläne der GSI sehen eine Erweiterung der Beschleunigeranlage vor. Diese trägt den Namen FAIR (Facility for Antiproton and Ion Research) und soll bis 2016 fertiggestellt werden. Das Herzstück der neuen Beschleunigeranlage soll ein Doppelringbeschleuniger mit einem Umfang von 1100 Metern werden, an den sich ein komplexes System von Speicherringen und Experimentierstationen anschließt. Die Errichtungskosten für die geplante Anlage betragen etwa 1,2 Milliarde Euro (Stand 2005). Durch die neue Anlage verspricht sich die GSI eine Verbesserung der Strahlintensität und Strahlqualität, um somit neue Erkenntnisse im Bereich der Grundlagenforschung gewinnen zu können. Die folgende Grafik soll einen Überblick über die geplante, sowie die vorhandene Beschleunigeranlage geben.

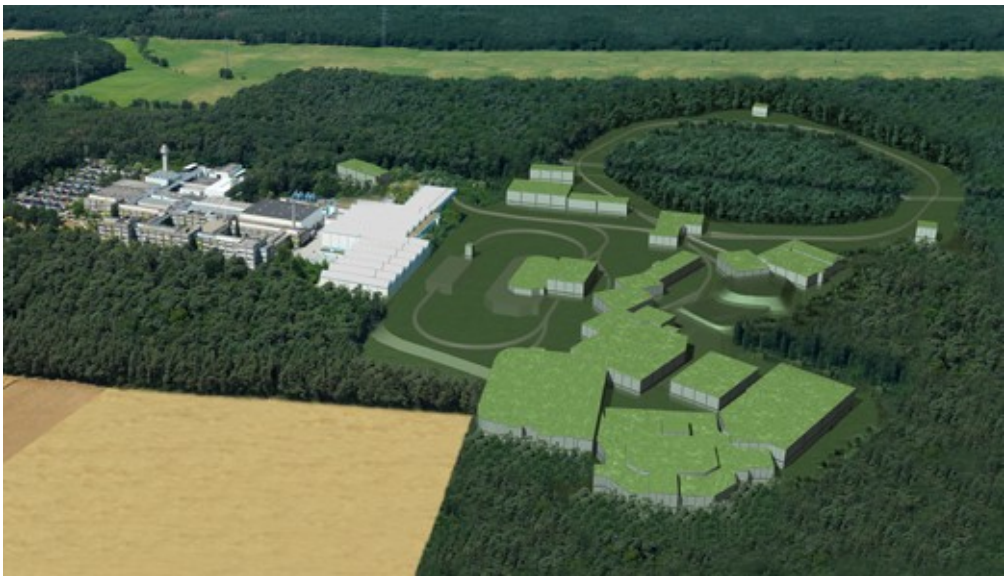

Abbildung 2: Montage der geplanten Beschleunigeranlage für FAIR

Motivation

LabVIEW existiert als Programmierumgebung schon seit dem Jahre 1986. Jedoch ist objektorientierte Programmierung erst seit LabVIEW Version 8.20, die im Jahre 2006 veröffentlicht wurde, verfügbar. Die objektorientierte Programmierung in LabVIEW (LVOOP) ist also noch vergleichsweise jung. Da LabVIEW sich in einigen Punkten, allem voran das Datenfluss-Paradigma, von herkömmlichen, textbasierten Programmiersprachen unterscheidet, soll ein mobiles Agentensystem als Basistest fungieren, um die Möglichkeiten und Restriktionen der objektorientierten Programmierung in LabVIEW auszuloten. Ein mobiles Agentensystem ist generisch, ereignisgesteuert, netzwerkverteilt und verfügt eventuell über Datenbankzugriff. Hierbei erlaubt es, Rückschlüsse auf Skalierbarkeit und Performanz einer Programmiersprache zu ziehen. Das mobile Agentensystem dient der vorbereitenden Untersuchung für anstehende, zukünftige Projekte, die sich aus der Erweiterung der bestehenden Beschleunigeranlage (FAIR) ergeben. Zusätzlich kann das mobile Agentensystem, sollte es hinsichtlich der zu untersuchenden Eigenschaften zu einem positiven Ergebnis kommen, als Grundlage für einen Ersatz des *CS-Framework*[1] dienen, das einen objektorientierten Ansatz künstlich auf LabVIEW aufsetzt.

[1]Nähere Informationen siehe: http://wiki.gsi.de/cgi-bin/view/CSframework/WebHome

LabVIEW

Allgemeines über LabVIEW

LabVIEW (Laboratory Virtual Instrument Engineering Workbench)[2] ist ein grafisches Programmiersystem von National Instruments, dessen Programmiersprache "G" genannt wird. Die Programmierung erfolgt nach dem Datenfluss-Paradigma und eignet sich besonders gut für Datenerfassung und –verarbeitung. LabVIEW Programme werden VIs, Virtuelle Instrumente, genannt. Ein VI besteht immer aus einem „Frontpanel“, einem „Blockdiagramm“ und einer „Connector Pane“. Das Blockdiagramm beinhaltet den Programmcode, der sich in LabVIEW ähnlich einem Stromlaufplan darstellt. Das Frontpanel beinhaltet Kontrollen und Indikatoren und dient als GUI[3]. Werden Kontrollen oder Indikatoren auf dem Frontpanel mit der Connector Pane verknüpft, stehen diese als Parameter für den Einsatz des VIs als SubVI[4] bereit.

Die Kontrollen und Indikatoren des Frontpanels sind mit Datenquellen und Datensenken des Blockdiagramms verbunden. So werden beispielsweise boolesche Variablen des Blockdiagramms auf dem Frontpanel durch eine Leuchtdiode repräsentiert, eine Zahl könnte durch einen Schieber dargestellt werden. Dies ermöglicht es dem LabVIEW Programmierer oftmals, eine übersichtliche grafische Benutzeroberfläche mit wenig Aufwand zu entwerfen. An dieser Stelle ist es jedoch wichtig zu betonen, dass die Werte der Frontpanel-Objekte und die verknüpften Blockdiagramm-Objekte durch die LabVIEW Runtime Engine[5] asynchron aktualisiert werden, was in einem Programm zu Race Conditions[6] führen kann.

Die Bezeichnungen für Frontpanel und Blockdiagramm symbolisieren die Analogie zu realen Messinstrumenten. Auch hier gibt es eine Frontplatte, auf der sich normalerweise die Bedien- und Anzeigeeinheiten befinden, während die Schaltkreise im Inneren die eigentlichen Funktionen übernehmen.

VIs lassen sich in LabVIEW beliebig verschachteln. Auch die von LabVIEW bereitgestellten Funktionen sind VIs, die sich gegebenenfalls sogar vom Benutzer verändern lassen. Letztlich basieren alle VIs auf so genannten Primitives, die die grundlegenden Funktionen implementieren.

Polymorphismus, also an den übergebenen Datentyp angepasste Funktionalität, wird von vielen VIs und Primitives genutzt. Auch ist es möglich, eigene Polymorphe VIs zu erstellen. Dies stellt nichts anderes dar, als eine Sammlung von VIs mit unterschiedlichen Ein- bzw. Ausgabetypen, aus denen der Compiler das passende VI auswählt.

---

[2] (Silmaril, 2010)
[3] Graphical User Interface, Benutzerschnittstelle
[4] Unterprogramm in LabVIEW
[5] Laufzeitumgebung
[6] (94.245.199.57, 2010)“Eine Konstellation, in der das Ergebnis einer Operation vom zeitlichen Verhalten bestimmter Einzeloperationen abhängt“

Programmieren in LabVIEW

Die Programmierung in LabVIEW erfolgt, indem innerhalb eines Hauptprogrammes die einzelnen Programmteile, zum Beispiel SubVIs oder Schleifenfunktionen, mittels sogenannter Drähte verbunden werden. Mit diesen Drähten werden Daten, also Werte eines bestimmten Datentyps, stets von einer Quelle zu mindestens einer Senke geleitet. Durch Drahtabzweige können auch mehrere Senken von einer Quelle Daten erhalten. Bei einem Drahtabzweig werden stets Kopien der Daten erzeugt. Die Ausführung eines Programmteiles erfolgt, sobald an allen beschalteten Eingängen, also Datensenken, Daten anliegen. Erst wenn die Ausführung des Programmteiles beendet ist, werden dessen Ausgänge mit Daten versorgt. Diese Ausgänge stehen als Datenquellen zur Verfügung, die mit Hilfe von Drähten mit weiteren Datensenken verbunden werden können. Ablaufsequenzen werden durch Datenabhängigkeiten definiert. Innerhalb eines VIs können mehrere Programmteile, so zum Beispiel Schleifen, parallel laufen. Bei Drahtabzweigen werden wie schon erwähnt Kopien von den Daten erstellt, wodurch die Senken automatisch datenunabhängig sind.

Von einem VI wird normalerweise nur eine Instanz erzeugt. Es wird also nur einmal Speicherplatz für die Daten des VIs durch den LabVIEW Compiler reserviert. Ein gleichzeitiger Aufruf dieser Instanz wird nicht erlaubt. Für VIs, die an unterschiedlichen Stellen aufgerufen werden sollen, ohne sich gegenseitig zu blockieren, existiert die Möglichkeit, sie als „reentrant[7]“ zu deklarieren. Hier wird, je nach Einstellung, mit sogenannten „Clones“ gearbeitet, LabVIEW minimiert hierbei die Anzahl der Instanzen auf die maximale Zahl der parallelen Zugriffe um Speicherplatz zu sparen, oder es wird für jedes Blockdiagrammelement des VIs eine separate Instanz gestartet. Ein gleichzeitiger Zugriff mehrerer Threads[8] auf eine Instanz eines VIs wird hierdurch ausgeschlossen.

Somit ist LabVIEW inhärent threadsicher[9] und multithreadingfähig[10].

Um datenflusskonform zwischen unterschiedlichen Threads[11] synchronisieren zu können, werden ereignisgesteuerte Methoden wie zum Beispiel Notifier[12], Queue[13], Okkurrenz oder Semaphore bereit gestellt. Diese Mechanismen arbeiten über Referenzen. So wird zum Beispiel beim Erzeugen einer Queue Speicherplatz für diese reserviert, und eine Referenz auf den Speicherbereich, eine „Queue Reference“ zurückgegeben. Diese kann dann über Drähte an verschiedene Threads geleitet werden. Auf die Funktionsweise einiger dieser Mechanismen wird zu einem späteren Zeitpunkt noch eingegangen.

Variablen, wie es sie in vielen Programmiersprachen gibt, existieren in LabVIEW nicht. Die sogenannten Variablen, die LabVIEW zur Verfügung stellt, repräsentieren ein Konstrukt aus asynchron aktualisierten Datenquellen und Datensenken, deren Verwendung zu Race Conditions führen kann. Deren Einsatz sollte deshalb möglichst vermieden werden.

Das Konzept, welches hinter Variablen steht, wiederspricht vollständig dem von LabVIEW verwendeten Datenflusskonzept: Werte eines bestimmten Datentyps bewegen sich entlang von

[7] Eintrittsinvariant
[8] Ein Ausführungsstrang innerhalb eines Prozesses
[9] Verschiedene Ausführungsstränge können ein VI parallel nutzen, ohne sich dabei zu behindern
[10] Verschiedene Ausführungsstränge innerhalb eines Prozesses können Parallel ausgeführt werden
[11] Das eigentliche Programm, in LabVIEW das „Top Level VI“
[12] Benachrichtigungen
[13] Gepufferte Warteschlangen

Drähten immer von einer Quelle zu Senken und werden bei Drahtabzweigen kopiert, wodurch weitere, datenunabhängige Ausführungsstränge entstehen können. Variablen hingegen sind Zeiger auf Speicherbereiche, wodurch sich sofort in LabVIEW unerwünschte Datenabhängigkeiten ergeben würden. Ohne ein Variablenkonzept existiert auch kein scope[14], der die lifetime[15] definiert. Der Sichtbarkeitsbereich von Daten in LabVIEW ist unbekannt. Speicher wird solange allokiert, wie er benötigt wird. Werden zum Beispiel Daten in ein Frontpanelobjekt kopiert, so existiert diese Kopie auch nach der Ausführung des VIs weiter.

(Http://www.ni.com, 2009) *In a pure theoretical data flow language, there would be a separate copy on every individual wire, because every wire is an independent computation unit. Of course, that would be inefficient to actually implement, so* ***LabVIEW's compiler optimizes the number of copies.*** *But the principle is the same: data lives for a long time, sometimes outliving the program that generated that data.*

[14] Sichtbarkeitsbereich einer Variablen

[15] Lebenszeit einer Variablen. Die Zeit, in der Speicherplatz für die Verwendung der Variablen reserviert ist

Objektorientiertes Programmieren in LabVIEW

Seit Version 8.20 existiert in LabVIEW ein Klassenkonzept, dass die objektorientierte Programmierung, kurz „LVOOP“, ermöglicht. Bedingt durch das Datenflusskonzept gibt es jedoch grundlegende Unterschiede gegenüber konventionellen Programmiersprachen.

LabVIEW Klassen verfügen ebenso wie Klassen anderer Programmiersprachen, wie zum Beispiel Java oder C++, über Attribute und Methoden. Attribute einer Klasse werden in LabVIEW durch Kontrollen dargestellt, die in einem privaten Cluster zusammengefasst sind. Nur Methoden, VIs, der eigenen Klasse können auf die Attribute zugreifen.

Methoden haben einen Access Scope, eine Zugriffsberechtigung, der regelt, welche VIs die Methoden aufrufen dürfen. Der Access Scope kann public[16], private[17], protected[18] und seit LabVIEW Version 2009 auch community[19] sein.

LabVIEW beherrscht Einfachvererbung, eine Kind-Klasse kann also jeweils nur von einer Klasse erben. Die Klassenhierarchie weist somit stets eine Baumstruktur auf, wobei National Instruments die Konvention vorgibt, dass Ahnenklassen weiter oben, Nachfahren weiter unten stehen sollen. Die LabVIEW Objekt Klasse ist hierbei die „ultimative Ahnenklasse“, die immer an oberster Stelle in der Klassenhierarchie steht. Wird ein Objekt an eine Senke mit dem Datentyp einer übergeordneten Klasse übergeben, so findet automatisch eine Typenwandlung statt. Eine Senke des Typs LabVIEW Objekt akzeptiert somit beliebige Objekte. Bei der Typenwandlung bleiben dennoch alle Objektdaten erhalten. Mittels *to more specific* VI kann später explizit wieder der originale Datentyp aus dem Objekt erstellt werden. Dies ist notwendig, falls Klassenmethoden auf das Objekt angewendet werden sollen, da keine automatische Typenwandlung auf spezialisierte Klassen erfolgt.

Dynamic dispatch genannte VIs repräsentieren in LabVIEW virtuelle Methoden, die von Kind-Klassen überschrieben werden können. Hierfür wird in einer Kind-Klasse ein VI mit gleichem Namen erstellt. Für Klassenmethoden ist kein Polymorphismus vorgesehen, weshalb das VI über eine identische Connector Pane verfügen muss wie das zu überschreibende. Zur Laufzeit wird automatisch das VI ausgeführt, welches zur Klasse des übergebenen Objektes gehört.

Ein Objekt ist ein passives Datenpaket, ein Wert vom Datentyp seiner Klasse. Objekte werden über Drähte von Quellen zu Senken geleitet und bei Drahtabzweigen dupliziert. In LabVIEW sind Objekte also keine Entitäten[20] und besitzen keine definierte lifetime. Aus diesem Grund existieren in LabVIEW keine Konstruktoren und Destruktoren.

Da Objekte streng nach den Regeln des Datenflusskonzeptes verarbeitet werden, ist auch die LVOOP intrinsisch multithreadingfähig und threadsicher.

[16] Öffentlich, das VI kann von jedem VI aus aufgerufen werden
[17] Privat, das VI kann nur von Methoden der eigenen Klasse aufgerufen werden
[18] Geschützt, das VI kann von Methoden der eigenen Klasse und von Methoden von Kind-Klassen aufgerufen werden
[19] Das VI kann von Methoden der eigenen Klasse und von Freunden aufgerufen werden.
[20] Eindeutig identifizierbares Informationsobjekt, eine bestimmbare Instanz

Agenten und mobile Agenten

(Mattern) *"Mobile Agenten sind autonome Programme, die in einem Netz von Rechnern umherwandern und dabei im Auftrag eines Nutzers Dienste verrichten."*

Ein Agent ist eine Software, die eigenständig eine ihr zugewiesene Aufgabe erfüllt und dazu gedacht ist, dem Benutzer Arbeit abzunehmen. Dies könnte zum Beispiel eine Software sein, die automatisch Messwerte aufnimmt und auswertet, ohne dabei ein regelmäßiges Eingreifen des Benutzers zu erfordern. Zusätzlich kann ein Agent über kommunikative Fähigkeiten verfügen, die es ihm ermöglichen, mit seiner Umgebung zu interagieren. Ein mobiler Agent bringt die Fähigkeit mit, seinen Ausführungsort zu wechseln. Hierzu unterbricht er seine aktuelle Arbeit und reist, mitsamt seiner Daten, zu einem anderen System, um dort mit der Arbeit fortzufahren. Diese Möglichkeit bringt diverse Vorteile mit sich. Wird zum Beispiel der Rechner, auf dem ein Agent zur Datenauswertung gerade läuft, heruntergefahren, so kann der Agent mitsamt seinen Daten zu einem anderen Rechner geschickt werden, um dort mit seiner Auswertung fortzufahren.

Voraussetzung für ein mobiles Agentensystem ist ein Netzwerk, in dem Hostprogramme die Schnittstelle für Agenten repräsentieren. Die Aufgabe eines Hosts besteht unter Anderem in der Bereitstellung von Arbeitsumgebungen und einer Verwaltung für die Agenten, während diverse Sicherheitskriterien berücksichtigt werden müssen.

Anforderungen an ein mobiles Agentensystem in LabVIEW

Um ein Agentensystem in LabVIEW realisieren zu können, bestehen ein paar grundlegende Anforderungen, die im Folgenden kurz zusammengefasst werden.

Kommunikationsschnittstellen und ein Kommunikationssystem

Ein Agent soll in der Lage sein, über kommunikative Fähigkeiten zu verfügen. Ist ein Agent auf einem Host gestartet sollen sowohl der zuständige Host, der Anwender sowie andere Agenten in der Lage sein, mit ihm zu kommunizieren. Hierfür müssen Schnittstellen und ein Kommunikationssystem bereit gestellt werden.

Entitäten[21]

Ein Agent ist eine Software mit eigenständigen Aufgaben und Fähigkeiten. Er wird durch ein Objekt repräsentiert. Die Aufgaben und Fähigkeiten die ein Agent besitzt, erfordern es, dass man ihn als Entität behandeln muss.

Passive Datenobjekte stehen agierenden Softwareprogrammen gegenüber

Wie schon erwähnt handelt es sich bei Objekten um passive Datenobjekte. Ein Agent stellt jedoch eine eigenständige Software dar. Es muss also eine Arbeitsumgebung existieren, die es ermöglicht, diese passiven Datenobjekte zu aktivieren und diese aktivierten Objekte als eigenständiges Programm zu verwenden.

Migration

Mobile Agenten kennzeichnen sich dadurch aus, dass sie zwischen verschiedenen Hosts migrieren können. Die Repräsentation durch ein passives Agentenobjekt ermöglicht es, die eigentliche Migration durch Übertragung einer Kopie des aktuellen Zustandes an ein Zielsystem zu implementieren, wobei anschließend die lokale Arbeit zu beenden ist. Hier bedarf es sowohl eines geeigneten Verfahrens um sicherzustellen, dass eine Migration erfolgreich war, als auch einer Möglichkeit einen unerwünschten Zugriff auf ein Agentenobjekt während der Übertragung zu verhindern.

[21] Eindeutig identifizierbare Instanz

Erweiterbarkeit des Agentensystems

Das mobile Agentensystem soll in erster Linie Basisfunktionen bereitstellen, die von Entwicklern je nach Bedürfnis erweitert werden können. Dies setzt voraus, dass ein bereits installiertes System über Möglichkeiten verfügt, ihm unbekannte Objekte zu empfangen und diese zu aktivieren, also mit diesen zu arbeiten.

Sicherheit

Softwareagenten dieses Agentensystems sollen grundsätzlich in der Lage sein, zwischen verschiedenen Hosts transferiert zu werden. Ein Host startet dabei eine Software, deren Funktion er nicht kennt. Da eine unbekannte Software mitunter auch unerwünschte und schädliche Funktionen haben kann muss es bezüglich der Ausführung Restriktionen geben können.

Die HGF Basisklassen[22]

Um Datenfluss-Objekte als Entitäten behandeln zu können erfordert die native objektorientierte Implementierung in LabVIEW einen erhöhten Programmieraufwand. Sie bleibt aber intrinsisch multithreadingfähig und threadsicher.

Bestehende Entwurfsmuster[23] beschreiben, wie bestimmte Objekte und Klassen interagieren, um ein wiederkehrendes Problem in der Softwarearchitektur zu lösen. Diese Entwurfsmuster basieren jedoch auf dem herkömmlichen Verständnis von Klassenobjekten als Entitäten. Für LVOOP müssen diese neu überdacht werden, um sowohl das eigentliche Problem zu lösen, als auch datenflusskonform anwendbar zu sein.

Was sind die HGF Basisklassen

Die HGF Basisklassenbibliothek stellt LVOOP Klassen, die HGF Basisklassen, bereit, die gängige Entwurfsmuster datenflusskonform implementieren und die Behandlung von LabVIEW Objekten als Entitäten erlauben. Hierbei wird streng unterschieden zwischen Klassen, die Datenobjekte repräsentieren (Beispiel: HGF_Visitable), und Klassen, die jene aktivieren (Beispiel: HGF_ThreadPool). Implementiert wurden die HGF Basisklassen von Dr. Holger Brand im Rahmen seiner Arbeit an der GSI. Folgende Bestandteile der Basisklassen werden näher erläutert:

- HGF_Factory (Fabrikmethode)
- HGF_FGV (Functional Global Variable)
- HGF_Visitable und HGF_Visitor (Besucher)
- HGF_Reference (Referenz)
- ThreadPool Entwurfsmuster
- HGF_Events (Ereignisse)
- HGF_ThreadPool, HGF_ThreadWorker, HGF_ThreadTask (ThreadPool Pattern)
- HGF_SV und HGF_DSC

Des Weiteren sieht das Basisklassensystem eine Klasse „Base“ vor. Diese Klasse enthält Attribute, die alle Klassen betreffen und dient dazu, auch nachträglich Attribute und Methoden hinzufügen zu können.

[22] HGF steht für „Helmholtz“, „GSI“ und „FAIR“
[23] Vorlagen, um wiederkehrende Probleme in der Softwarearchitektur zu lösen

HGF_Factory

Die HGF_Factory Klasse implementiert die Fabrikmethode. Diese dient der generischen Erzeugung von initialisierten Objekten zur Laufzeit.

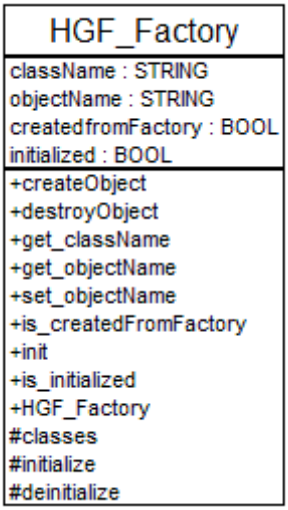

Abbildung 3: Die HGF_Factory-Klasse

Hierzu verfügt die HGF_Factory Klasse über ein *createObject* VI. Diesem werden als Parameter der Klassenname, eventuell ein Objektname und die Initialisierungsdaten für das neue Objekt übergeben. Das *createObject* VI ruft nun das *classes* VI auf, welches ein Objekt der entsprechenden Klasse zurückgibt.

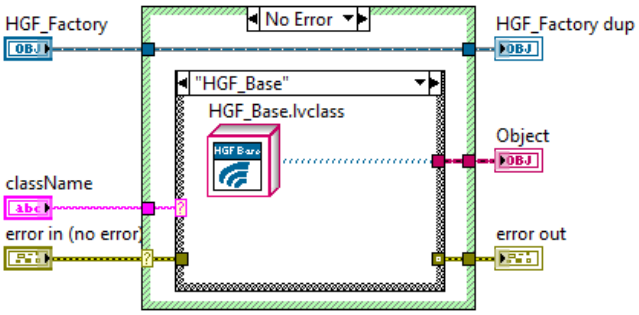

Abbildung 4: Ausschnitt Blockdiagramm Classes.vi

Das *classes* VI ist ein dynamic dispatch VI. Die HGF_Factory stellt hierin alle Objekte der Basisklassen bereit. Über ein Case-Diagramm wird das entsprechende Objekt ausgewählt und zurückgegeben. Für den Fall, dass der Klassenname nicht auffindbar ist wird er als Pfad interpretiert und das *classes* VI versucht mit Hilfe des *Get LV Class Default Value*.vi die Klasse von der Festplatte zu laden. Dies ermöglicht es der Factory, auch Objekte bisher unbekannter Klassen zu erzeugen. HGF_Factory-Kind-Klassen überschreiben dieses VI, wodurch sich zusätzliche Objekte ergänzen lassen, die anschließend über die Wahl der richtigen Fabrik erzeugt werden können. Innerhalb einer Applikation sollte es eine Fabrik geben, die alle nötigen Objekte der Applikation beinhaltet. Hierdurch wird sichergestellt, dass alle nötigen Klassen für die Applikation gelinkt werden.

Das Entwurfsmuster „Fabrikmethode“ wird für LabVIEW noch ergänzt. Da Konstruktoren nicht existieren, übernimmt die HGF_Factory Klasse zusätzlich die Aufgabe, die Instanz zu initialisieren. Hierfür wird das dynamic dispatch VI *initialize* bereitgestellt. Dieses kann von allen HGF_Factory-Kind-Klassen überschrieben werden und kann eine spezialisierte Initialisierungsroutine für Instanzen dieser Klasse enthalten. Durch Aufrufen der *Call Parent VI* Methode wird hierbei die Initialisierungsroutine der übergeordneten Klassen ebenfalls aufgerufen. Die HGF_Factory Klasse steht in der Klassenhierarchie der HGF Basisklassenbibliothek direkt unterhalb der LabVIEW Objekt-Klasse. Alle weiteren Klassen dieser Bibliothek haben also von HGF_Factory geerbt, und können somit eine eigene Initialisierungsroutine implementieren. Da Klassenmethoden nicht polymorph sein dürfen muss jedes *initialize* VI über die gleiche Connector Pane verfügen. Um dennoch unterschiedliche Daten für die Initialisierung verwenden zu können werden diese als Variant übertragen. Variant ist ein LabVIEW Datentyp, den man mit verschiedenen Attributen versehen kann. Attribute sind dabei beliebige Datentypen, die mit Wert und einem Namen gesetzt werden und über den Namen und einen Standardwert, welcher der Typenfestlegung dient, wieder ausgelesen werden können.

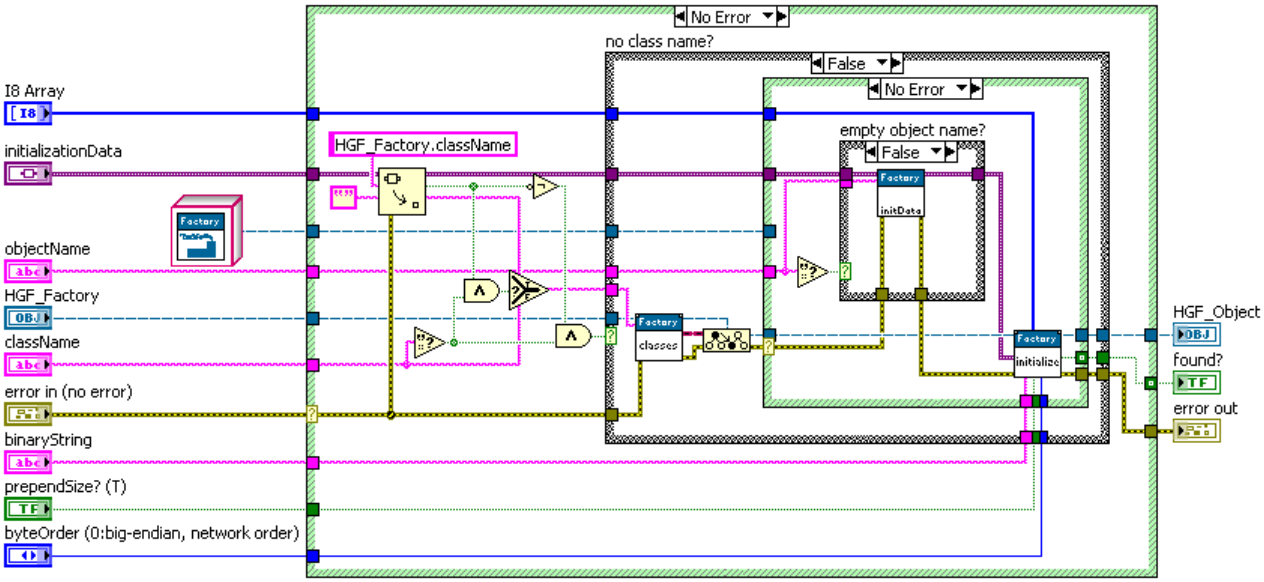

Abbildung 5: HGF_Factory createObject.vi

Die Factory stellt zusätzlich noch ein *destroyObject* VI bereit. Dieses übernimmt Funktionalitäten des in LabVIEW nicht existenten Destruktors. Das VI ist ein dynamic dispatch VI, welches die Aufgabe übernehmen soll, ein Objekt kontrolliert zu „zerstören“. Erzeugt ein Objekt zum Beispiel eine Queue, so ist es ratsam, diese wieder zu zerstören, wenn sie nicht mehr benötigt wird. Dies wird in anderen Programmiersprachen häufig in einem Destruktor realisiert. Anders als ein Destruktor muss dieses VI explizit aufgerufen werden. Zu betonen ist, dass diese Methode bedingt durch das Datenflusskonzept immer nur auf eine Kopie eines Objektes angewendet wird, und somit nicht geeignet ist, alle Kopien eines Objektes zu zerstören.

Funktionale globale Variablen (FGV)

Eine häufig genutzte Möglichkeit, um fehlende Variablen in LabVIEW zu ersetzen, ist die Verwendung von funktionalen globalen Variablen. Bei diesen sogenannten Variablen handelt es sich um ein separates VI, welches eine While-Schleife mit einem nicht initialisierten Shift-Register beinhaltet. Dieses wird nach einer Iteration beendet wird. Ein Shift-Register dient dem Transport von Daten von einer Iteration zur nächsten, wobei der Datentyp des Shift-Registers durch die Verbindung mit einer Datenquelle festgelegt wird. Bei einem nicht initialisierten Shift-Register steht bei Mehrfachaufrufen, also zum Beispiel bei verschachtelten Schleifen oder bei Verwendung innerhalb eines SubVIs das mehrfach ausgeführt wird, der Wert der letzten Iteration zur Verfügung, auch wenn die Schleifenfunktion zwischendurch beendet wurde.

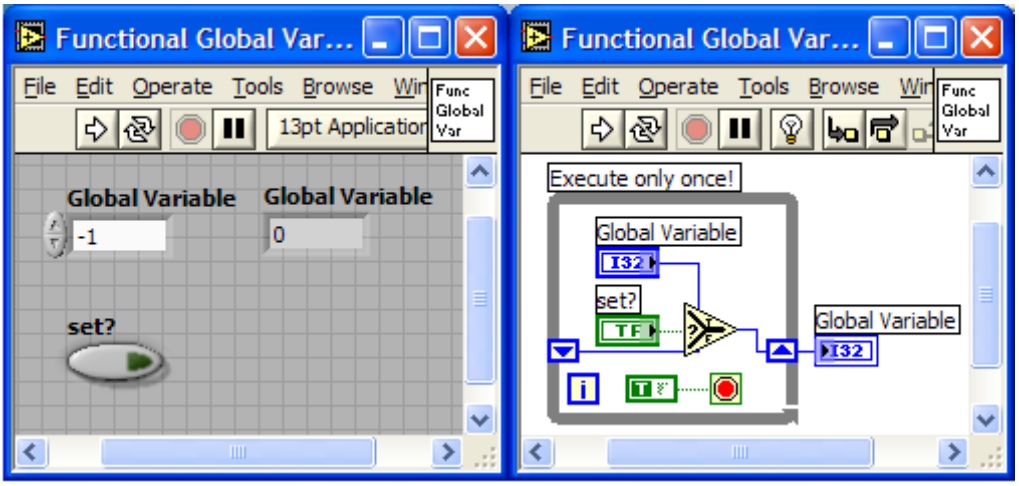

Abbildung 6: Globale Variablen 2. Art (Brand H. , LabVIEW Techniken, 2005)

Abbildung 6 zeigt die Implementierung einer funktionalen globalen Variablen. Verwendet wird eine solche funktionale globale Variable, indem sie als SubVI eingesetzt wird. Beim Aufruf wird als Parameter das boolesche **set?** und, in diesem Fall, ein Integerwert **Global Variable** übergeben. Ist **set?=**true wird der Integerwert im Shift-Register gespeichert. Ist **set?**=false bleibt der Wert der letzten Iteration erhalten. Anschließend wird der Wert des Shift-Registers zurückgegeben.

Da ein SubVI nur von einem VI gleichzeitig aufgerufen werden kann, ist hierbei der Ausschluss von konkurrierenden Aufrufen gewährleistet.

Mit funktionalen globalen Variablen ist es also möglich, Daten zwischen zu speichern und wieder auszulesen. Da Objekte Datenpakete sind lassen Sie sich auf gleiche Weise zwischenspeichern.

Soll mit den Daten gearbeitet werden, so zeigen sich schnell die Grenzen des bisherigen Konzeptes. Um eine Verarbeitung zu ermöglichen wird zuerst der aktuelle Zustand ausgelesen. Ist die Verarbeitung beendet, so wird der alte Wert der funktionalen globalen Variablen mit dem neuen Wert überschrieben. Zwar verhindert die Verwendung eines SubVIs den parallelen Zugriff mehrerer Threads, nicht jedoch den Zugriff während der Verarbeitungszeit. Um hierbei Race Conditions zu vermeiden existieren zwei Möglichkeiten. Eine Möglichkeit ist es, den Zugriff auf die funktionale globale Variable über Semaphoren zu kontrollieren. Hierbei wird der Aufruf der FGV explizit gesperrt und wieder freigegeben.

Eine andere Möglichkeit ist es, die Verarbeitung innerhalb der funktionalen globalen Variablen auszuführen, wodurch Race Conditions automatisch vermieden werden. Um hierfür eine objektorientierte Möglichkeit zu bieten, wurden die HGF_FGV Klassen implementiert.

HGF_FGV

Die HGF_FGV Klassen sind in einer Bibliothek (*HGF_FGV.lvlib*) zusammengefasst und erweitern die HGF Basisklassenbibliothek. Es wird ein VI bereitgestellt, welches die funktionale globale Variable enthält.

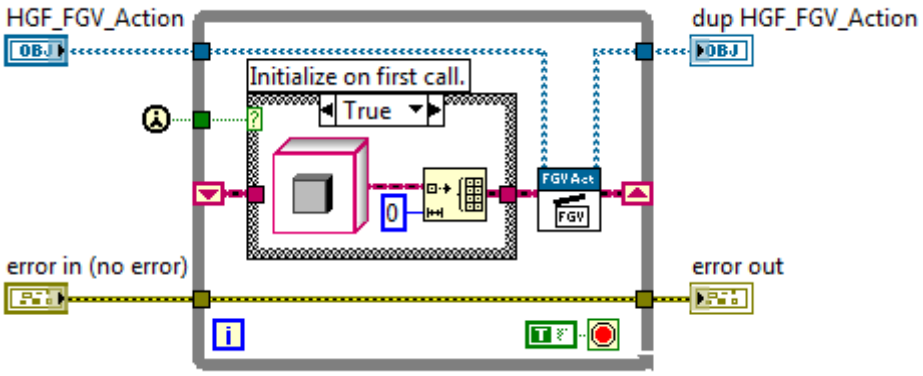

Abbildung 7: Blockdiagramm der objektorientierten funktionalen globalen Variablen

Abbildung 7 zeigt das Blockdiagramm mit der funktionalen globalen Variablen. Beim Erstaufruf[24] wird mittels Case-Diagramm ein leeres Array vom Typ LabVIEW Objekt erzeugt. Bei jeder weiteren Iteration wird der Wert des Shift-Registers durchgereicht. Bei einem Aufruf der funktionalen globalen Variablen wird ein Objekt der HGF_FGV_Action Klasse übergeben und das öffentliche *doaction* VI, welches von dieser Klasse bereitgestellt wird, mit der Objektliste als Parameter aufgerufen. Hierin wird die geschützte dynamic dispatch Methode *action* aufgerufen. Kind-Klassen von HGF_FGV_Action implementieren die eigentliche Aktion, die auf das Objekt Array angewendet wird. Je nachdem welche Kindklasse an die FGV übergeben wird, wird also eine unterschiedliche Aktion ausgeführt.

Zwei Typen von Aktionen sind hierbei denkbar. Aktionen, die auf die Liste und Aktionen, die auf die Objekte selbst angewendet werden sollen. Für Listenoperationen existieren Kindklassen, die es ermöglichen, Objekte hinzuzufügen, zu entfernen, auszutauschen oder abzufragen. Hierbei können gezielt einzelne Objekte, oder aber auch das gesamte Array verwendet werden.

Aktionen, die auf Objekte angewendet werden, sollten im Kontext einer generischen Verarbeitung an dieser Stelle nicht explizit implementiert werden. Um hierfür eine Lösung zu bieten, wird auf das Besucher Muster zurückgegriffen, welches im Folgenden beschrieben wird.

[24] FirstRun wird von LabVIEW zur Verfügung gestellt und ist nur beim ersten Aufruf innerhalb eines VIs True, ansonsten ist es False

HGF_Visitable und HGF_Visitor

Die Klassen HGF_Visitable und HGF_Visitor implementieren das Besucher Muster (englisch: Visitor Pattern). Dieses Entwurfsmuster dient dazu, Verarbeitungsalgorithmen von den Objektdaten, auf die sie angewendet werden sollen, zu entkoppeln und so entfernte Methodenaufrufe ausführen zu können. HGF_Visitable repräsentiert die Objekte auf die zugegriffen werden soll, HGF_Visitor dient als „Besucher“, welcher Methodenaufrufe ausführt.

Das UML Diagramm zeigt die beiden Bestandteile des Besuchermusters.

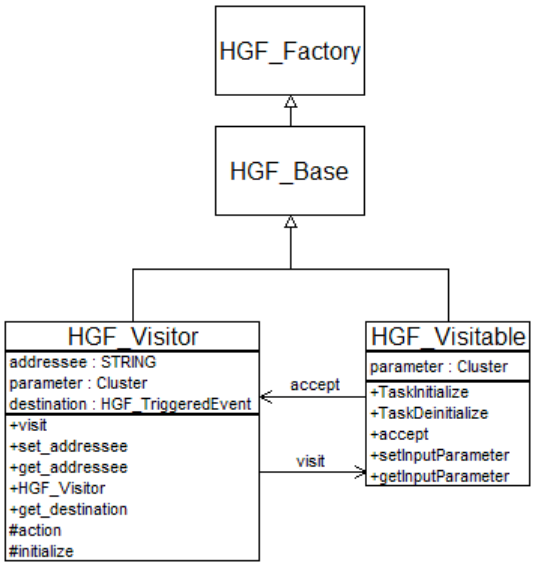

Abbildung 8: UML Klassendiagramm des imlementierten Besuchermusters

HGF_Visitable verfügt über eine *accept* Methode, die ein HGF_Visitor Objekt als Eingabeparameter hat. Diese Methode ruft die öffentliche *visit* Methode der HGF_Visitor Klasse auf, der das HGF_Visitable Objekt als Parameter übergeben wird. Diese *visit* Methode ruft nun die geschützte dynamic dispatch Methode *action* auf, welcher wiederum das HGF_Visitable Objekt übergeben wird. Im *action* VI können nun alle öffentlichen VIs der HGF_Visitable Klasse aufgerufen werden. Die Methodenaufrufe der HGF_Visitable Klasse finden somit innerhalb des *action* VIs der HGF_Visitor Klasse statt.

Eine Klasse, die besuchbar sein soll (zukünftig Visitable genannt), muss von HGF_Visitable erben und kann zusätzliche Schnittstellen, öffentliche VIs, bereitstellen. Besucher, Klassen die von HGF_Visitor geerbt haben, überschreiben das *action* VI und implementieren dadurch eigene Methodenaufrufe, wofür sie innerhalb Ihrer Attribute nötige Parameter mitbringen können. Hierbei kann ein fertiger Algorithmus als Schnittstelle bereitgestellt werden. Der Besucher dient also entweder lediglich als Wrapper oder kann, durch Kombination mehrerer Methodenaufrufe innerhalb des *action* VIs, einen der Visitable Klasse unbekannten Algorithmus implementieren. Jeder Besucher kann hierbei nur einmal das *action* VI überschreiben und damit auch nur einen Algorithmus implementieren. Für jeden Algorithmus ist somit ein separater Besucher zu programmieren.

Mit Hilfe dieses Entwurfsmusters ist es möglich, über Thread- und Prozessgrenzen hinweg, mit Datenobjekten zu arbeiten. Hierfür müssen Methoden vorgesehen werden, mit denen ein Besucher übertragen werden kann, wofür sich zum Beispiel Ereignismechanismen (siehe hierzu HGF_Event) eignen. Um jedoch einen Besucher auf ein besuchbares Objekt anzuwenden, muss die *accept* Methode explizit aufgerufen werden. Hieraus ist ersichtlich, dass es sich auch beim Besucher um ein passives Objekt handelt, welches den auszuführenden Algorithmus zur Verfügung stellt.

Das „Besucher“ Entwurfsmuster besitzt sowohl Vor- als auch Nachteile. Durch Verwendung dieses Musters erreicht man eine gute Erweiterbarkeit, ist jedoch an die öffentlichen Schnittstellen der Visitable Klassen gebunden. Für jeden Algorithmus muss ein separater Besucher geschrieben werden. Ist eine Veränderung der Schnittstellen nötig, so müssen auch alle zugehörigen Besucher überarbeitet werden.

Innerhalb der objektorientierten FGV wird ebenfalls das Besuchermuster angewendet. Hierzu existieren Kind-Klassen von HGF_FGV_Action, die einen Besucher in ihren Attributen mitbringen. Werden sie durch die FGV aufgerufen, so wird, je nach verwendeter Kind-Klasse, der Besucher auf alle Objekte angewendet oder mit Hilfe von Listenoperationen die Anwendung auf einzelne Objekte beschränkt.

HGF_Reference

Das „Besucher“ Entwurfsmuster ermöglicht es, mit einem Objekt in einem anderen Prozessen zu interagieren, ohne direkten Zugriff auf dieses zu haben. Um jedoch auf ein und dasselbe Objekt von verschiedenen Threads und Prozessen aus zugreifen zu können, kann man entweder auf die FGV zurückgreifen, oder die HGF_Reference Klasse verwenden. Bei Nutzung der FGV bleibt das Objekt innerhalb der FGV, es wird also von unterschiedlichen Stellen aus mittels Besuchern mit dem Objekt gearbeitet. Bei Verwendung der Methoden der HGF_Reference Klasse hat jeder Prozess, beziehungsweise jeder Thread, direkten Zugriff auf das Objekt selbst.

Um über Prozessgrenzen hinaus Daten synchronisieren zu können, existieren in LabVIEW mehrere Ereignismechanismen. Um eine referenzierbare, datenflusskonforme Entität darstellen zu können, wird auf eine Queue zurückgegriffen. Eine Queue ist ein Ereignismechanismus in LabVIEW, der es ermöglicht, Daten an einer referenzierten Speicherstelle zu puffern.

Die HGF_Reference Klasse bildet insofern eine Ausnahme, dass ihre Objekte nicht über die Factory erzeugt werden. Die Klasse stellt selbst ein *create* VI bereit, dem ein LabVIEW Objekt und ein Name übergeben werden kann. Dieses VI erzeugt eine neue Queue der Länge 1, mit dem Datentyp LabVIEW Objekt und dem übergebenen Namen. Der Name und die Referenz auf die Queue werden in den Objektattributen gespeichert und der Queue wird das übergebene Objekt angehängt. Da alle Klassen in LabVIEW letzten Endes von LabVIEW Objekt geerbt haben, kann ein beliebiges Objekt in dieser Queue zwischengespeichert werden. Wird nun das HGF_Reference Objekt kopiert, so wird lediglich die Referenz auf die Queue kopiert. Dies erlaubt den Zugriff auf Instanzen einer Klasse von verschiedenen Prozessen aus.

Da die Queue die Länge 1 besitzt ist sichergestellt, dass sich immer nur eine Instanz darin befinden kann. Die Verwendung der Queue bringt sowohl Vor- als auch Nachteile mit sich. Um mit dem Objekt zu arbeiten, muss es zuerst aus der Queue entfernt werden. Hierfür stellt die HGF_Reference Klasse ein *checkout* VI bereit. Anschließend muss das Objekt mit Hilfe des ebenfalls bereitgestellten *checkin* VIs wieder in der Queue zwischengespeichert werden. Da das Auschecken nur erfolgreich ist wenn sich ein Objekt in der Queue befindet, ist ein Mehrfachzugriff auf das Objekt ausgeschlossen. Da allerdings keine zusätzlichen Sicherungsverfahren vorgesehen sind liegen hier auch mögliche Fehlerquellen. Tritt nach einem erfolgreichen Auschecken ein Fehler auf, muss der Programmierer dafür sorgen, dass das Objekt dennoch nicht verloren geht und wieder eingecheckt wird. Desweiteren darf ein Einchecken nur nach einem erfolgreichen Auschecken erfolgen, da sonst das ursprüngliche Objekt durch ein anderes ersetzt wird. Ist ein Objekt gerade ausgecheckt, so wird es auch weiterhin datenflusskonform behandelt, also zum Beispiel bei Drahtabzweigen kopiert.

Diese Probleme können vermieden werden, wenn eine Klasse alle eigentlichen Methoden als privat deklariert, und den Zugriff auf diese Methoden durch öffentliche VIs ermöglicht, die mit einer HGF_Reference als Parameter aufgerufen werden können.

ThreadPool Entwurfsmuster

Die Klassen HGF_ThreadPool, HGF_ThreadWorker und HGF_ThreadTask implementieren das ThreadPool Entwurfsmuster.

Ein Threadpool dient dazu Tasks abzuarbeiten. Hierfür stellt er Threads bereit. Typischerweise stehen weniger Threads zur Verfügung als Aufgaben zu erledigen sind, weswegen diese in einer Queue gehalten werden. Hat ein Thread (Abbildung 9 grünes Quadrat) einen Task (Abbildung 9 Kreise) abgearbeitet, wird dieser an eine Queue für beendete Tasks geschickt und der Thread beginnt mit der Verarbeitung der nächsten in der Queue wartenden Aufgabe. Ist die Queue abgearbeitet, so wird der Thread entweder beendet, oder er wartet auf den nächsten kommenden Task.

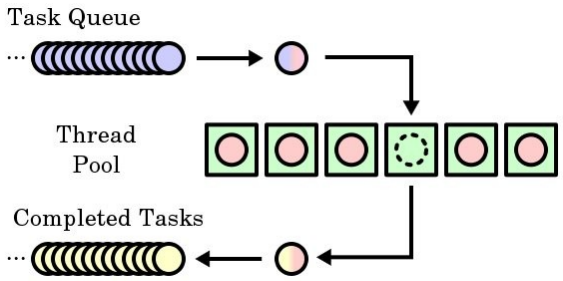

Abbildung 9: Schema ThreadPool (http://upload.wikimedia.org/wikipedia/commons/0/0c/Thread_pool.svg)

In den HGF Basisklassen nimmt der ThreadPool eine wichtige Stellung ein. Bisher wurde gezeigt, dass Objekte passive Datenelemente sind. Sie sind also nicht in der Lage, eigenständig Methoden aufzurufen. Ein Objekt, das eine bestimmte Aufgabe erledigen soll, muss also explizit aufgerufen werden. Um dies zu ermöglichen, wurde das ThreadPool Pattern in den HGF Basisklassen realisiert.

Ein Thread im Sinne des ThreadPool Musters ist von Natur aus aktiv. Nach dem Datenflusskonzept ist der einzige aktive Prozess allerdings das TopLevel VI. Für die Realisierung des ThreadPool Musters wurde also eine Ausnahme im Bezug auf das Datenflussparadigma gemacht. Der ThreadPool startet hierbei einen separaten Prozess, der unabhängig vom TopLevel VI abläuft. Um hierbei in der Lage zu sein, datenflusskonform zwischen verschiedenen Prozessen synchronisieren und Daten schicken zu können, wurde ein Objektorientierter Ereignismechanismus implementiert, der im Folgenden beschrieben wird.

HGF_Event

In LabVIEW existieren verschiedene Mechanismen zur Ereignissteuerung. Hierzu zählen Trigger gesteuerte Ereignisquellen wie Queues, Notifier und Okkurrenzen, aber auch interne Ereignisse wie Wait oder Rendezvous. Diese Ereignismechanismen dienen der Synchronisierung unterschiedlicher Prozesse und Threads, sowie der Übertragung von Daten zwischen verschiedenen Prozessen und Threads. Die HGF_Event Klassen gliedern diese Eventmechanismen datenflusskonform in das objektorientierte HGF Basisklassensystem ein.

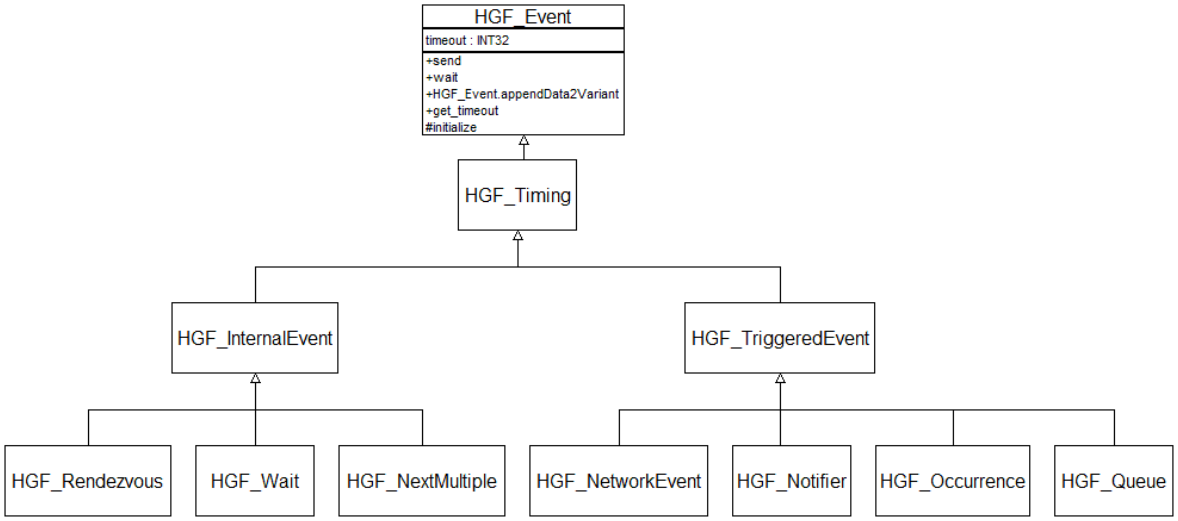

Abbildung 10: UML Klassendiagramm der HGF_Event-Klasse und ihrer Kind-Klassen

Abbildung 10 zeigt eine Übersicht über die HGF_Event Klassen. Die Event Basisklasse stellt die dynamic dispatch VIs *send* und *wait* zur Verfügung. Diese werden von den Kind-Klassen überschrieben um den jeweiligen Mechanismus zu implementieren. Der LabVIEW Compiler wählt je nach Objekt zur Laufzeit das richtige Overwrite-VI aus. Die generische Implementierung der HGF_Event-Klassen erlaubt es auch, verschiedene Ereignismechanismen zu kombinieren.

HGF_Timing verfügt in seinen Attributen über ein LabVIEW Objekt. Dieses dient denjenigen Kind-Klassen, die in der Lage sind, Objekte zu übertragen (zum Beispiel HGF_Notifier, HGF_Queue und HGF_Wait), als Standard-Rückgabeobjekt bei einer Zeitüberschreitung und kann bei der Initialisierung eines konkreten Events mit einem beliebigen Objekt überschrieben werden.

Beispielhaft wird die Implementierung des Ereignismechanismus Notifier durch die HGF_Notifier Klasse beschrieben.

HGF_Notifier

Die HGF_Notifier Klasse verfügt über vier Methoden. Sie überschreibt folgende dynamic dispatch VIs von Ahnenklassen:

- Initialize
- Wait
- Send
- Deinitialize

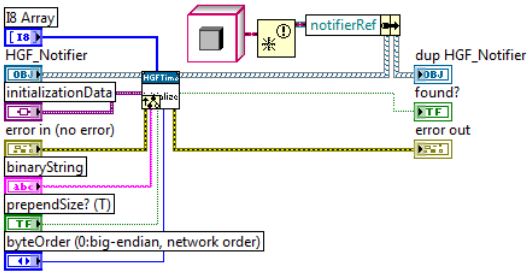

Abbildung 11: Blockdiagramm HGF_Notifier:initialize.vi

Das *initialize* VI, das beim Erzeugen eines HGF_Notifier Objektes durch die Fabrik aufgerufen wird, erzeugt mittels *Obtain Notifier* VI einen neuen Notifier vom Typ LabVIEW Objekt und speichert die zurückgegebene Referenz in seinen Attributen. Weiterhin wird das *initialize* VI der Eltern-Klasse aufgerufen. Innerhalb der *initialize* VIs der Eltern-Klassen kann zusätzlich das schon erwähnte Timeout Objekt (als Attribut der HGF_Timing Klasse) und eine Auszeit (als Attribut der HGF_Event Klasse) gespeichert werden.

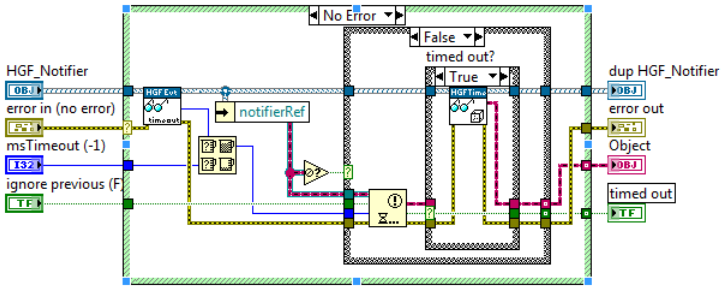

Abbildung 12: Blockdiagramm HGF_Notifier:wait.vi

Das *wait* VI bekommt als Parameter ein Notifier Objekt, den Error Cluster, eine Auszeit und einen booleschen Wert **ignore previous.** Zuerst wird mittels einer öffentlichen Methode die eingestellte Auszeit abgefragt. Dies ist nötig, da das Objekt keinen direkten Zugriff auf das private Attribut hat. Anschließend wird der Maximalwert der objekteigenen Auszeit und der als Parameter übergebenen Auszeit ermittelt. Durch Übergabewert bei Aufruf des *wait* VIs kann explizit eine Auszeit von -1 vermieden werden[25]. Die Notifier Referenz wird ausgelesen und überprüft. Handelt es sich hierbei nicht um eine Referenz, was der Fall ist, wenn das Objekt nicht von einer Fabrik erzeugt wurde, wird das VI beendet. Anschließend wird die *wait on notification* Methode mit den entsprechenden Parametern ausgeführt. Das boolesche **ignore previous** legt hierbei fest, ob eine seit dem letzten

[25] Wird eine Auszeit von -1 gesetzt, so wartet das *wait on notification* VI bis zur nächsten Benachrichtigung. Falls keine Benachrichtigung erfolgt, so blockiert dies alle weiteren Operationen

Aufruf verpasste Notifikation ein Ereignis auslöst, oder ob diese ignoriert wird und bis zur nächsten Benachrichtigung (oder der Auszeit) gewartet wird. Ist hierbei eine Auszeit erfolgt, wird das Timeout Objekt abgefragt, ansonsten wird das empfangene Objekt zurückgegeben.

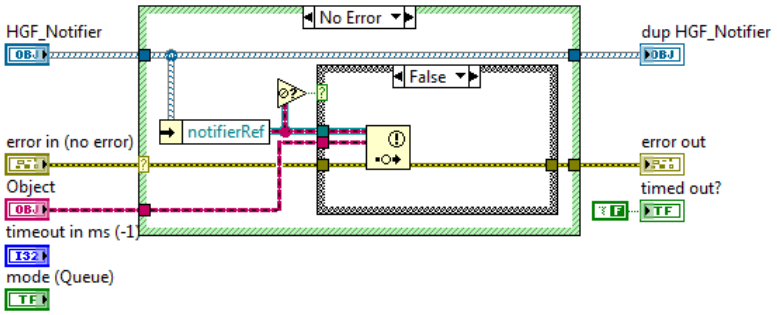

Abbildung 13: Blockdiagramm HGF_Notifier:send.vi

Das *send* VI der Notifier Klasse verwendet die Parameter Error Cluster, Objekt und HGF_Notifier. Es wird die Notifier Referenz ausgelesen, überprüft, und falls es wirklich eine Referenz ist, das Objekt mittels *send notification* Methode als Benachrichtigung gesendet. Wird an einer anderen Stelle auf diese Benachrichtigung mittels *wait* VI und einer Kopie des gleichen Notifier Objektes gewartet, so wird dort ein Ereignis ausgelöst, das eine Kopie des gesendeten Objektes erzeugt.

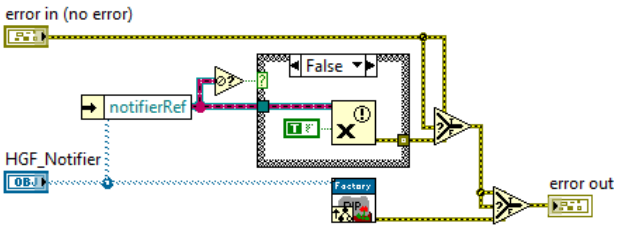

Abbildung 14: Blockdiagramm HGF_Notifier:deinitialize.vi

Wird ein Notifier Objekt nicht mehr benötigt, so empfiehlt es sich, an einer Stelle das destroy Objekt mit einer Kopie des Notifier Objektes aufzurufen. Hierdurch wird, wie schon erwähnt, dessen *deinitialize* VI aufgerufen, innerhalb dessen die erzeugte Benachrichtigung zerstört, der Speicher also wieder freigeben wird. Wird anschließend mit einer Kopie des Notifier Objektes das *wait* VI aufgerufen, so liefert das darin enthaltene *wait on notification* einen Fehler.

Exemplarisch wurde gezeigt, wie ein Ereignismechanismus objektorientiert und datenflusskonform implementiert sein kann. Diese Ereignisse können nun zur Synchronisation zwischen verschiedenen Prozessen, aber, wie gezeigt, auch zur Übertragung von Objekten dienen. Dies ist zum Beispiel notwendig, um bei Verwendung des ThreadPools Besucher übertragen zu können.

HGF_ThreadPool, HGF_ThreadWorker, HGF_ThreadTask

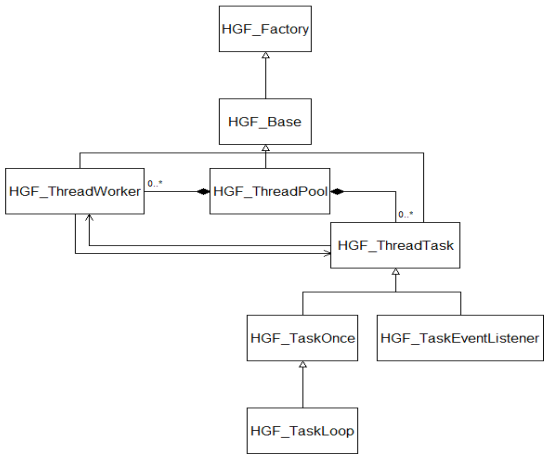

Abbildung 15: UML Klassendiagramm ThreadPool

Die grundlegende Funktionsweise des ThreadPool Entwurfsmusters wurde bereits gezeigt. Im folgenden Unterkapitel wird gezeigt, wie das ThreadPool Muster in LVOOP implementiert wurde und wie es angewendet werden kann. Die Implementierung beinhaltet 3 Basisklassen, welche die Komponenten des Musters darstellen. Dies sind HGF_ThreadPool, HGF_ThreadWorker und HGF_ThreadTask. Um den ThreadPool anwenden zu können, wurden zusätzliche Tasks implementiert, die von HGF_Task geerbt haben. Aufgrund der generischen Implementierung sind diese Tasks für viele Anwendungen bereits ausreichend.

HGF_ThreadPool

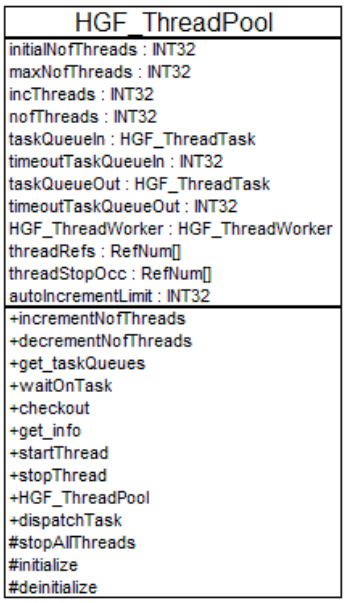

Abbildung 16: Klassendiagramm HGF_ThreadPool

Vor der Initialisierung wird mit Hilfe des *HGF_ThreadPool.AppendData2Variant* VIs festgelegt, wie viele Threads zu Beginn gestartet werden sollen, wie viele Threads maximal gestartet werden dürfen, ab welcher Anzahl an Tasks in der Task Queue (**taskQueueIn**) automatisch Threads gestartet werden, und wie viele Threads auf einmal gestartet werden. Weitere Attribute der ThreadPool Objekte enthalten Referenzen auf die gestarteten Worker und die verwendeten Queues, welche während der Initialisierung erzeugt werden.

Einige der Methoden der ThreadPool Klasse stehen nur Freunden bereit. Da LabVIEW Version 2009, die über Freunde und einen Community-Scope verfügt, zum Zeitpunkt der Erstellung der ThreadPool Klasse noch nicht zur Verfügung stand, wird hierfür ein anderes Konzept verwendet. Mit Hilfe des *CallChain* VIs wird eine Liste der übergeordneten VIs erzeugt. Diese wird mit einer Zeichenkettenliste verglichen. Falls hierbei Übereinstimmungen auftreten, wird ein True zurückgegeben, ansonsten ein False. Dies kann dann verwendet werden, um mittels eines Case Diagramms die Ausführung zu erlauben oder zu verbieten. So kann ein Worker Objekt auf die Queue-Referenzen zugreifen und den Zähler für laufende Threads inkrementieren, beziehungsweise dekrementieren.

Die öffentliche *dispatchTask* Methode erlaubt es dem Programmierer, einen Task zu starten, also ein HGF_Task Objekt der Task Queue hinzuzufügen. Darüber hinaus ruft diese Methode, falls das Auto-Increment Limit überschritten und größer Null ist, die Methode *startThread* auf.

Die Methode *startThread* des ThreadPools dient der Erzeugung neuer Threads. Sie ruft das öffentliche s*tartThread* VI der HGF_ThreadWorker Klasse, welches nach schon erwähntem Freunde-Prinzip den Aufruf nur durch den ThreadPool zulässt.

HGF_ThreadWorker

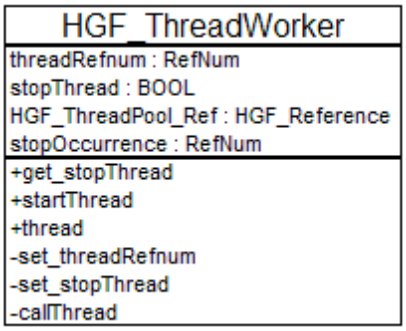

Abbildung 17: Klassendiagramm der HGF_ThreadWorker Klasse

Der ThreadWorker, also ein Objekt der HGF_ThreadWorker Klasse, repräsentiert den Thread des ThreadPool Entwurfsmusters. Der ThreadWorker wird in einem separaten Prozess, also unabhängig vom TopLevel VI, gestartet und ist im Rahmen der Basisklassen das einzige aktive Objekt.

startThread VI

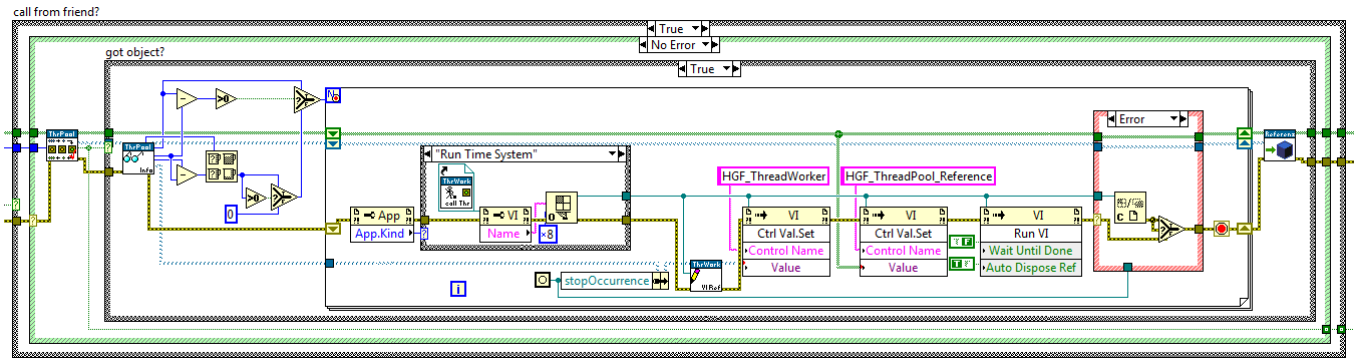

Abbildung 18: Blockdiagramm-Ausschnitt CallThread.vi

Das *startThread* VI der HGF_ThreadWorker Klasse wird von durch das *startThread* VI der HGF_ThreadPool Klasse aufgerufen. Hierbei wird auf das schon beschriebene Freunde-Verfahren zurückgegriffen, um den Aufruf ausschließlich auf die ThreadPool Klasse zu beschränken. Hierbei wird auch die Referenz für das ThreadPool Objekt übergeben. Dies ermöglicht es dem VI, mittels einer öffentlichen Methode einige Attribute des ThreadPools (Threadzähler, AutoIncrement und maximale Threadzahl) auszulesen, aus denen sich die Anzahl der zu startenden Threads errechnen lässt. Anschließend wird auf VI Server Methoden[26] zurückgegriffen, um die benötigte Anzahl an Threads zu starten.

[26] VI Server Methoden erlauben sowohl den dynamischen Zugriff auf Frontpanel Objekte, VIs als auch den Zugriff auf LabVIEW selber. Weiterhin ist es mit diesen Methoden möglich, programmatisch sowohl auf dem lokalen Rechner als auch über Netzwerk VIs zu starten.

Hierfür wird das *callThread* VI explizit in den Arbeitsspeicher geladen. Die nötigen Parameter für die Ausführung werden in die Frontpanel-Objekte dieses VIs geschrieben und anschließend das VI gestartet. Da das VI eine Methode der HGF_ThreadWorker Klasse ist, wird ihr ein HGF_ThreadWorker-Objekt übergeben, das eine Referenz auf das VI und eine Stop-Okkurrenz, die das spätere Beenden des Threads erlaubt, enthält. Zusätzlich wird noch die ThreadPool-Referenz des ausführenden ThreadPools übergeben. Ist der Start erfolgreich, so wird der Threadzähler des ThreadPools inkrementiert. Da das call*Thread* VI reentrant ist, kann auf diese Weise mehr als ein Thread gestartet werden. Das *callThread* VI ruft das *thread* VI auf, welches die eigentliche Funktionalität des Threads beinhaltet.

Thread VI

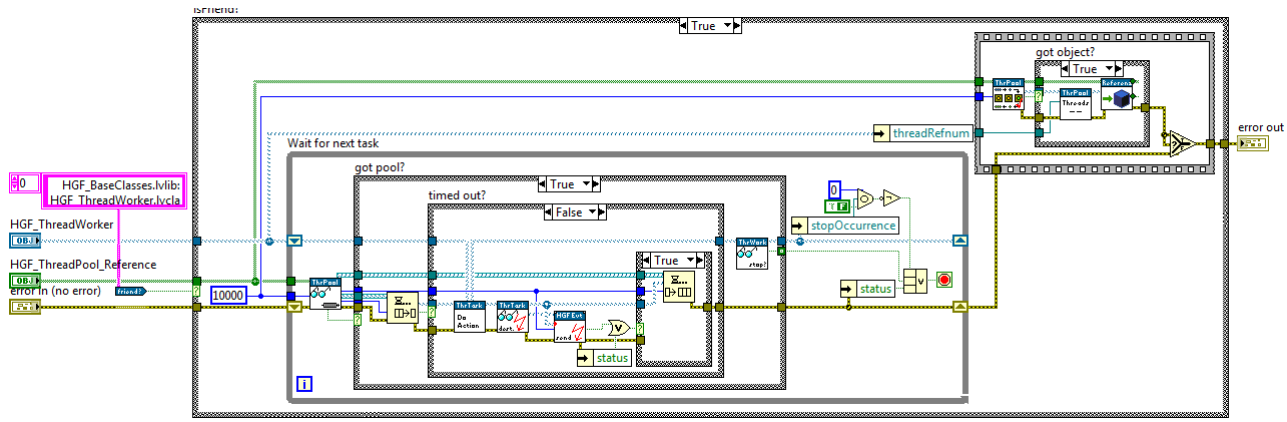

Abbildung 19: Blockdiagramm HGF_ThreadWorker:thread.vi

Das *Thread* VI der HGF_ThreadWorker Klasse läuft streng nach dem Datenfluss-Paradigma. Hier wird die Task-Queue des ThreadPools abgefragt, was dem Warten auf einen Task entspricht. Konnte der Queue ein Task entnommen werden, wird dessen öffentliche *execute* Methode ausgeführt. Nach deren Beendigung wird der Task, falls ein Ziel in Form eines HGF_Event Objekts angegeben wurde, versendet. Ansonsten wird der Task der Completed Task Queue des ThreadPools (taskQueueOut) angehängt.

Solange kein Fehler aufgetreten ist wird mit Hilfe der While-Schleife ununterbrochen auf Tasks gewartet. Da dieses VI unabhängig vom Hauptprogramm läuft, wird eine Stop-Okkurrenz verwendet, um dennoch die Stop-Bedingung der While-Schleife auszulösen und das VI damit zu beenden. Wird ein Thread-VI, also ein Worker, beendet, so wird noch der Zähler des ThreadPools dekrementiert.

HGF_ThreadTask

Der Worker ruft die öffentliche *execute* Methode des Task Objektes auf. Er führt also eine Funktion aus, ohne deren Implementierung zu kennen. Das Task Objekt implementiert den Algorithmus, der ausgeführt werden soll. Dies wird erreicht, indem das *execute* VI das geschützte dynamic dispatch *action* VI aufruft. Dieses kann von Kind-Klassen überschrieben werden und einen beliebigen Algorithmus enthalten.

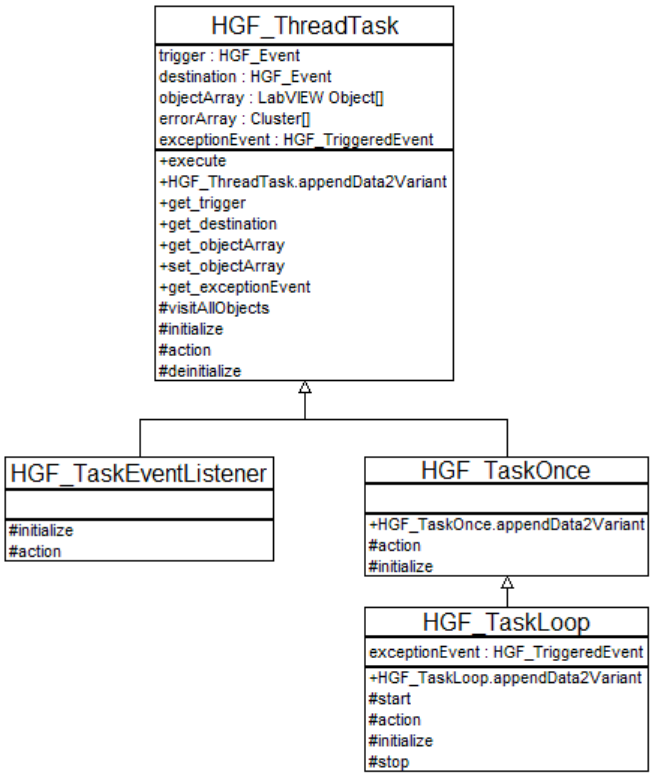

Abbildung 20: UML Diagramm der HGF_ThreadTask-Klasse und der Kind-Klassen

Durch die HGF_ThreadTask Klasse werden Attribute und öffentliche Zugriffsmethoden für diese Attribute bereitgestellt, die ein einheitliches Arbeiten mit spezialisierten Tasks, Klassen die von HGF_ThreadTask geerbt haben, ermöglichen. So verfügt jeder Task über einen Trigger, ein HGF_Event, welches innerhalb eines Tasks genutzt werden kann, um eine Aktion zu starten. Weiterhin existiert ein Ziel, ebenfalls vom Typ HGF_Event, wohin ein beendeter Task gesendet werden kann (dies geschieht innerhalb des *thread* VI des Workers). Ein HGF_TriggeredEvent, das exceptionEvent, ist vorgesehen, um HGF_Task Kind-Klassen das generische Senden von Ereignissen, zum Beispiel im Fehlerfall, implementieren können. Weiterhin steht ein LabVIEW Objekt Array zur Verfügung, in denen beliebige Objekte zwischengespeichert werden können.

HGF_Task Once

HGF_TaskOnce ist die Realisierung eines Tasks, der einmalig ausgeführt werden soll. Um ein möglichst großes Anwendungsfeld abzudecken wird hierin das „Besucher“ Entwurfsmuster angewendet. Hierfür nutzt das HGF_TaskOnce *action* VI die von HGF_ThreadTask bereitgestellten Attribute und Methoden.

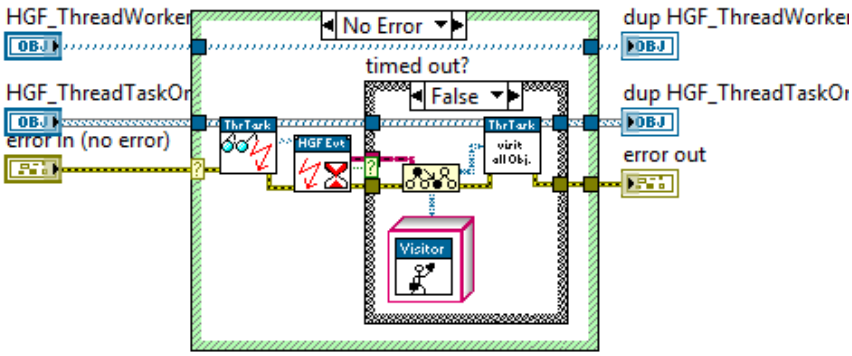

Abbildung 21: Blockdiagramm HGF_TaskOnce:action.vi

HGF_TaskOnce erhält vom Anwender während der Initialisierung eine Reihe von besuchbaren Objekten, die in der LabVIEW Objekt Liste (HGF_ThreadTask) gespeichert werden. Wird HGF_TaskOnce von einem Worker ausgeführt, also durch den Worker das *action* VI aufgerufen, so wartet dieses einmalig auf ein Triggerereignis (HGF_ThreadTask), welches ein LabVIEW Objekt zurückgibt. Es wird ein Besucher erwartet, weshalb eine Typenwandlung zu HGF_Visitor stattfindet. Ist dies erfolgreich, werden anschließend alle Objekte der LabVIEW Objekt Liste besucht. Hierfür stellt HGF_ThreadTask das *visitAll* VI bereit, welches innerhalb einer For-Schleife für jedes LabVIEW Objekte eine Typenwandlung zu HGF_Visitable stattfindet und anschließend das *accept* VI mit dem HGF_Visitable Objekt und dem HGF_Visitor Objekt aufgerufen wird. Die dynamic dispatch Funktionalität gewährt hierbei, dass die überschriebenen VIs der spezialisierten Besucher durch LabVIEW aufgerufen werden.

HGF_TaskOnce nimmt also beliebige besuchbare Objekte bei der Initialisierung auf, wartet nach dem Start durch den Worker auf ein Triggerereignis, dass einen beliebigen Besucher überträgt, und wendet diesen auf alle besuchbaren Objekte an.

HGF_Task Loop

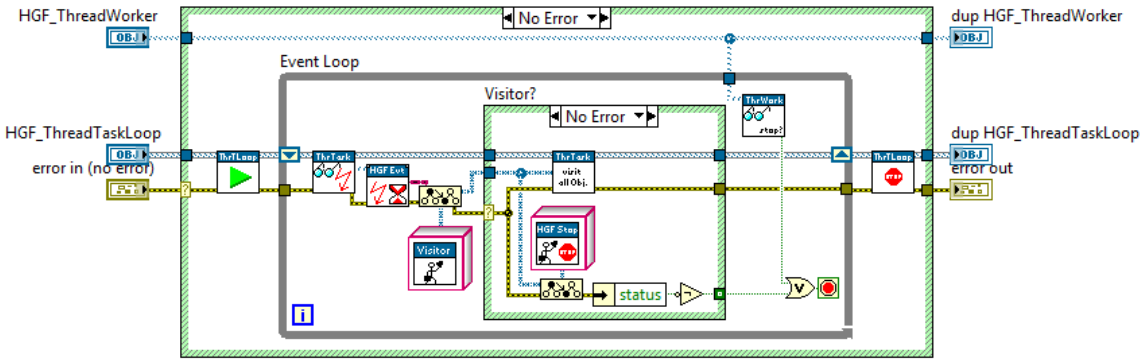

Abbildung 22: Blockdiagramm HGF_TaskLoop:action.vi

HGF_TaskLoop arbeitet wie HGF_TaskOnce auch mit Hilfe des „Besucher“ Entwurfsmusters. Hier wird jedoch innerhalb einer Schleife das Triggerereignis abgefragt. Beendet wird der Task erst, wenn ein Stop Besucher Objekt gesendet oder der ausführende Worker gestoppt wird.

Die Funktion innerhalb der Schleife verläuft analog zu HGF_TaskOnce. Besuchbare Objekte, die für die Ausführung in einem Task-Loop konzipiert sind, können eine Initialisierungs- beziehungsweise Deinitialisierungsroutine durch Überschreiben des *start* und *stop* VIs der HGF_Visitable Klasse implementieren. Diese werden vor, beziehungsweise hinter der Schleife durch das HGF_TaskLoop *action* VI aufgerufen.

Über das Standard-Rückgabeobjekt der HGF_Timing Klasse kann ein Standard-Besucher eingerichtet werden, der bei Zeitüberschreitung der Ereignisabfrage zurückgegeben wird. Dieser wird also jedes Mal auf alle Objekte angewendet, wenn kein anderes Triggerereignis aufgetreten ist.

Shared Variables

LabVIEW verfügt über so genannte Shared Variables[27]. Diese kommunizieren zwischen VIs, Remote Computern und Hardware über die Shared Variable Engine, welche das proprietäre NI Publish-Subscribe Protokoll (NI-PSP) als Datentransferprotokoll nutzt, um Daten zu schreiben und es Anwendern erlaubt, diese live auszulesen.

Das Zusatzmodul „Datalogging and Supervisory Control Module“ (kurz DSC), welches nicht im Basispaket von LabVIEW enthalten ist, erweitert die Möglichkeiten mit Shared Variables zu arbeiten. Dieses Modul ergänzt LabVIEW um Funktionen mit denen Shared Variables programmatisch erzeugt, gelöscht, veröffentlicht oder auch Events für Shared Variables registriert werden können.

HGF_SV und HGF_DSC

Die HGF_SV beziehungsweise HGF_DSC Klassen sind jeweils in einer separaten Bibliothek implementiert und erweitern die HGF Basisklassenbibliothek. Die HGF_SV Bibliothek steht hierbei in der Basisversion von LabVIEW zur Verfügung. Um die HGF_DSC Bibliothek nutzen zu können, muss zusätzlich zur Basisversion noch das DSC Modul installiert sein. Die Klassen der HGF_DSC Bibliothek nutzt die HGF_SV Bibliothek und spezialisiert einige der Klassen.

Die beiden Bibliotheken ermöglichen eine objektorientierte Nutzung der Shared Variables und implementieren hierfür einen Ereignismechanismus. Die Shared Variables können hiermit sowohl nach dem Publisher-Subscriber Prinzip als auch nach dem Kommando Prinzip verwendet werden. Das Publisher-Subscriber Prinzip stellt hierbei eine „One To Many“ Kommunikation dar, das Kommando Prinzip eine „Many To One“ Kommunikation.

Beide Bibliotheken stellen einen Monitor zur Verfügung. Dieser hat von HGF_Visitable geerbt und ist dazu gedacht, in einem TaskLoop verwendet zu werden. Über einen *run* Besucher wird eine Iteration des Monitors gestartet. Um die Änderung der Shared Variables zu überwachen, muss also regelmäßig der *run* Besucher auf das Monitorobjekt angewendet werden. Dies geschieht, indem dieser als Timeout Objekt für den Trigger des TaskLoops eingerichtet wird. Während der Monitor der SV-Klassen im Polling-Betrieb[28] läuft nutzt der Monitor der DSC-Klassen den durch das DSC Modul bereitgestellten Ereignismechanismus um Änderungen zu detektieren.

Zusätzlich zum Monitor verfügt die DSC Bibliothek über eine weitere besuchbare Klasse, den Variable Manager. Dieser wird ebenfalls durch einen TaskLoop, beziehungsweise durch einen Worker aktiviert. Durch Übermittlung eines *register Variable* Besuchers an das Triggerereignis des TaskLoop kann mit Hilfe des Variable Managers programmatisch eine neue Variable erzeugt werden.

Um das Publisher-Subscriber Prinzip zu realisieren, verfügen die Bibliotheken über eine Service Klasse. Diese Klasse hat von der Klasse HGF_PVPublisher geerbt, welche eine Klasse der Basisklassenbibliothek ist und dynamic dispatch Methoden für die Veröffentlichung von Daten bereitstellt. Diese werden von den Service Klassen überschrieben um die Veröffentlichung der Daten

[27] Umgebungsvariablen

[28] Bei jeder Iteration werden alle für eine Überwachung registrierten Variablen abgerufen und mit dem letzten Wert verglichen, wodurch eine Änderung registriert werden kann

an eine Shared Variable zu implementieren. Ein Serviceobjekt wird bei der Initialisierung an eine Shared Variabel gebunden und stellt den Publisher dar.

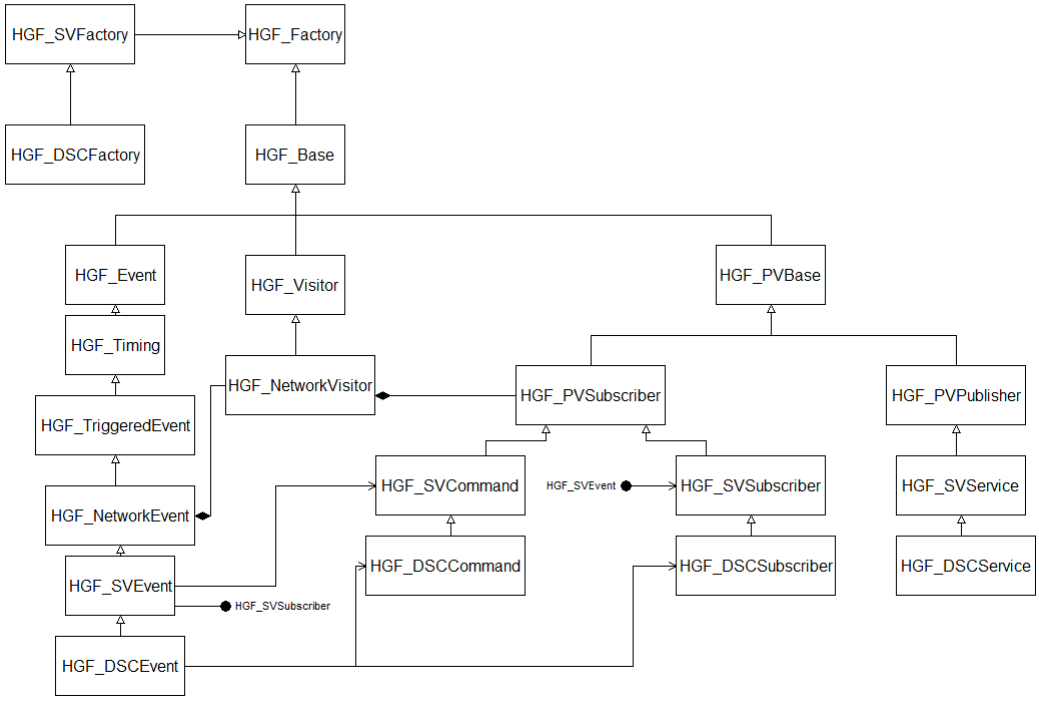

Abbildung 23: UML Klassendiagramm der HGF_SV und HGF_DSC Bibliotheken

Der Suscriber des Publisher-Subscriber-Prinzips und das Kommando haben ähnliche Aufgaben. Beide müssen sich für einen Service anmelden und über Änderungen informiert werden. Der implementierte Ereignismechanismus ist in der Lage, mehrere Kommando- beziehungsweise Subscriberobjekte zu verarbeiten.

Sowohl die SV-Bibliothek als auch die DSC-Bibliothek verfügen hierfür über eine Eventklasse, die von HGF_NetworkEvent geerbt hat und ein Bestandteil der bestehenden HGF_Eventklassen ist. Das Netzwerkereignis besitzt eine Reihe von Netzwerkbesucher, also Objekte der HGF_NetworkVisitor Klasse. Jedes dieser Objekte hat als Attribut ein HGF_PVSubscriberobjekt. Da sowohl Kommando als auch Subscriber von HGF_PVSubscriber geerbt haben, kann ein Netzwerkbesucher also jeweils ein Kommando- oder Subscriberobjekt in seinen Attributen haben. Bei der Initialisierung eines SV- oder DSC-Ereignisses werden alle Netzwerkbesucher ausgelesen und, falls sie ein Entsprechendes Kommando- oder Subscriberobjekt in ihren Attributen haben, bei einem Monitor für die Überwachung registert.

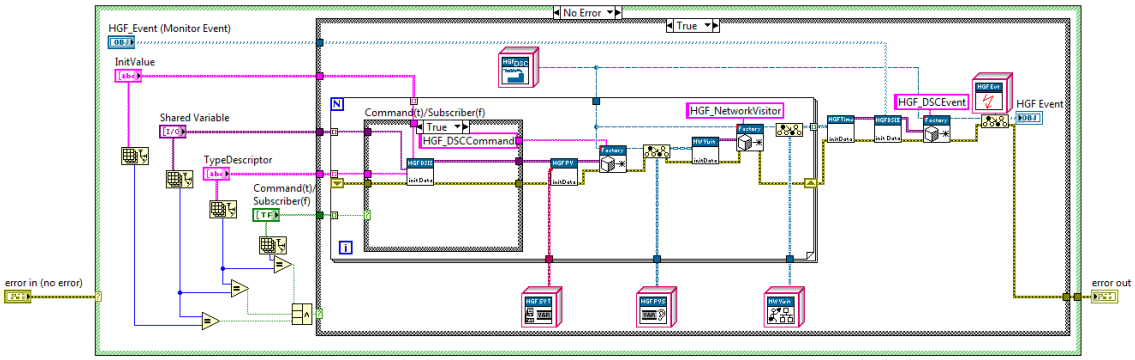

Abbildung 24: Initialisierungsparameter für ein DSC Ereignis

In Abbildung 24 ist dargestellt, wie ein DSC Ereignis die nötigen Initialisierungsparameter erhalten kann, um für eine Änderung einer oder mehrerer Shared Variables registriert zu werden. Die Listenelemente zu Beginn der Operation enthalten hierbei die nötigen Initialisierungsparameter für die Kommando- beziehungsweise Subscriberobjekte. Die Anzahl der zu erstellenden Objekte ergibt sich aus der Zahl der Elemente in den Listen. Über einen booleschen Parameter wird ausgewählt, ob es sich um ein Kommandoobjekt oder um ein Subscriberobjekt handeln soll. Die Zugewiesene Shared Variable so wie eine für die Datenbehandlung verwendete Typeninformation werden als Initialisierungsparameter für das Objekt angegeben woraufhin das Objekt mit Hilfe einer Fabrik erstellt wird. Per Typenwandlung wird aus dem so erstellten Objekt ein HGF_PVSubscriberobjekt, welches als Initialisierungsparameter für einen Netzwerkbesucher verwendet wird. Nach der Erzeugung des Netzwerkbesuchers findet wieder eine Typenwandlung statt. Dies ist nötig, da das *createObject* VI immer ein Objekt vom Typ Fabrik ausgibt. Auf diese Weise wird mittels For-Schleife die gewünschte Anzahl an Netzwerkbesuchern erzeugt. Diese dienen anschließend, neben dem TaskTrigger des DSC Monitor-Tasks bei dem die Registrierung erfolgen soll, als Initialisierungsparameter für das DSC Ereignis.

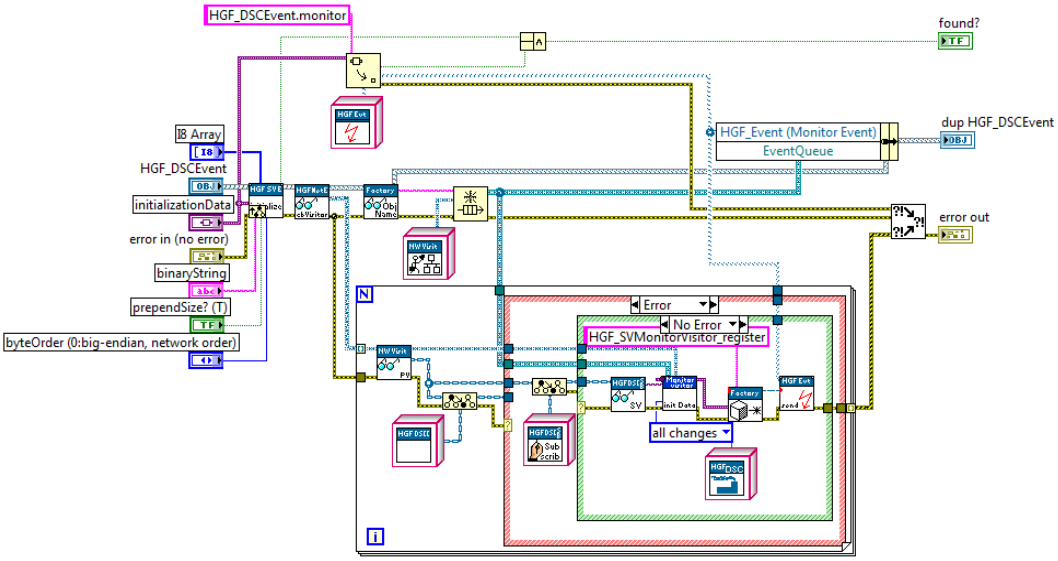

Abbildung 25: Initialisierungsroutine HGF_DSCEvent

Während der Initialisierung wird eine Queue des Typs Netzwerkbesucher erzeugt. Diese wird vom *wait* VI des DSC-Ereignisses überprüft, um neue Ereignisse festzustellen. Zusätzlich zur Erstellung der Queue werden die Netzwerkbesucher ausgelesen. Anschließend wird das HGF_PVSubscriber Objekt

jedes Netzwerkbesuchers ausgelesen. Nun erfolgt eine Typenwandlung zu einem Kommandoobjekt. Ist diese Typenwandlung erfolgreich, so kann die Shared Variable des Objektes ausgelesen werden. Diese wird zusammen mit der Queue-Referenz der erstellten Queue und dem Netzwerkbesucher, zu dem das Objekt gehört, als Initialisierungsparameter für einen Registrierungsbesucher verwendet, der anschließend an das Triggerereignis des Monitor-Tasks gesendet wird. Falls die Typenwandlung nicht erfolgreich war, so wird die Typenwandlung zu einem Subscriberobjekt mit dem gleichen Anschlussverfahren. Ist auch diese Typenwandlung nicht erfolgreich, so kann das DSC Ereignis nicht mit dem HGF_PVSubscriber Objekt umgehen.

Durch dieses Verfahren hat nun der Monitor-Task einen oder mehrere Besucher gesendet bekommen. Diese registrieren die Shared Variable in diesem Falle für ein Ereignis. Zusätzlich wird der übertragene Netzwerkbesucher zusammen mit der Queue Referenz in einem Cluster gespeichert.

Detektiert der DSC Monitor eine Änderung der Shared Variable, in diesem Falle mittels Ereignismechanismus, so bekommen alle für diese Shared Variable registrierten Netzwerkbesucher den neuen Wert der Shared Variable in ihren Attributen gespeichert. Anschließend werden sie der verknüpften Queue angehängt.

Die *wait* Funktion des DSC Events wartet also auf eingehende Netzwerkbesucher, die die neuen Werte der Shared Variable als Attribut verfügbar haben. Zusätzlich hat der Netzwerkbesucher in den Attributen noch das HGF_PVSubscriber Objekt, wodurch sich der Besucher eindeutig zuordnen lässt.

Mit Hilfe dieses Ereignismechanismus lässt sich somit sowohl das Publisher-Subscriber Prinzip als auch das Kommando Prinzip objektorientiert und datenflusskonform implementieren.

Auf gleiche Weise ist auch eine Bibliothek für eine objektorientierte Verwendung von DIM[29] implementiert. Aufgrund der generischen Implementierung ist es möglich, beide Systeme auch kombiniert zu betreiben.

[29] Distributed Information Management System (Cern, 1997), (Beck, Brand, & Kurz, 2005)

Ein mobiles Agentensystem, das auf den HGF Basisklassen aufsetzt

Die Grundlage für die Realisierung des mobilen Agentensystems bildet die HGF Basisklassenbibliothek. Die Anforderungen für ein mobiles Agentensystem in LVOOP, so zum Beispiel die passiven Objekte, die einem aktiven Agenten gegenüberstehen, lassen sich durch Anwendung der Entwurfsmuster oder Kombination mehrerer Entwurfsmuster wie schon gezeigt lösen. Dies geschieht, indem entweder die Basisklassen direkt eingesetzt oder, mittels Vererbung, diese spezialisiert werden.

Die wesentlichen Bestandteile und Eigenschaften des als Prototyp realisierten mobilen Agentensystems, die nachfolgend näher erläutert werden, sind hierbei:

- Host
- Agentenklassen und Agentenobjekte
- Arbeitsumgebung
- Kommunikationssystem
- Erweiterbarkeit
- Sicherheitskonzept

Host

Die zentrale Stelle in einem Agentensystem nimmt der Host ein. Der Host dient als Schnittstelle zwischen Rechner und Agent. Durch Bereitstellen von Arbeitsumgebungen ermöglicht er es einem Agenten, auf einem Rechner als aktive Software zu laufen. Durch den generischen Start von Agentenobjekten ist der Host dabei nicht an einen bestimmten Typ von Agentenobjekt, also einen spezialisierten Agenten, gebunden, sondern ist in der Lage, jedem, auch bisher unbekannten, Agentenobjekt eine Arbeitsumgebung bereit zu stellen. Hierbei müssen aus Sicherheitsgründen Restriktionen für den Start neuer Agenten vorgesehen sein, worauf im Abschnitt Sicherheitskonzept näher eingegangen wird.

Der Host des mobilen Agentensystems besteht aus zwei Teilen, einer Hostklasse und einer Host Applikation, dem *runHost* VI. Die Hostklasse dient dazu, alle nötigen Daten der Applikation in einem zentralen Objekt zur Verfügung zu stellen und ist nicht in das Basisklassensystem eingegliedert. Das *runHost* VI repräsentiert den eigentlichen Host, also die Applikation, welche Arbeitsumgebungen und eine lokale Kommunikationsumgebung bereit stellt. Weiterhin empfängt und startet sie mobile Agentenobjekte und ruft die Methoden der Hostklasse auf wodurch Verwaltungsaufgaben realisiert werden. Zusätzlich wird über das Frontpanel der Applikation eine lokale Benutzerschnittstelle zur Verfügung stellt.

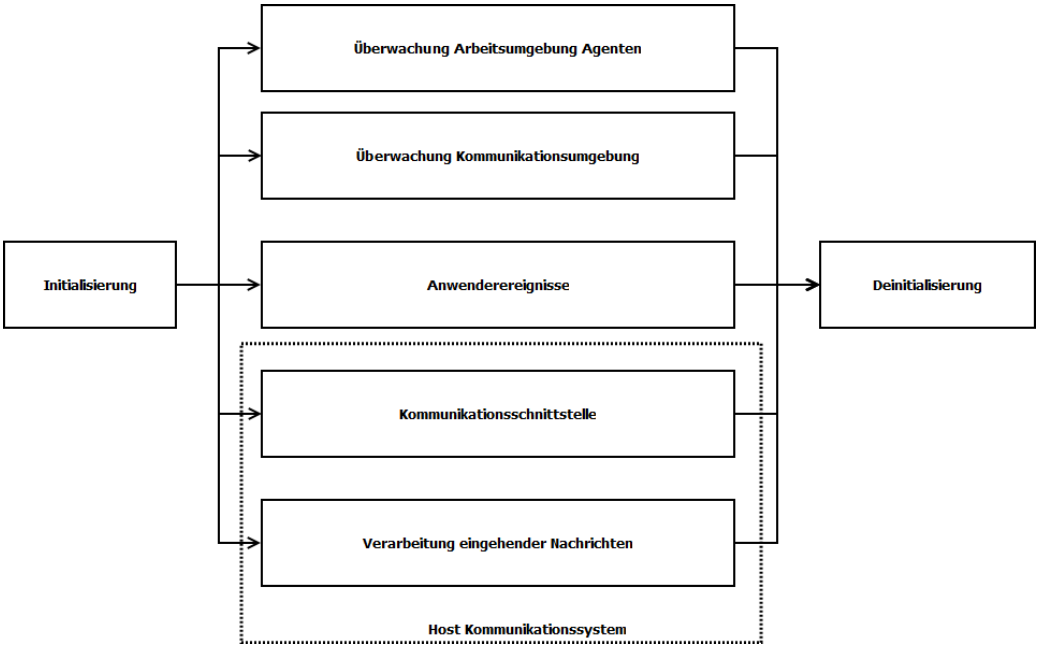

Abbildung 26: Flussdiagramm des *runHost* VIs

Abbildung 26 zeigt schematisch den Ablauf des *runHost* VIs. Nach einer Initialisierungsphase wird eine Reihe parallel arbeitender Prozesse gestartet. Nach der Ausführung der Prozesse, die jeweils mittels einer While-Schleife implementiert sind, erfolgt noch eine Deinitialisierung.

Während der Initialisierung wird ein Host Objekt entweder neu erzeugt oder von Festplatte rekonstruiert. Des Weiteren wird eine Kommunikationsumgebung sowie die Kommunikationsschnittstelle für den Host erstellt.

Die Deinitialisierung beinhaltet das Beenden der lokalen Kommunikationsschnittstelle, das Beenden der Arbeitsumgebungen und die Speicherung des Host Objektes auf der Festplatte, um die Arbeitsdaten des Hosts zu einem späteren Zeitpunkt wieder zugänglich zu machen.

Auf die verschiedenen Threads des *runHost* VIs wird zu einem späteren Zeitpunkt eingegangen.

Die Hostklasse beinhaltet Attribute, welche für die lokale Verwaltung notwendig sind und stellt alle zur Verarbeitung notwendigen VIs bereit. Da das Host Objekt von allen Prozessen aus zugänglich sein muss wird auf die HGF_Reference Klasse zurückgegriffen und so das Host Objekt über die verschiedenen Prozesse hinweg als Entität behandelt. Um das Arbeiten mit den referenzierten Host Objekten zu erleichtern, verfügen diejenigen Hostklassenmethoden, welche vom *runHost* VI aus zugänglich sein sollen, über eine öffentliche Wrapper-Methode[30]. Diese Wrapper-Methode übernimmt jeweils die Aufgabe, das Host Objekt aus dem Referenzobjekt auszuchecken[31], das VI mit dem entsprechenden Algorithmus aufzurufen und das Objekt anschließend wieder einzuchecken[32], um es für den nächsten Methodenaufruf zugänglich zu machen.

[30] Eine Wrapper-Methode „umschließt“ die eigentliche Methode. Durch eine öffentliche Wrapper-Methode kann eine private oder geschützte Methode öffentlich zugänglich gemacht werden.
[31] Das Objekt wird mittels *dequeue* Element aus der referenzierten Queue entfernt
[32] Das Objekt wird in die referenzierte Queue eingereiht

Agentenklasse und Agentenobjekte

Für das Agentensystem wird eine Agenten-Basisklasse vorgesehen. Die Agenten-Basisklasse implementiert grundlegende Attribute und Methoden aller Agenten.

Ein Agentenobjekt ist, wie jedes andere Objekt in LabVIEW, passiv. Ein aktiver Agent wird erzeugt, indem ein Agentenobjekt durch eine Arbeitsumgebung aktiviert wird. Um mit dem Agenten interagieren zu können, also Methoden der Agentenklasse auf das Agentenobjekt anwenden zu können, ohne dabei das Objekt selber aus seiner Arbeitsumgebung zu entfernen, wird auf das Besucher Entwurfsmuster zurückgegriffen. Die Agenten-Basisklasse erbt hierfür von HGF_Visitable. Durch die Verwendung des Besucher Entwurfsmusters ist es möglich, ein Agentenobjekt innerhalb eines Prozesses, der Arbeitsumgebung, zu kapseln und so vor unerwünschtem Zugriff zu schützen.

Zur Realisierung von Basisaufgaben werden static dispatch VIs[33] eingesetzt. Diese enthalten die Funktionen für die Identifikation eines Agentenobjektes, das Auslösen einer Migration und das gezielte Beenden eines Agenten. Zusätzlich existiert ein dynamic dispatch *message handler* VI, auf das im Abschnitt Kommunikationssystem noch einmal eingegangen wird.

Die Attribute der Agenten Basisklasse enthalten Informationen, die für die Identifizierung und die Migration nötig sind. Weiterhin stehen ein Standard-Besucher bereit, der ein Agieren des Agenten ermöglicht, sowie ein Objekt zur Sicherung der Kommunikation. Eine Erläuterung hierzu findet in den entsprechenden Abschnitten „Arbeitsumgebung“, beziehungsweise „Kommunikationssystem“ statt.

[33] Nicht überschreibbare Klassenmethoden

Arbeitsumgebung

Eine Möglichkeit die Agentenobjekte wie agierende Software erscheinen zu lassen, besteht darin, sie ähnlich der Worker des ThreadPools über VI Servermethoden zu starten. Dieser „Verstoß“ gegen das Datenfluss-Paradigma ist zwar für die Funktion des ThreadPools unabdingbar, jedoch für die Ausführung mobiler Agenten unerwünscht. Stattdessen werden Agenten mittels des schon implementierten ThreadPool Patterns „zum Leben erweckt“.

Viele Funktionen eines Agenten sind abhängig von seinem Zustand. National Instruments bietet für LabVIEW das sogenannte „Statechart Module“ an, welches Möglichkeiten für die Implementierung einer Zustandsmaschine bereitstellt.

LabVIEW Statechart Modul

Die LabVIEW Statecharts[34] verfügen über Trigger, Eingänge, Ausgänge und Zustandsdaten, über die sie gesteuert werden. Zusätzlich verfügen Sie noch über ein Diagramm, in dem sich die Zustände, Zustandsübergänge und die damit verbundenen Aktionen programmieren lassen. Diese Zustandsmaschinen können sowohl mit internen als auch mit externen Triggern arbeiten, die Zustandsübergänge bewirken können. Ein Agentenobjekt kann über einen Eingang an die Zustandsmaschine übergeben und dort in den internen Zustandsdaten gespeichert werden. Diese Daten sind nach außen hin gekapselt, also nicht mehr zugänglich. Dies schützt innerhalb des Agentensystems zusätzlich vor ungewollten Zugriffen auf die Agentenobjekte.

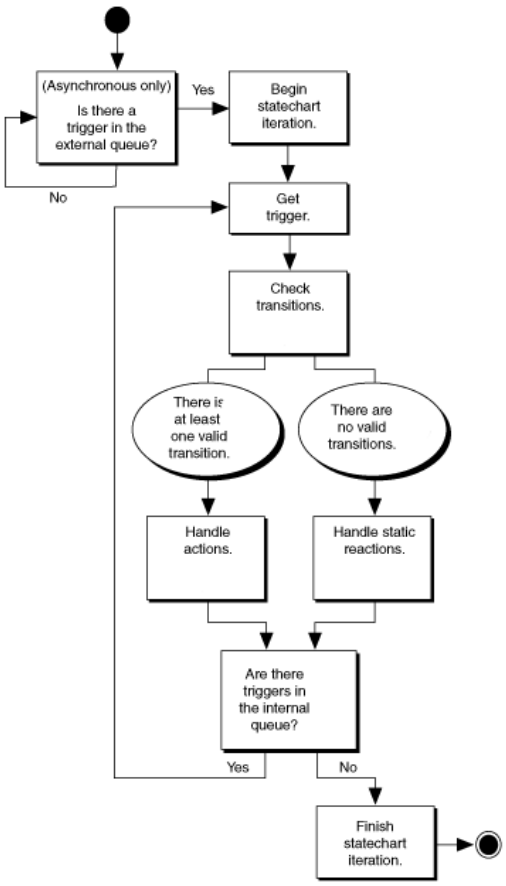

Abbildung 27: Einzelner Durchlauf einer Zustandsmaschine (NI LabVIEW Hilfe zu „Executing Statechart Iterations (Statechart Module)“)

[34] Zustandsmaschinen

Eine Zustandsmaschine wird mittels *Run Statechart* VI ausgeführt und erledigt bei jedem Aufruf eine Iteration, die wie in Abbildung 27 dargestellt abläuft.

Bei Zustandsmaschinen existieren zwei Ausführungsmodi, synchron und asynchron. Diese unterscheiden sich darin, dass synchrone Zustandsmaschinen ausgeführt werden, sobald mittels runStatechart VI alle Eingänge anliegen, während bei asynchronen Zustandsmaschinen zusätzlich noch ein externer Trigger vorhanden sein muss. Externe Trigger werden bei synchronen Zustandsmaschinen als Parameter des *runStatechart* VIs angegeben. Bei asynchronen Zustandsmaschinen hingegen steht für das Senden eines externen Triggers ein separates VI zur Verfügung, welches von anderen VIs aus aufgerufen werden kann. Die Zuordnung der Trigger zu einer speziellen Zustandsmaschine erfolgt hierbei über einen Instanznamen, welcher im asynchronen Fall an das *RunStatechart* VI übergeben wird. Zusätzlich werden in beiden Fällen, also bei synchroner und asynchroner Implementierung, die Eingangsparameter der Zustandsmaschine über das *runStatechart* VI angegeben. Im Falle des Agentensystems sind Triggerereignis und Eingangsdaten verknüpft. Es wird, wie schon erwähnt, mit dem Besuchermuster gearbeitet. Ein Triggerereignis für die Zustandsmaschine erfolgt in diesem Falle parallel mit dem Erhalt des Besuchers. Aus diesem Grund wird die Zustandsmaschine für das Agentensystem synchron verwendet.

Wie in Abbildung 27 ersichtlich wird zuerst überprüft, ob der Trigger einen Zustandswechsel veranlasst. Jeder Zustand kann mit mehreren statischen, einer Eingangs- und einer Ausgangsaktion versehen werden. Löst das Triggerereignis keinen Zustandswechsel aus, werden statische Aktionen ausgeführt. Auch diese können abhängig von einem bestimmten Trigger sein. Zustandswechsel können mit einer Wächteraktion[35] und zusätzlich mit einer Aktion versehen werden. Falls ein Triggerereignis einen Zustandswechsel auslösen kann, so wird dieser zuerst mittels Wächteraktion verifiziert. Anschließend wird die Ausgangsaktion des alten Zustands, die Aktion des Zustandsüberganges und die Eingangsaktion des neuen Zustandes ausgeführt. Falls kein interner Trigger vorliegt, so wird die Iteration beendet. Für interne Trigger verfügt die Zustandsmaschine über eine interne Queue. Mittels *send internal Trigger* VI kann, von innerhalb der Zustandsmaschine, ein Trigger zu dieser Queue hinzugefügt werden. Dieser Trigger wird dann wie ein externer Trigger behandelt. Eine Iteration der Zustandsmaschine wird weitergeführt, so lange interne Trigger in dieser Queue enthalten sind.

Um als Umgebung für einen Agenten zu dienen, muss eine Zustandsmaschine wiederholt aufgerufen werden. Ein erweiterter HGF_TaskLoop, der MoAgSy_Engine Task, übernimmt diese Aufgabe. Um auszuschließen, dass sich Agenten gegenseitig blockieren, wird für jeden Agenten ein separater Task durch die vom ThreadPool erstellten Worker aufgerufen.

[35] Wächteraktionen liefern immer ein boolesches Ergebnis. Dieses erlaubt oder verbietet den Zustandswechsel.

Für die Bereitstellung des ThreadPools ist der Host, beziehungsweise das *runHost* VI, zuständig. Ein Worker, also ein Thread, kann bei aktueller Implementierung nur einen Task auf einmal starten. Erst wenn der Task beendet ist, ist es möglich, den nächsten Task auszuführen. Da jeder Agent einen eigenen Task zur Verfügung hat, ist die Anzahl der möglichen aktiven Agenten bei einem Host durch die maximale Zahl der Worker des verarbeitenden ThreadPools limitiert.

Die Arbeitsumgebung für ein Agentenobjekt besteht also aus einem ThreadPool, der eine Reihe von Threads startet. Diese aktivieren eine Reihe von Tasks, welche wiederholt eine Zustandsmaschine ausführen, in der sich das Agentenobjekt befindet.

Der schematische Ablauf demonstriert, wie aus einem ankommenden passiven Agentenobjekt ein aktiver Agent werden kann.

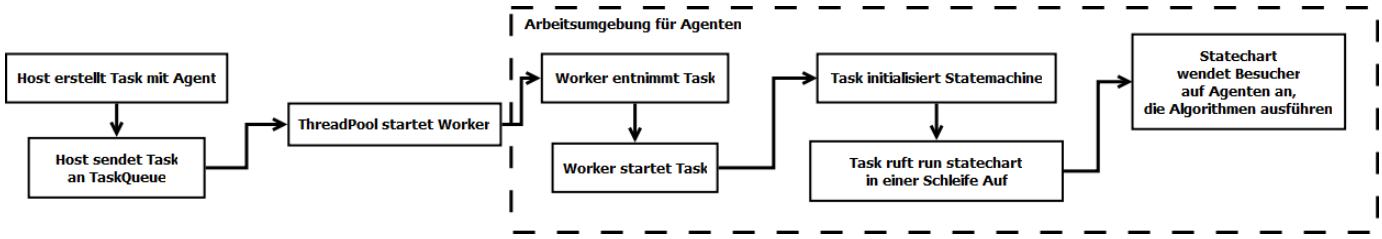

Abbildung 28: Schematischer Ablauf für den Start eines Agenten durch den Host

Die Funktionsweise, wie externe Besucher zum Statechart gelangen, wird ab Seite 48 näher erläutert.

Ein Agentenobjekt ist ein besuchbares passives Objekt, welches durch die Arbeitsumgebung aktiviert wird. Als dieses kann es auf ankommende Besucher reagieren. Um jedoch auch agieren zu können, muss der Agent dazu in der Lage sein, selbständig Methoden aufzurufen. Innerhalb der Agenten Basisklasse ist hierfür ein Standardbesucher vorgesehen, der mit einer Auszeit kombiniert ist. Ist die Auszeit abgelaufen, so wird der Standardbesucher auf den Agenten angewendet. Spezialisierte Agenten können über diesen Mechanismus selbständig einen beliebigen Algorithmus ausführen, sind somit also auch in der Lage zu agieren.

Um einen aktiven Agenten zu stoppen existieren mehrere Möglichkeiten. Eine Möglichkeit ist die Übertragung eines Stop-Besuchers. Dieser setzt das Attribut **nextState** des Agentenobjektes, welches ein Trigger der Statemachine ist, auf „stop“. Innerhalb der Statemachine wird dieser Trigger nach jeder Anwendung eines Besuchers mittels öffentlicher Methode ausgelesen und der internen Triggerqueue angehängt.

Der Host erstellt bei der Erzeugung eines Engine Tasks eine Stop-Okkurrenz. Diese wird vom Host in einer Liste bereit gehalten und kann gesetzt werden, um einen Agenten gezielt zu beenden. Wird der Host beendet, so werden alle Okkurrenzen dieser Liste gesetzt. Dies ist wichtig, da die Agenten in einem vom Host unabhängigen Prozess verarbeitet werden, der bei Beendigung des Hosts nicht automatisch mit beendet wird. Wurde eine Stop-Okkurrenz gesetzt, so wird innerhalb des Engine

Tasks dem *runStatechart* VI ein Stop-Trigger übergeben. Dies stellt die zweite Möglichkeit dar, einen aktiven Agenten zu beenden. Die letzte Möglichkeit besteht darin, eine Migration auszulösen. Hierfür wird ein Reisebesucher an den Agenten übertragen, der unter Anderem das **nextState** auf „travel“ setzt. Ist die Übertragung des Agenten, die im Status „travel“ der Zustandsmaschine erfolgt, erfolgreich, so wird anschließend der Agent ebenfalls beendet. In allen drei Fällen wechselt die Zustandsmaschine in den Zustand „shutdown“. Hierin findet, neben der Deinitialisierung der lokalen Kommunikationsschnittstelle, der Aufruf des *deinitialize* VIs des Agentenobjektes statt. Das *deinitialize* VI der Agentenklasse überschreibt das entsprechende dynamic dispatch VI der HGF_Factory Klasse und ist, da es dynamic dispatch ist, geschützt. Um der Statemachine den Aufruf dennoch zu ermöglichen, verfügt die Agenten-Basisklasse über eine öffentliche Wrapper Methode. Ein Objekt der Agenten-Basisklasse wird beim Aufruf nur beendet. In spezialisierten Klassen ist es jedoch Möglich, andere Maßnahmen für eine Beendigung des Agenten zu treffen. Über das **nextState** Attribut der Basisklasse lässt sich hierbei unterscheiden, aus welchem Grund das Herunterfahren erfolgt. So lassen sich in spezialisierten Agenten unterschiedliche Reaktionen für das Herunterfahren durch den Host oder die Beendigung durch einen Besucher implementieren.

Kommunikationssystem

Als Kommunikationsschicht wird für das Agentensystem die Verwendung von Shared Variables vorgesehen. Um die Shared Variables in LVOOP nutzen zu können, wurden die Basisklassen um die schon auf Seite 36 erläuterten Bibliotheken, HGF_SV und HGF_DSC, ergänzt.

Kommunikation im Rahmen des mobilen Agentensystems

Kommunikation im Rahmen des mobilen Agentensystems beinhaltet die Übertragung von Nachrichten, die Übertragung von Besuchern für entfernte Methodenaufrufe und die Übertragung von Agenten für eine Migration.

Für die Kommunikation werden die HGF_DSC Klassen verwendet. Das Host-VI stellt einen ThreadPool für die Kommunikation bereit, in dem ein VariableManager-Task und ein Monitor-Tasks gestartet werden.

Es wird für jeden Agenten (innerhalb der Zustandsmaschine), sowie für den Host, programmatisch je eine Shared Variable für die Kommunikation erzeugt. Dies erfolgt, indem ein Besucher der Klasse *HGF_SVVarManVisitor_create* über das entsprechende Triggerereignis an den VariableManager-Task gesendet wird, dort die entsprechende Methode zur programmatischen Erzeugung aufruft und anschließend mit einer Kopie der Shared Variable über ein zuvor erzeugtes Zielevent zum Erzeuger zurückkehrt. Nun steht die Shared Variable für die weitere Verarbeitung zur Verfügung. Anschließend wird ein DSC-Event für diese Shared Variable, mit Hilfe eines weiteren Besuchers, bei dem Monitor-Task registriert. Durch warten auf das DSC-Event ist eine Kommunikationsschnittstelle für eingehende Nachrichten vorhanden.

Die Übertragung von Nachrichten findet in Form von Strings[36] statt. LabVIEW verfügt von Haus aus über ein „Flatten to String“, beziehungsweise ein „Unflatten From String“ VI. Mit Hilfe dieser VIs ist es möglich, Daten jeglichen Datentyps in einen String und anschließend wieder zu ihren ursprünglichen Datentypen zu wandeln. Dies wird verwendet, um eine einheitliche Basis für die Übertragung zur Verfügung zu stellen.

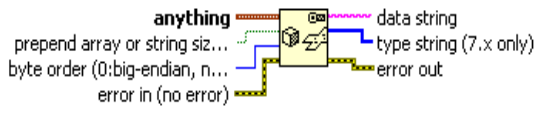

Abbildung 29: Flatten To String VI

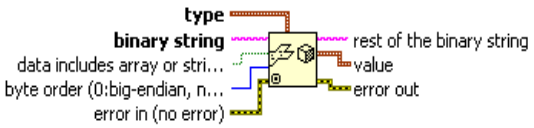

Abbildung 30: Unflatten From String VI

[36] Zeichenketten

Abbildung 30 zeigt das polymorphe VI „Unflatten From String“. Der Eingang **type** repräsentiert den Datentyp, in den der **binary string** umgewandelt werden soll. Hier muss eine entsprechende Konstante oder ein Control vom Zieldatentyp angelegt werden. **Value** enthält anschließend die Daten.

Dem ist zu entnehmen, dass empfängerseitig der passende Datentyp für eine Rekonstruktion bekannt sein muss. Um stattfindende Kommunikation im Agentensystem zu vereinheitlichen wird eine Message-Klasse bereitgestellt.

Objekte der Message-Klasse können neben einigen für die Kommunikation relevanten Zusatzdaten in ihren Attributen sowohl Strings als auch direkt LabVIEW Objekte transportieren. Da die LabVIEW Objekt-Klasse die ultimative Ahnenklasse ist, sich also Jeder Klassentyp über einen Typecast in ein LabVIEW-Objekt wandeln lässt, kann mit Hilfe der Message-Objekte jeder Klassentyp übertragen werden. Dies wird genutzt, um im Rahmen des Agentensystem sowohl Besucher an Agenten zu übertragen, als auch die Agenten selber zu verschicken.

Nachrichtenempfang durch den Host

Das *runHost* VI verfügt über zwei parallel laufende Prozesse für die Kommunikation. Der erste Prozess wartet auf eingehende Nachrichten, indem er das für die Host-Shared Variable registrierte DSC-Event abfragt. Ein per Ereignis empfangenes HGF_NetworkEvent Objekt enthält, wie schon im Kapitel „Shared Variables“ erwähnt, den neuen Wert der Variablen in Zeichenkettenform. Aus diesem Zeichenkettenwert wird versucht, das Message-Objekt zu rekonstruieren. Ist eine Rekonstruktion nicht möglich, so kann die eingehende Nachricht nicht verarbeitet werden und wird ignoriert. Ist die Rekonstruktion erfolgreich, so wird der Empfänger, ein Attribut des Message Objektes, ausgelesen. Nur falls der Empfänger der Host ist, wird das Message Objekt weiter verarbeitet. Hierzu wird es mit Hilfe einer Queue an den zweiten Prozess übergeben.

Im zweiten Prozess wird die schon erwähnte Queue ausgelesen, also auf eingegangene Message Objekte gewartet. Anschließend wird der Typ der Nachricht, ein weiteres Attribut des Message-Objektes, welches Auskunft darüber gibt, ob es sich um eine Textnachricht, einen Besucher oder ein Agentenobjekt handelt und bei der Erstellung des Message Objektes als Parameter angegeben wird, ausgelesen. Dieser Nachrichtentyp regelt mit Hilfe eines Case Diagrammes die weitere Verarbeitung.

Nachrichtenempfang durch einen Agenten

Wie beim Nachrichtenempfang durch den Host auch wird beim Nachrichtenempfang durch einen Agenten ein Ereignis, in diesem Falle wiederum das beim Monitor registrierte DSC-Ereignis, abgefragt. Dies geschieht innerhalb des Engine Tasks. Bei einem eingehenden Ereignis wird das empfangene Objekt und ein Ereignistrigger an das Statechart übergeben. Die weitere Verarbeitung findet innerhalb des Statecharts statt. Der Ereignistrigger löst einen Zustandswechsel (neuer Zustand soll hierbei „handle Event“ sein) aus. In der *Guard* Funktion des Zustandswechsels wird überprüft, ob das eingegangene Objekt wirklich ein Netzwerkereignis ist, indem ein Typecast gemacht wird. Nur falls dies gelingt findet der Zustandswechsel statt.

In der *action* Methode des Zustandswechsels erfolgt die Abfrage des Strings und der Versuch der Rekonstruktion eines Message Objektes. Falls hierbei ein Fehler erfolgt ist, wechselt die Zustandsmaschine wieder in den Zustand „waiting“ zurück. War die Rekonstruktion eines Message Objektes erfolgreich, so wird, wie auch beim Host, der Typ der Nachricht ausgelesen und die entsprechende weitere Verarbeitung eingeleitet.

Textnachrichten

Mit Hilfe der Message Objekte können Textnachrichten, also Zeichenketten, verschickt werden. Hierfür wird ein neues Message Objekt mit Hilfe einer Factory erzeugt, welches als Parameter den Typ „Message“ und die zu übertragende Textnachricht erhält. Dieses Objekt wird anschließend zu einem String gewandelt und an das gewünschte Ziel übertragen. Das Host Objekt ist nicht in die Basisklassen mit eingegliedert, also kein besuchbares Objekt. Eine Kommunikation zwischen Agent und Host erfolgt immer auf Basis von Textnachrichten. Hierfür stehen sowohl in der Host-Klasse, wie auch in der Agenten-Klasse, ein *MessageHandler* VI bereit, welches beim Empfang einer Textnachricht aufgerufen wird. Das *MessageHandler* VI des Agenten ist ein dynamic dispatch VI, welches beim Erstellen neuer Agentenklassen, Klassen die von HGF_AgentBaseClass geerbt haben, die Möglichkeit bietet, zusätzliche Textnachrichten verarbeiten zu können. Dies ermöglicht eine einfache Erweiterung der Agent-Agent-Kommunikation.

Die Message-Handler werden innerhalb des *runHost* VIs oder der Statemachine mit dem Message Objekt als Parameter aufgerufen. Innerhalb der Message-Handler wird dann die String-Nachricht des Message-Objektes ausgelesen und an eine Case-Struktur übergeben, die eine vordefinierte Aktion, zum Beispiel das Senden einer Antwort, implementiert.

Besucher

Das Senden eines Besuchers erfordert mehrere Schritte. Zuerst muss der benötigte Besucher erstellt werden. Dies kann entweder mit Hilfe einer Factory geschehen (dies empfiehlt sich, falls Besucher Parameter transportieren sollen) oder durch direkte Verwendung eines Besucherobjektes (falls die Standarddaten für eine Anwendung auf einen Agenten ausreichend sind). Anschließend wird dieser Besucher als Initialisierungsparameter für ein neues Message Objekt angegeben, welches mit Hilfe der Factory erstellt wird. Hierbei muss zusätzlich der Typ „Visitor“ als Parameter für das Message Objekt angegeben werden. Das so erzeugte Message Objekt wird mittels *Flatten To String* VI zu einem String umgewandelt und kann dann in die Shared Variable des gewünschten Agenten geschrieben werden.

Über den schon beschriebenen Eventmechanismus gelangt das Message Objekt in das Statechart, in welcher der Typ „Visitor“ ausgelesen wird, woraufhin die Zustandsmaschine in den Zustand „Visitor“ wechselt. Hier wird das LabVIEW Objekt aus dem Message-Objekt abgefragt und mittels typecast, LabVIEW stellt hierfür das *To More Specific* VI bereit, in ein Besucher Objekt gewandelt. Anschließend wird die *accept* Methode des internen Agentenobjektes, mit dem Besucher als Parameter, aufgerufen und so der Besucher auf den Agenten angewendet. Hierfür wird das Agentenobjekt aus den Zustandsdaten kopiert und nach erfolgreicher Beendigung wieder in den Zustandsdaten gespeichert.

Agentenmigration

Wie schon gezeigt wurde, erfolgt die Übertragung eines Agenten innerhalb eines Message Objektes auf dem normalen Kommunikationsweg. Um die Migration zu starten, wird ein Reisebesucher an den Agenten geschickt. Dieser wird, ebenso wie jeder andere Besucher auch, auf den Agenten angewendet. Der Reisebesucher bringt ein Ziel mit, welches in der *action* Methode über die Agentenmethode *set Destination* an den Agenten übertragen wird. Zusätzlich wird, wie schon erwähnt, in dem Attribut **nextState** der **travel**-Trigger eingestellt. Die vom Reisebesucher aufgerufenen Methoden sind nicht vom Typ dynamic dispatch, können also von Kind Klassen nicht überschrieben werden. Hierdurch wird sichergestellt, dass ein Reisebesucher auch bei unbekannten Agentenobjekten angewendet werden kann, ohne das ein unerwünschter Algorithmus hierdurch aufgerufen wird.

Ist travel?=**true** wechselt die Zustandsmaschine nicht in den „wait“-Zustand zurück, sondern wechselt in einen „travel“-Zustand. Hier erfolgt die eigentliche Migration. Zunächst wird das Ziel des Agenten ausgelesen. Anschließend wird eine Nachricht an den Ziel-Host gesendet, welche einen „travel-Request“ beinhaltet. Dies ist eine Anfrage an den Ziel-Host, ob dieser für den Empfang des Agenten bereit ist. Hierbei werden der Bibliotheksname und der Klassenname des Agenten übermittelt. Der Host-Message-Handler, welcher die Nachricht verarbeitet, reagiert, indem er zunächst überprüft, ob die maximale Zahl an Agenten schon erreicht ist. Ist dies nicht der Fall, so wird überprüft, ob der Agententyp, identifiziert über Klassenname und Bibliotheksname, auf dem Host zulässig ist. In einer Antwortnachricht erfolgt nun die Mitteilung an die Agenten-Statemachine ob eine Übertragung stattfinden kann.

Ist eine Übertragung möglich, so wird das Agentenobjekt in ein Message-Objekt gebettet und an den Ziel-Host übertragen. Anschließend wird die Statemachine und damit auch der Enginge-Task beendet. Das Agentenobjekt ist nunmehr nur noch auf dem Ziel-Host erreichbar.

Falls die Übertragung nicht stattfinden kann, so ist in den Agentenattributen vorgesehen, wie weiter verfahren werden soll. Entweder, die Ausführung wird lokal fortgesetzt, beendet, oder es wird versucht, den Agenten an ein alternatives Ziel zu senden.

Erweiterbarkeit

Das Agentensystem ist im laufenden Betrieb erweiterbar. Dies wird durch die generische Implementierung des Aktivierens von Agentenobjekten ermöglicht. Durch Erzeugung neuer Agentenklassen ist einer Erweiterung von Funktionalitäten oder das hinzufügen neuer Algorithmen zur Laufzeit möglich.

Um neue Agenten zu erzeugen, die das bestehende Agentensystem erweitern, muss eine Bibliothek mit den benötigten Klassen erstellt werden. Die benötigten Klassen sind hierbei die Klasse für das Agentenobjekt selbst, die Besucherklassen und eine zugehörige Fabrik. Wie eine Erzeugung eines neuen Agenten stattfindet, wird im Anhang „HowTo – Agentenerzeugung“ näher erläutert. Hierbei wird eine Zip-Datei erstellt, welche alle benötigten Klassen beinhaltet. Um einen Agenten auf einem Host verfügbar zu machen, muss diese Zip-Datei im Klassenverzeichnis des Hosts entpackt werden. In der Prototypenversion des Agentensystems muss dieses noch manuell geschehen. Das Zip-Format wurde hierbei gewählt, da LabVIEW Methoden für dessen Verarbeitung besitzt, was eine spätere Automatisierung vereinfacht. Um einen Agenten zu verwenden, müssen seine Klassen zur Laufzeit geladen werden. Dies muss geschehen, bevor der Agent übertragen werden kann, da eine Rekonstruktion des Message-Objektes, welches den Agenten transportiert, sonst nicht erfolgreich stattfinden kann. Das Laden der Klassen geschieht mit Hilfe einer Fabrik. Um dies zu ermöglichen existiert eine Namenskonvention für die Erzeugung neuer Agenten.

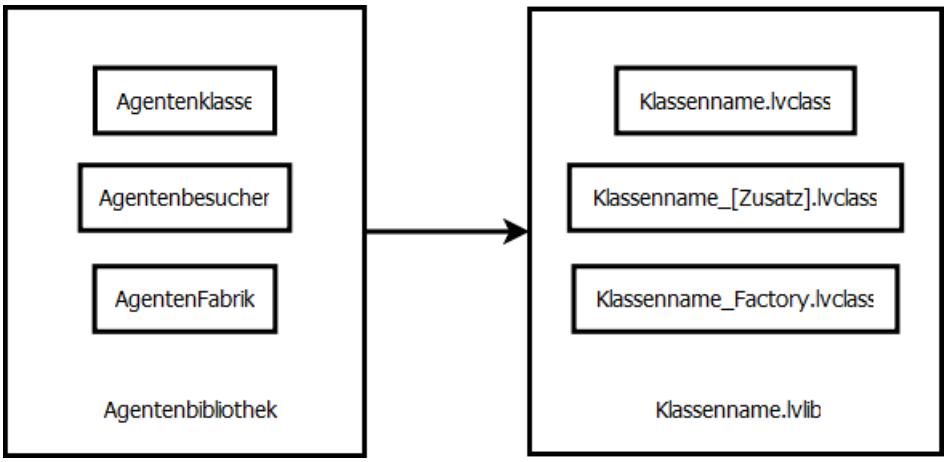

Abbildung 31: Namenskonvention für neue Agenten

Eine zugehörige Ordnerstruktur ergibt sich daraus, dass jede Klasse zusammen mit den Klassenmethoden in einem separaten Unterordner (vollständiger Klassenname ohne „.lvclass“) gespeichert wird. Das speichern der Klassenmethoden in separaten Ordnern ist notwendig, da unterschiedliche Klassen dieselbe dynamic dispatch Methode überschreiben können. So implementieren zum Beispiel alle Besucher ein *actions* vi. Da alle überschreibenden VIs den gleichen Namen haben müssen, können sie nur über eine Ordnerstruktur implementiert werden. Zusätzlich erhöht dies die Übersichtlichkeit über das gesamte System.

Um einen Host dazu zu veranlassen, die Klassen eines Agenten in den Arbeitsspeicher zu laden, wird eine Message mit einem „travel request“, eine Migrationsanfrage, an den Host gesendet. Ein „travel request“ ist ein Message-Objekt, welches den Objektnamen „travel request“ besitzt und in seinem Textnachirchten-Attribut den Klassennamen des Agenten enthält. Der Message-Handler des Hosts reagiert auf eine Migrationsanfrage, indem er zuerst die Klasse überprüft. Dies wird im Kapitel „Sicherheitskonzept“ noch einmal erläutert. Anschließend wird der Klassenname verwendet, um die Klasse in den Arbeitsspeicher zu laden. Dies geschieht mit Hilfe des *Get LV Class Default Value* VIs, welches über eine Pfadangabe ein Objekt einer Klasse erzeugen kann. Durch die Namenskonvention ist es möglich, programmatisch den Pfad der Fabrik zu bestimmen und diese zu laden. Da die Fabrik Objekte aller zur Bibliothek gehörenden Klassen beinhalten sollte, werden so alle Klassen in den Arbeitsspeicher geladen. War dies erfolgreich, so überprüft der Message-Handler zusätzlich ob der Host noch Agenten starten kann, oder ob bereits die Maximalzahl an Threads erreicht ist. Anschließend wird eine Antwort gesendet, ob die Migration stattfinden kann oder nicht.

Falls ein Agent nicht direkt von der Agenten-Basisklasse geerbt hat, also noch Generationen dazwischen liegen, so müssen diese ebenfalls manuell per „travel request“ in den Speicher geladen werden, bevor das eigentliche Agentenobjekt erfolgreich übertragen werden kann.

Sicherheitskonzept

Das Thema Sicherheit spielt in einem mobilen Agentensystem eine wichtige Rolle. Ein Agentensystem zeichnet sich dadurch aus, dass zur Laufzeit eine im Grunde beliebige Erweiterbarkeit ermöglicht wird. Ein Host eines Agentensystems ist in der Lage, auf einem Rechner programmatisch eine Software zu starten, deren Funktion er nicht kennt. Hierbei besteht die Gefahr, dass es sich mitunter auch um Software, also Agenten handelt, die einen für den Rechner unerwünschten oder sogar schädlichen Programmcode enthält. Zusätzlich ist es denkbar, dass ein Agent sensible Daten mit sich bringt, welche nicht jedem verfügbar sein sollen. Der Zugriff auf ein Agentenobjekt ist, bedingt durch Arbeitsumgebung und die für den Zugriff implementierten Methoden, vor unerwünschtem Zugriff schützbar. Die Aufgabe, einen laufenden Agenten vor unerwünschtem Zugriff zu schützen, liegt bei dem Programmierer des speziellen Agenten, da dies über die von ihm bereitgestellten Methoden geschehen muss. Bei einer Migration ist es jedoch so, dass sämtliche Daten, auch und speziell sensible Daten, auf einen Host übertragen werden. Es muss also sichergestellt werden, dass ein Abhören der Kommunikation, hier vor allem der Migration von Agenten, keinen Rückschluss auf sensible Daten oder sogar die Rekonstruktion eines vollständigen Agentenobjektes ermöglicht. Weiterhin muss sichergestellt werden, dass eine Manipulation von Daten während der Übertragung nicht möglich ist, um auch hier das Gefahrenpotential, also unerwünschte Effekte durch manipulierte Agenten, zu minimieren.

Sicherheit bei der Kommunikation

Die Kommunikation erfolgt bei der Verwendung von Shared Variables über eine öffentliche Kommunikationsschicht. Daten, die über Shared Variables gesendet werden, sind für jeden Rechner im Netzwerk frei zugänglich. Hinzu kommt, dass eine Zeichenkette, die mit Hilfe des *Flatten To String VIs* aus einem Objekt erstellt worden sind, sämtliche Daten offenlegen. Die eigentlich privaten Attribute werden unverfälscht in der Zeichenkette angezeigt und sind mit einfachsten Mitteln zu rekonstruieren. Enthält ein Objekt in seinen (stets privaten) Attributen eine Zeichenkette, so zum Beispiel ein Passwort, so ist diese nach ausführen des *Flatten To String* VIs unverfälscht in der Ausgabezeichenkette enthalten. Da auch Agentenobjekte über diesen Weg transportiert werden, ist dies besonders kritisch. Aus diesem Grunde ist es unumgänglich die zu übertragenden Daten zu verschlüsseln und somit vor ungewolltem Zugriff zu schützen.

LabVIEW selbst stellt keine Verschlüsselungsmechanismen zur Verfügung. Da die Implementierung eines eigenständigen sicheren Verschlüsselungsverfahrens sehr aufwändig ist, wird auf eine bestehende Software zurückgegriffen. Über die LabVIEW Kommandozeilenfunktionen lassen sich die Funktionen des, unter der Gnu Public License veröffentlichten Programmes, „Gnu Privacy Guard“[37], kurz GPG, in LabVIEW einbinden.

Eine Schnittstelle zwischen den HGF Basisklassen und dem Programm GPG wird durch die HGF_GPG Klassen implementiert. Da die HGF Basisklassenbibliothek, sowie auch das mobile Agentensystem, unter der Gnu Public License veröffentlicht werden, entstehen hierbei keine lizenzrechtlichen Probleme.

[37] (Free Software Foundation)

Die HGF_GPG Klasse

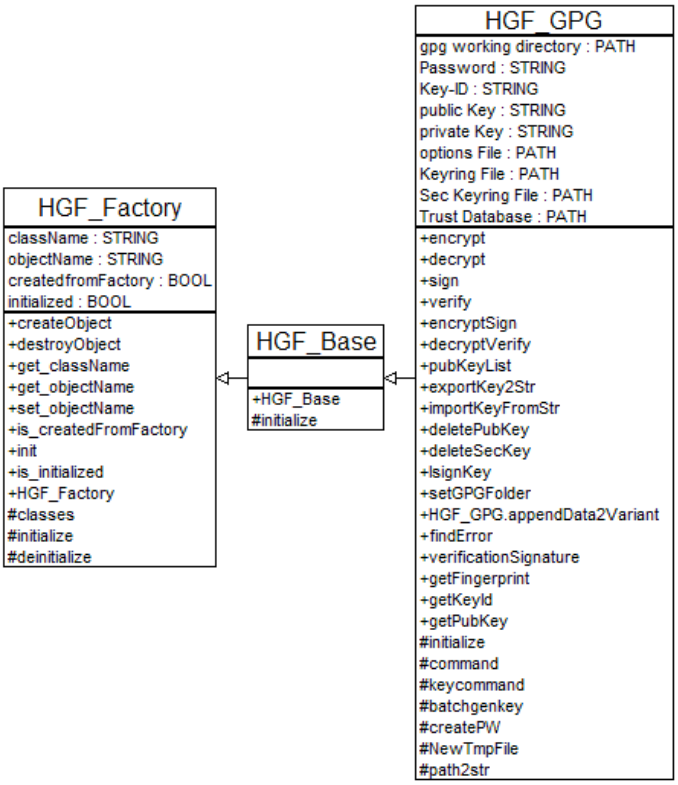

Abbildung 32: Klassendiagramm der HGF_GPG Klasse

Die Attribute der HGF_GPG Klasse enthalten die Pfade zu den lokalen Schlüsselring-Dateien, sowie die für das Arbeiten mit GPG wichtigen Daten. Diese sind die eigene ID[38], der öffentliche und der private Schlüssel, sowie das Passwort.

Die Methoden, die von der Klasse bereitgestellt werden, entsprechen den Funktionen von GPG. Es werden Methoden zur Schlüsselverwaltung und Methoden zur Verschlüsselung beziehungsweise Signierung bereitgestellt.

Instanziierung

Ein Objekt wird, wie in den HGF Basisklassen üblich, erzeugt, indem zuerst die notwendigen Parameter mit Hilfe des *appendData2Variant* VIs angegeben werden. Mit Hilfe der Factory wird anschließend ein Objekt erzeugt und initialisiert.

[38]Die ID wird bei der Schlüsselerzeugung automatisch mit erzeugt

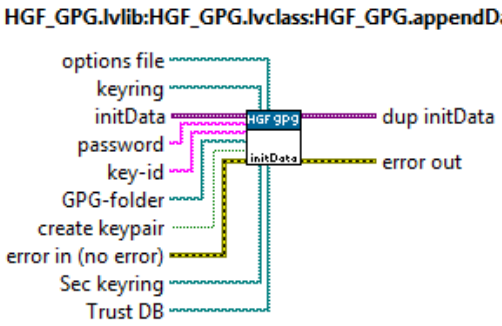

Abbildung 33: Ein- und Ausgabeparameter des HGF_GPG appendData2Variant VIs

Über die Initialisierungsparameter lassen sich die nötigen Pfade einstellen. Soll ein vorhandenes Schlüsselpaar verwendet werden, muss das Passwort und die ID angegeben werden, da sonst ein Error zurückgegeben wird. Über **create keypair** wird festgelegt, ob ein neues Schlüsselpaar erzeugt werden soll.

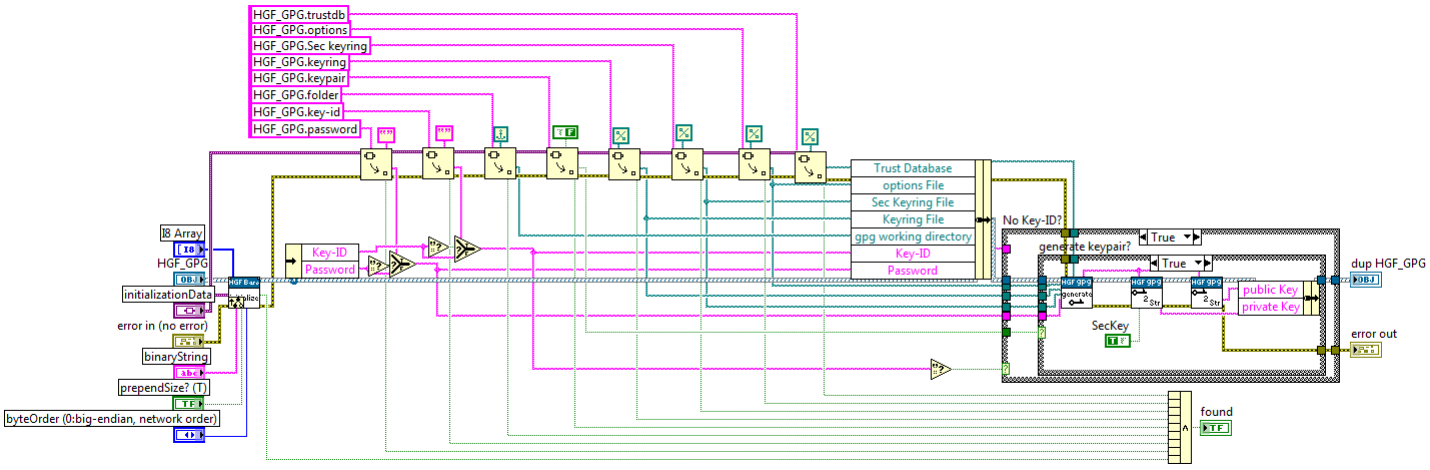

Abbildung 34: Blockdiagramm des HGF_GPG initialize VI's

Während der Initialisierung werden die Pfade in die Objektattribute übernommen. Für den Fall dass ein neues Schlüsselpaar angelegt werden soll, wird das private VI „batch genKey“ aufgerufen, innerhalb dessen über einen Kommandozeilenaufruf ein neues Schlüsselpaar erzeugt wird. Für den Fall dass ein Passwort angegeben ist, wird dieses für das neue Schlüsselpaar verwendet. Andernfalls wird hierfür ein Zufallswert erzeugt. Das „initialize“ VI fragt zusätzlich noch die öffentlichen und privaten Schlüssel ab.

Um mit GPG arbeiten zu können, ist es nötig, seine Schlüssel verwalten zu können. Durch die HGF_GPG Klasse werden Methoden bereitgestellt, mit denen sich die eigenen Schlüssel als String exportieren, neue Schlüssel sich hinzufügen oder alte Schlüssel sich löschen lassen. Ebenso ist es möglich, einen Schlüssel lokal zu signieren und sich eine Liste der Schlüssel des öffentlichen Schlüsselbundes ausgeben zu lassen. Zusätzlich zu den Schlüsselverwaltungsmethoden stehen noch Methoden bereit, um die eigentlichen Aufgaben dieser Klasse zu erfüllen: Signieren, verifizieren, verschlüsseln, dekodieren.

Alle Funktionen dieser Klasse basieren auf dem gleichen Prinzip. Über das LabVIEW „System Exec“ VI wird ein Kommandozeilenaufruf ausgeführt. Dieser ruft die Datei gpg.exe mit entsprechenden Parametern auf. Exemplarisch wird dies für die Funktion „verschlüsseln und signieren“ gezeigt.

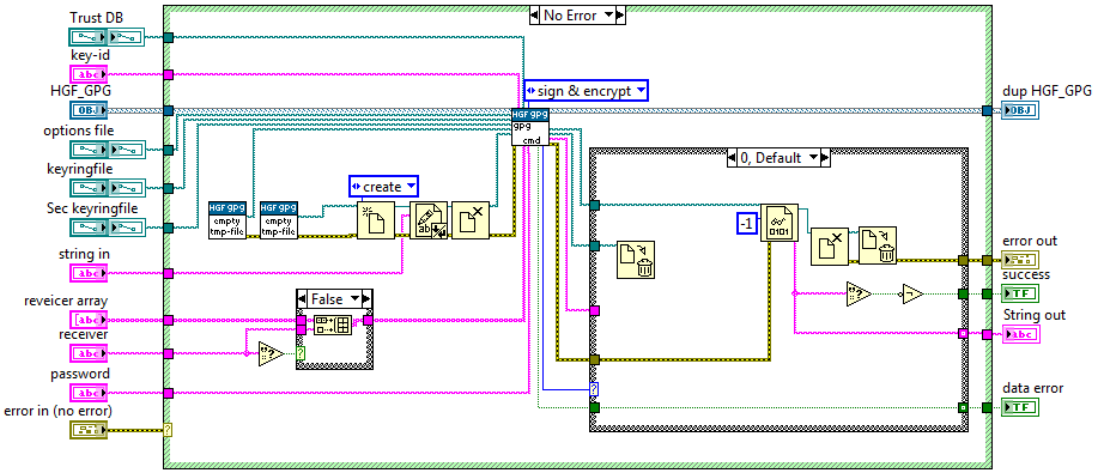

Abbildung 35: Blockdiagramm des HGF_GPG "encryptSign" VI's

Eine Hauptanwendung von GPG ist die Verschlüsselung von Kommunikation. Da diese häufig, so auch im Agentensystem, über Strings stattfindet, wird auch bei den HGF_GPG Klassen mit diesem Datentyp gearbeitet.

Mit GPG werden lokale Dateien verschlüsselt. Der zu verschlüsselnde String wird mittels des „write to text file“ VIs in eine neue Datei innerhalb des temporären Verzeichnisses geschrieben. Der Pfad zu einer zweiten temporären Datei dient GPG als Ausgabepfad. Dateien werden durch GPG mit Hilfe des öffentlichen Schlüssels verschlüsselt und können nur mit den entsprechenden privaten Schlüsseln wieder entschlüsselt werden. Bei der Verschlüsselung müssen also alle Empfänger angegeben werden. Zusätzlich ist es möglich, andere als die bei der Initialisierung angegebenen Pfade für die lokalen GPG Dateien, sowie Passwort und die gewünscht Key ID für die Signierung anzugeben. Dies sorgt dafür, dass auch ein nicht initialisiertes GPG-Objekt zur Verschlüsselung eingesetzt werden kann.

Das VI ruft schließlich das Sub VI *command* auf. Hier werden alle Parameter zu einem Kommando zusammengeführt. Bei leeren Inputs werden die Objektparameter verwendet. Schließlich erfolgt der eigentliche Kommandozeilenaufruf.

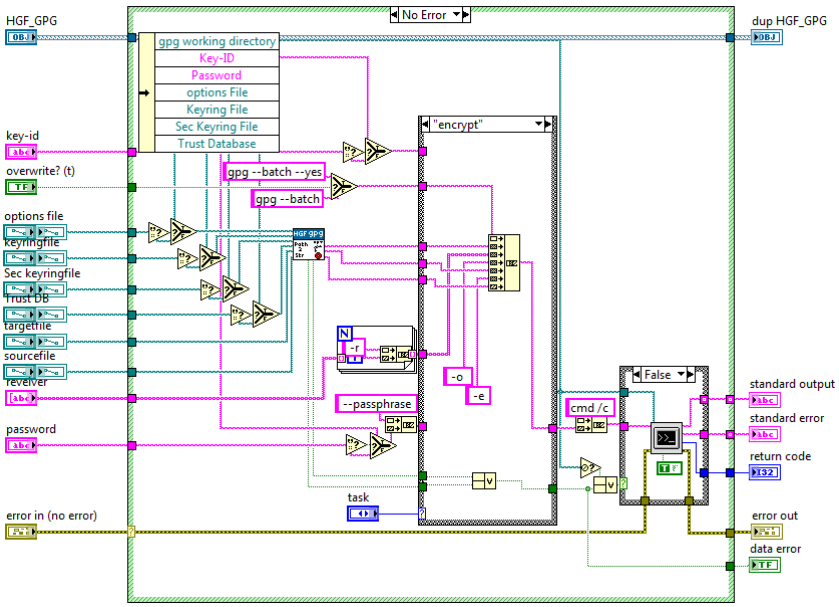

Abbildung 36: Blockdiagramm HGF_GPG:command.vi

Nach erfolgreicher Verschlüsselung wird nun die Zieldatei binär ausgelesen und als String ausgegeben. Die temporären Dateien werden gelöscht. Der Ausgabestring kann nun an die Empfänger übertragen werden, die ihn mittels des privaten Schlüssels wieder entschlüsseln können.

Mit Hilfe der HGF_GPG Klasse lässt sich lokal das Programm GPG verwenden, welches mit den auf der Festplatte vorliegenden Schlüsselringen arbeitet. Um jedoch eine sichere Kommunikation innerhalb des Agentensystems zu gewährleisten, ist es nötig, dass jeder Agent über ein eigenes Schlüsselpaar und einen eigenen Schlüsselring verfügt und somit in der Lage ist, Nachrichten für andere zu verschlüsseln, aber auch verschlüsselte Nachrichten zu empfangen. Diese Daten muss jeder Agent auch bei einer Migration behalten. Dies zu ermöglichen erfordert eine Erweiterung der HGF_GPG Klassen.

Hierfür existiert die MoAgSy_GPG Klasse. Diese hat von der HGF_GPG-Klasse geerbt. Die Schlüsselringe, Vertrauensdatenbank und die Optionen-Datei werden hier jedoch in den Attributen der Klasse als Strings verwahrt. Um dennoch das Programm GPG benutzen zu können, werden die dynamic dispatch VIs command und keycommand von dieser Kind-Klasse überschrieben. Bevor mittels „call Parent“ VI die Methode der HGF_GPG Klasse aufgerufen wird, werden die Strings in temporäre Dateien geschrieben, welche nach Beendigung der Methode wieder ausgelesen und anschließend gelöscht werden. Auf diese Weise kann jedes MoAgSy_GPG Objekt in seinen Attributen über eigene GPG Arbeitsdaten verfügen, die nur für die Nutzung in temporären Dateien lokal vorliegen.

Jedes Agentenobjekt verfügt in seinen Attributen über ein MoAgSy_GPG Objekt, welches bei der Initialisierung des Agentenobjektes selbst instanziiert wird. Die bei der generischen Schlüsselerzeugung innerhalb der Initialisierung des MoAgSy_GPG Objektes erzeugte ID ermöglicht zusätzlich eine eindeutige Identifizierung jedes Agentenobjektes.

Eine gesicherte Übertragung des Agentenobjektes verläuft nach folgendem Schema:

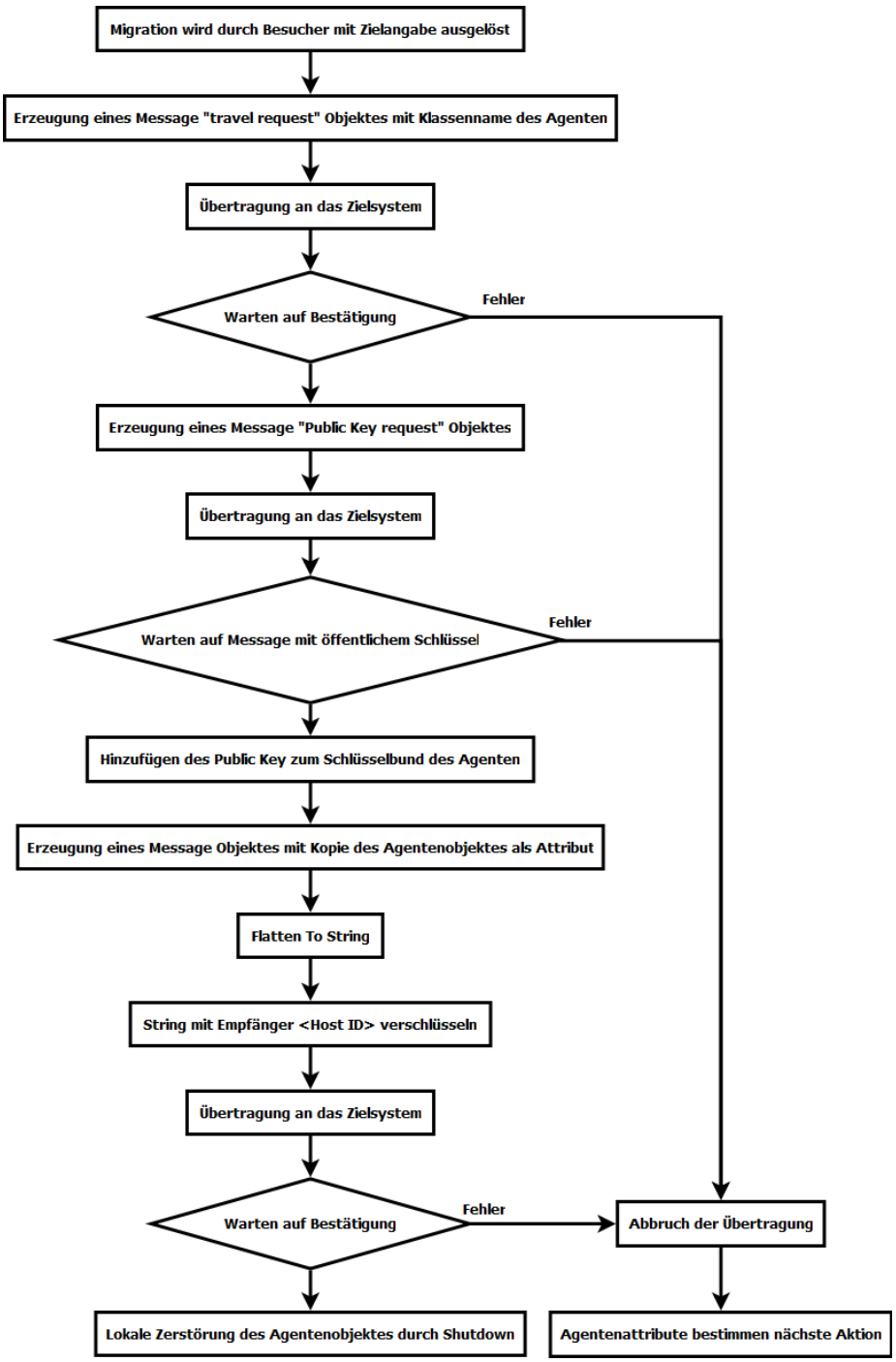

Abbildung 37: Schema der Agentenmigration

Sicherheit des Hostsystems

Mit welchen Mitteln eine Kommunikation abgesichert werden kann wurde soeben gezeigt. Ein anderer Aspekt der Sicherheit betrifft die Ausführung neuer Agenten.

Eine der Stärken eines mobilen Agentensystems liegt in seiner Flexibilität. Das System ist zur Laufzeit dynamisch erweiterbar. Ein laufender Host bietet hierbei Programmen eine Arbeitsumgebung, ohne deren Funktion zu kennen. Dies geschieht, indem programmatisch und generisch Agenten in einem Thread gestartet werden.

Dieser separate Thread unterliegt in LabVIEW keinerlei Restriktionen. Hieraus ergibt sich nicht nur eine Stärke des Systems, sondern auch eine Gefahr. Ein Agent hat, sobald er erst einmal gestartet wurde, vollen Zugriff auf das System. Agenten sind in der Lage auf die lokalen Festplatten zuzugreifen und können zum Beispiel, durch aufrufen von Schleifenfunktionen, die lokalen Ressourcen vollständig verbrauchen.

Folgende Ansätze werden erläutert:

- VI Hierarchie
- Web of Trust
- Blacklist/Whitelist

VI Hierarchie

Mittels VI-Server Methoden ist es möglich die Abhängigkeiten eines VIs abzufragen. Hierfür muss mittels *open VI reference* das zu untersuchende VI über einen lokalen Pfad referenziert werden. Die Abhängigkeiten können dann mittels *invoke node* Methode ausgelesen werden. Aus diesen Abhängigkeiten lassen sich Rückschlüsse darauf ziehen, welche VIs durch das zu untersuchende VI aufgerufen werden. Ein Vergleich mit einer Liste von unerwünschten VIs kann die Ausführung anschließend erlauben oder verhindern.

Im Rahmen des mobilen Agentensystems ist diese Methode nicht vorgesehen. Es ist hiermit zwar möglich, eine Liste aller Abhängigkeiten abzufragen, jedoch zählt hierbei vor allem das *sysexec* VI zu den kritischen VIs. Dieses wird jedoch innerhalb der GPG-Klasse für die Kommandoaufrufe genutzt. Eine rekursive Abfrage einzelner Abhängigkeiten wäre zwar möglich, ist jedoch sehr zeitaufwändig. Des Weiteren müsste die Überprüfung für jedes VI einer Agentenbibliothek vorgenommen werden, was einen nicht zu vertretenden Zeitaufwand darstellen würde.

Web of Trust (WOT)

Das mobile Agentensystem nutzt die Verschlüsselung mittels GPG-Schnittstelle. GPG arbeitet mit öffentlichen und privaten Schlüsseln. Die öffentlichen Schlüssel können hierbei bedenkenlos über öffentliche Schnittstellen verbreitet werden, was auch für das Agentensystem genutzt wird. Um die Authentizität der Schlüssel zu überprüfen, wird bei GPG das WOT als Verfahren eingesetzt. Hierbei werden die öffentlichen Schlüssel signiert, um deren Echtheit zu bestätigen. Ist ein Schlüssel von vertrauenswürdigen Stellen signiert worden, so wird er ebenfalls als Vertrauenswürdig angesehen.

Im Rahmen des mobilen Agentensystems ist das WOT als Verfahren zum Schutz eines Hostsystems bisher nicht vorgesehen. Hierbei handelt es sich um ein Verfahren, welches einzelne Schlüssel und somit bei programmatischer Schlüsselerzeugung einzelne Objekte authentifizieren kann. Für die Sicherung eines Hosts ist dies jedoch irrelevant, da auf jedes Agentenobjekt einer Klasse der identische Code angewendet werden kann. Es bedarf also eines Verfahrens, welches Agenten-Klassen beziehungsweise -Bibliotheken und nicht einzelne Objekte auf einem Host erlaubt oder verbietet.

Blacklist/Whitelist

Blacklist[39] und Whitelist[40] stellen ein Verfahren dar, um mit Hilfe von Vergleichslisten explizit etwas zu erlauben oder zu verbieten. Im Rahmen des Prototyps für das mobile Agentensystem sind beide Listen für eine eventuelle spätere Verwendung vorgesehen, jedoch wird aktuell nur eine Whitelist angewendet. Um einen Agenten auf einem Host starten zu können, muss die Bibliothek des Agenten in die Whitelist des Hosts eingetragen sein. Diese wird sowohl bei einem „travel request“, als auch bei der Ankunft eines Agenten vor dessen Ausführung, überprüft. Die Abfrage der Agentenbibiliothek erfolgt vor der Übertragung an den Host, innerhalb der Initialisierung des Message Objektes, durch Auslesen des Klassenpfades des übergebenen Objektes. Da die Ausführung eines Objektes nur stattfinden kann, wenn die entsprechenden Klassen in den Arbeitsspeicher geladen werden, besteht durch diese Methode ein effektiver Schutz vor unerwünschter Ausführung von Klassen.

[39] Schwarze Liste / Verbotsliste
[40] Weiße Liste / Erlaubnislist

Sicherheit des Agentenobjektes

Soeben wurde demonstriert, durch welche Methoden ein Hostsystem vor unerwünschten Objekten geschützt werden kann. Ein weiterer Aspekt der Sicherheit betrifft die Sicherheit des Agenten. Wird ein Agent an einen Host übermittelt, so ist er darauf angewiesen, dass dieser das Agentenobjekt korrekt mit Hilfe eines Enginge-Tasks startet. Hierbei darf der Zugriff auf das Agentenobjekt ausschließlich innerhalb der Zustandsmaschine des Engine-Tasks erfolgen wodurch gewährleistet wird, dass keine Kopien von Objekten erzeugt werden und damit kein unerwünschter Zugriff auf die Objektattribute stattfindet. Ein bösartiges Hostsystem könnte genau an dieser Stelle angreifen, indem es entweder vor dem Start eines Agenten ein Duplikat des Objektes erzeugt oder das Objekt lediglich empfängt, nicht jedoch aktiviert. Da die Übertragung eines Agenten einem festgelegten Algorithmus unterliegt, ist ein bösartiges Hostsystem, welches das Objekt empfängt aber nicht startet, mit einfachen Mitteln zu realisieren.

Eine Möglichkeit, einen Agenten vor einer Übertragung an ein bösartiges Hostsystem zu schützen, bietet hier ein WOT. Hierbei kann, bevor ein Agent zu einem Host übermittelt wird, der öffentliche Empfängerschlüssel zur Verifikation des Ziels eingesetzt werden. Nur falls der öffentliche Schlüssel von vertrauenswürdigen Schlüsseln signiert wurde, erfolgt die Übertragung an das Zielsystem.

Der Schutz des Agentenobjektes ist bisher nicht implementiert. Dies liegt daran, dass im Rahmen des Prototyps eine Migration durch Übertragung eines Reisebesuchers an das Agentenobjekt ausgelöst wird. Dieser Reisebesucher bringt in seinen Attributen die Informationen zum Migrationsziel mit. Die Übertragung auf ein anderes System ist also immer bewusst ausgelöst und das Ziel-System vorher bekannt.

Realisierung eines mobilen Agentensystems in Form eines Prototypen

Host Applikation

Die Host-Applikation dient, wie schon erwähnt, sowohl der lokalen Verwaltung der Agenten, als auch als Benutzerschnittstelle. Das Frontpanel zeigt hierbei über einen Statusbildschirm (Abbildung 38) die ID nummern der aktuell laufenden Agenten an. Diese werden bei der Erzeugung der Schlüsselpaare mit erstellt und dienen einem Agenten als Namen. Hierüber wird auch der Name der lokalen Benutzerschnittstelle festgelegt. Weiterhin wird der Status des ThreadPools angezeigt, der die Benötigten Worker-Threads für die Ausführung der Agenten startet. Über die Funktion „kill Agent“ ist es möglich, gezielt einzelne Agenten zu beenden.

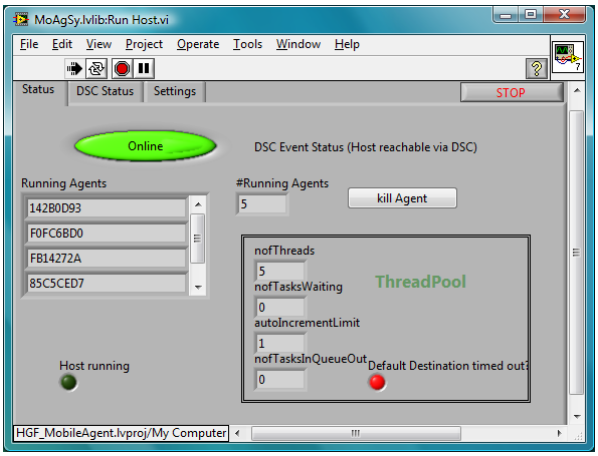

Abbildung 38: Frontpanel der Host-Applikation - Statusanzeige

Der Reiter DSC Status zeigt zum einen den Status des separat für die Kommunikation bereitgestellten ThreadPools sowie die aktuell bei dem DSC-Monitor registrierten Variablen an. Im Reiter Settings (Abbildung 39, Abbildung 40) können diverse Einstellungen verändert werden. Die Einstellungen im Reiter „Blacklist“ und „FTP-Settings“ dienen der späteren Ergänzung des Hosts um eine explizite Sperrfunktion für Agentenklassen und die programmatische Ergänzung von lokalen Agenten durch herunterladen der entsprechenden Dateien von einem FTP-Server. Der Reiter „General Settings“ (Abbildung 39) zeigt Name, ID und die maximale Zahl an Agenten des Hosts an. Der Reiter „Whitelist“ (Abbildung 40) enthält die Liste der für den Host explizit erlaubten Klassen. Der Prototyp muss hierin alle Agentenklassen enthalten, die auf ihm lauffähig sein sollen. Mit Hilfe der *Change General Settings* beziehungsweise der *Edit Whitelist* Kontrolle wird ein separates VI aufgerufen, das das Editieren der Einstellungen erlaubt. Die ID des Hosts kann dabei jedoch nicht geändert werden, da diese an das bei der Host-Erzeugung erstellte Schlüsselpaar gebunden ist.

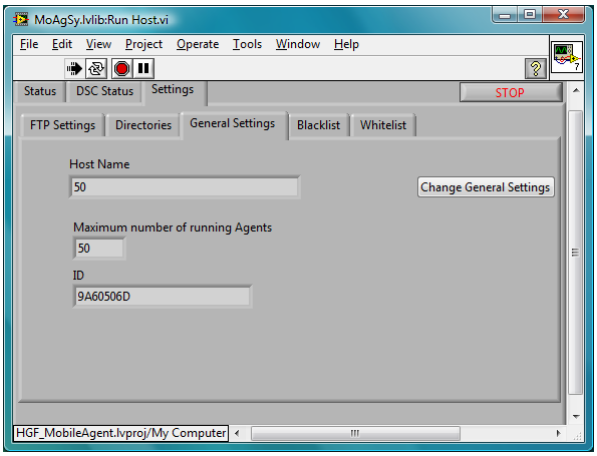

Abbildung 39: Frontpanel der Host-Applikation – "General Settings"

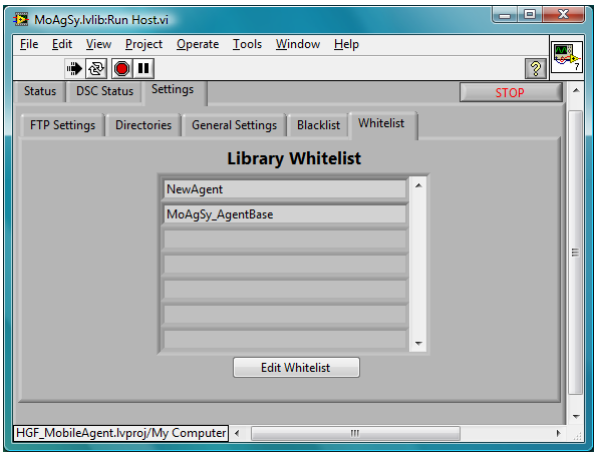

Abbildung 40: Frontpanel der Host-Applikation - "Whitelist"

Das Blockdiagramm enthält den eigentlichen Programmcode. Zuerst wird der Host initialisiert. Hierbei können die Einstellungen entweder von Festplatte geladen werden (die Speicherung des Host Objektes im Arbeitsverzeichnis erfolgt während der Deinitialisierung standardmäßig) oder ein neues Objekt mit den entsprechenden Einstellungen initialisiert werden. Anschließend wird die DSC-Initialisierung vorgenommen. Hierbei wird ein separater ThreadPool für die Kommunikation erstellt, je ein Task für einen DSC-Monitor und einen DSC-Variablenmanager gestartet und die Kommunikationsschnittstelle mit dem Host-Namen erzeugt.

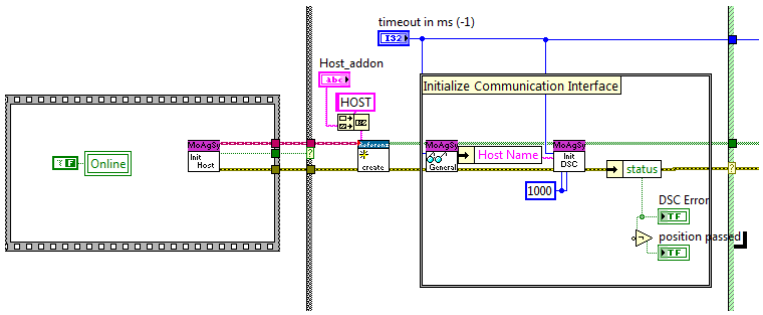

Abbildung 41: Blockdiagramm Host-Applikation - Initialisierung

Ist die Initialisierung erfolgreich verlaufen, so werden die verschiedenen Threads der Applikation gestartet. Dieses ist jeweils eine While-Schleife, die die entsprechenden Funktionalitäten beinhalten. Aufgrund der Multithreadingfähigkeit von LabVIEW lassen sich diese Threads innerhalb der Host-Applikation parallel darstellen. Synchronisiert werden sie mittels einer Okkurrenz, über die Verwendung der Methoden der *HGF_Reference* Klasse haben alle Threads Zugriff auf das Host-Klassenobjekt.

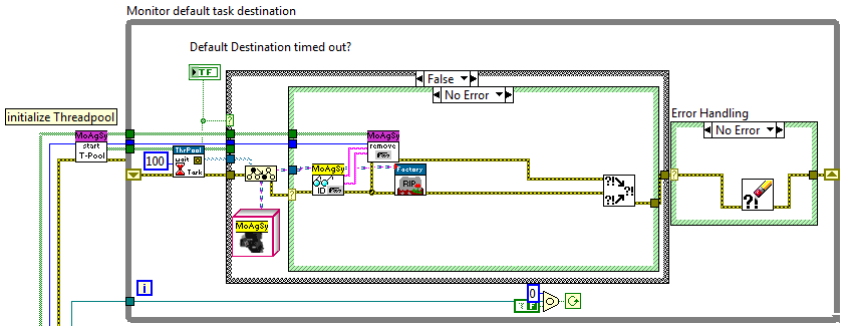

Abbildung 42: Blockdiagramm Host-Applikation - ThreadPool Monitor

Der erste Thread (Abbildung 42) initialisiert und überwacht den ThreadPool, der für die Worker zum Starten der Agentenobjekte zuständig sein soll. Hier werden die Attribute des ThreadPools abgefragt, die über das Frontpanel für den Anwender sichtbar sein sollen. Zusätzlich werden hier zurückkommende Tasks (TaskOut Queue) deinitialisiert und die Agenten aus der Liste der laufenden Agenten entfernt. Ein weiterer Thread (Abbildung 43) widmet sich dem ThreadPool, der für die Kommunikation zuständig ist. Hier wird regelmäßig ein Besucher zum Monitor-Task geschickt, der die aktuell registrierten Shared Variables abfragt und anschließend mit diesen zur Host-Applikation zurückgesendet wird. Diese werden anschließend über das Frontpanel im Reiter „DSC Status“ angezeigt.

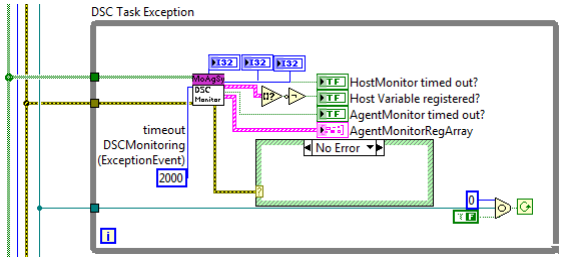

Abbildung 43: Blockdiagramm Host-Applikation – DSC ThreadPool Monitor

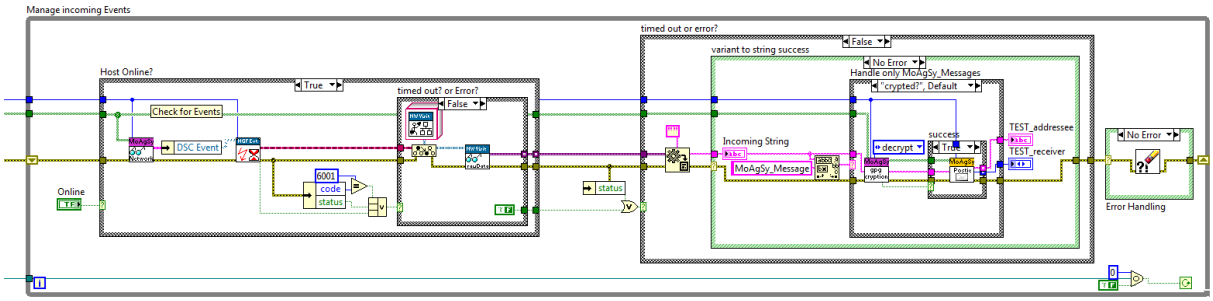

Abbildung 44: Blockdiagramm Host-Applikation – Incoming Messages

Abbildung 44 zeigt den dritten Thread. Hierin werden die einkommenden Nachrichten verarbeitet. Über die *wait* Funktion der HGF_Event Klasse wird ein DSC Ereignis abgefragt, welches über einen Besucher beim Monitor für die Host-Variable registriert wurde. Falls ein Ereignis stattgefunden hat, so sollte dies mit Hilfe eines Netzwerkbesuchers den aktuellen Wert der Variablen zurückliefern. Als Wert wird ein MoAgSy_Message Objekt erwartet. Nach einer *Flatten To String* Operation ist der Name des Objektes im Klartext enthalten. Ist dies nicht der Fall, so handelt es sich entweder um keine dem System konforme Nachricht, also keine Nachricht die mittels Message Objekt transportiert wird, oder um eine verschlüsselte Nachricht. Aus diesem Grund wird, falls der eingegangene String nicht die Zeichenkette „MoAgSy_Message“ enthält, zuerst ein Versuch der Entschlüsselung unternommen, bevor das Objekt mittels *Unflatten From String* VI rekonstruiert wird. Ist die

Rekonstruktion nicht erfolgreich, so wird die Nachricht ignoriert, andernfalls wird der Empfänger der Nachricht ausgelesen. Nur falls es sich dabei um den Namen des Hosts handelt, wird die Nachricht weiter behandelt. Dies geschieht, indem sie einer internen Queue angehängt wird. Diese Queue dient als Zwischenpuffer und der Übertragung der eingehenden Nachrichten an einen separaten Thread für die Verarbeitung (Abbildung 45).

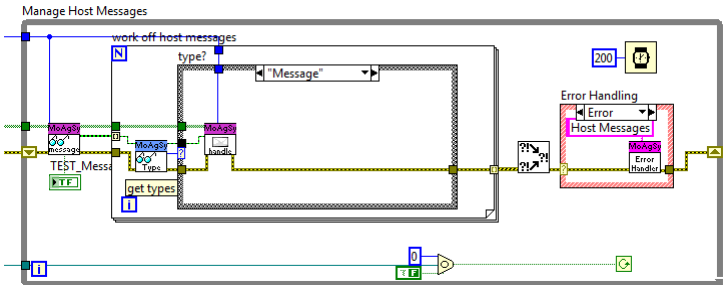

Abbildung 45: Blockdiagramm Host-Applikation - Nachrichtenverarbeitung

In diesem wird der Typ der Nachricht ausgelesen und entsprechend des Typs die weitere Verarbeitung vorgenommen. Handelt es sich um eine Textnachricht, so wird, wie in der Abbildung gezeigt, der Message Handler des Hosts aufgerufen. Handelt es sich um einen Besucher, so wird eine Fehlerantwort gesendet, da es sich bei einem Host Objekt nicht um ein besuchbares Objekt handelt. Ist die Nachricht vom Typ „mobile Agent“, so wird untersucht, ob die Agentenklasse auf dem Host zum Start berechtigt ist und ob der Host noch weitere Agenten aufnehmen kann (die Maximalzahl wird durch die Einstellungen für den ThreadPool begrenzt).

Der fünfte Thread (Abbildung 46) dient der Verwaltung der Benutzerschnittstelle. Hier wird mittels eines Ereignisdiagramms auf Benutzereingaben gewartet. Findet ein Benutzerereignis statt, so wird hierüber das entsprechende VI für die Aktion gestartet (zum Beispiel editieren der „Whitelist“). Falls eine Auszeit erfolgt, so wird der Status des Hosts aktualisiert. Wird durch den Benutzer der Host beendet, so wird anschließend an diesen Prozess eine Okkurrenz gesetzt, die alle anderen Threads beendet.

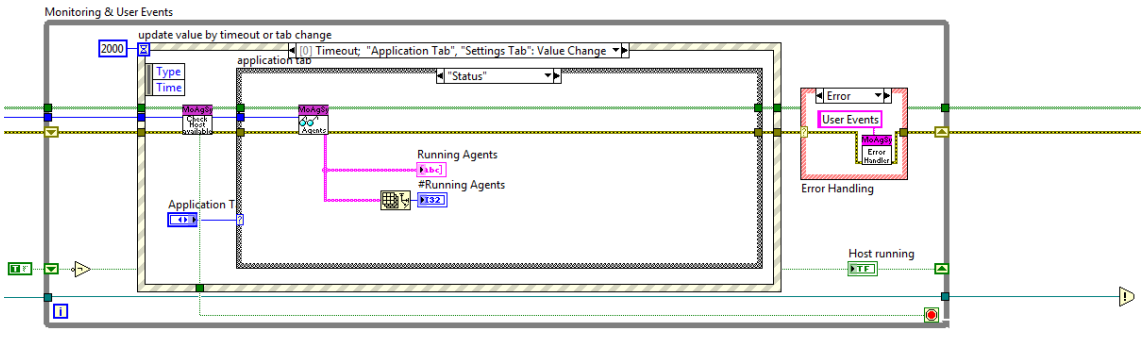

Abbildung 46: Blockdiagramm Host-Applikation - GUI Verwaltung

Ist ein Host beendet worden, so findet im Anschluss an die Threads eine Deinitialisierung statt (Abbildung 47). Hierbei werden die beiden ThreadPools beendet, die erzeugten Shared Variables zerstört und eine Kopie des Host-Objektes auf der Festplatte gespeichert. Diese kann bei einem Neustart des Hosts für die Initialisierung verwendet werden.

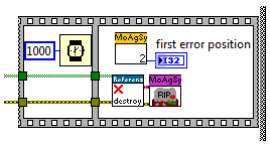

Abbildung 47: Blockdiagramm Host-Applikation - Deinitialisierung

Agenten Basisklassen

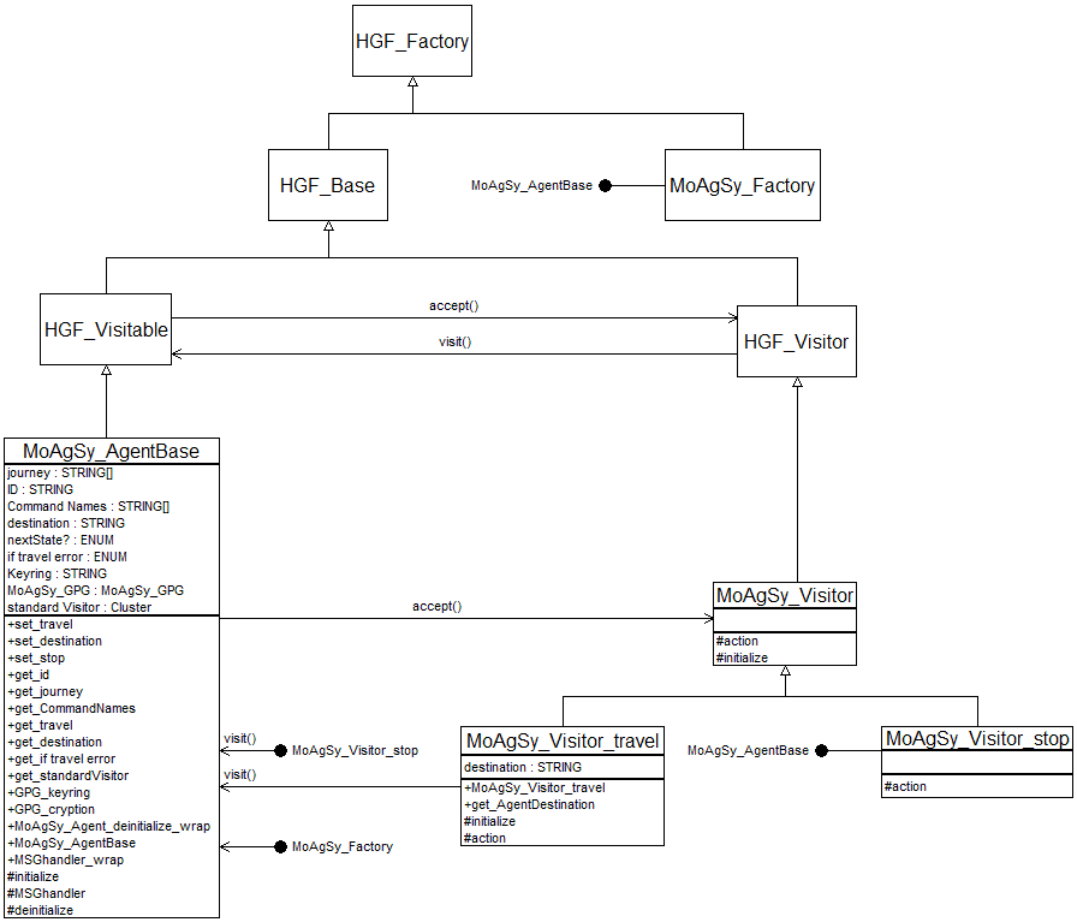

Abbildung 48: UML Klassendiagramm der Agenten Basisklassen

Die Klasse MoAgSy_AgentBase hat von HGF_Visitable geerbt. Jedes Agentenobjekt, das mit dem mobilen Agentensystem verwendet werden soll, muss von dieser Klasse geerbt haben, da vor dem Start eines Objektes dieses über eine Typenwandlung zu einem Basisagenten gewandelt wird. Die Funktionen dieser Klasse implementieren die Verwendung von GPG innerhalb des Agentensystems sowie die grundlegenden Funktionen des Message Handlers.

Die MoAgSy_Visitor Klasse repräsentiert alle Besucher für einen Agenten. Innerhalb der Zustandsmaschine findet bei eingehenden Besuchereignissen eine Typenwandlung auf diese Klasse statt, sie ist also die Ahnenklasse für alle Agentenbesucher. Die beiden spezialisierten Agentenbesucher MoAgSy_Visitor_travel und MoAgSy_Visitor_stop führen beide Standardfunktionen der Agenten Basisklasse aus und können auf jeden Agenten angewendet werden. Hierbei setzt der Stop Besucher das Attribut *nextState?* auf „Shutdown“, wodurch das Herunterfahren des Agenten ausgelöst wird. Der Reisebesucher wird bei der Initialisierung mit einem Ziel versehen. Dieses übergibt er an den Agenten und setzt zusätzlich das Attribut *nextState?* Auf „travel“, was eine Migration veranlasst.

Wie ein spezieller Agent implementiert werden kann, der weitergehende Funktionen zur Verfügung stellt, ist im Anhang ab Seite 76 näher erläutert. Wie ein Agent schließlich gestartet wird, wird im Folgenden Absatz „Starten eines Agenten“ erklärt.

Starten eines Agenten

Die Initialisierung eines spezialisierten Agenten setzt mitunter die Kenntnis von relevanten Parametern voraus. Die Festlegung von Namenskonventionen für diese Parameter kann dazu verwendet werden, eine Einheitliche Initialisierungsroutine für eine Gruppe von Agenten zu implementieren, in dem mittels VI-Server Methoden die Parameter der für die Initialisierungsdaten des Agenten zuständigen *appendData2Variant* VIs auszulesen und zum Beispiel mit Hilfe von Datenbanken diese automatisch zu setzen. Dies gehört jedoch in den Bereich der Entwicklung spezieller Agenten. Für die Agenten-Basisklasse ist die Festlegung von Namenskonventionen für Parameter nicht geschehen. Die Aufgabe, Methoden für eine Initialisierung bereit zu stellen, übernimmt somit der Programmierer spezieller Agenten. Ist ein Agentenobjekt initialisiert, so kann der Agent gestartet werden indem das Objekt an einen Host übermittelt wird. Die Übermittlung kann hierbei nach dem Standardschema für eine Agentenübertragung (Abbildung 37) stattfinden. Die verschlüsselte Übertragung wird hierbei empfohlen, ist allerdings nicht zwingend vorgeschrieben. Für die Übertragung eines Agenten nach dem Standardschema stellt die Fabrik der Agenten-Basisbibliothek eine Methode bereit.

Engine Task

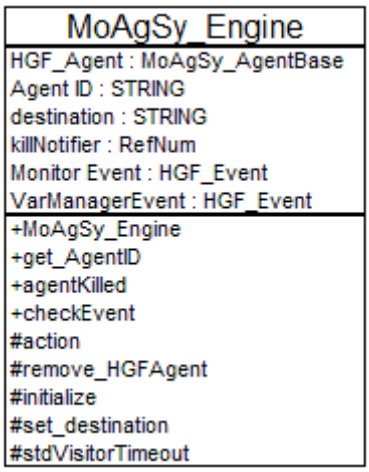

Abbildung 50 Klassendiagramm MoAgSy_Engine:

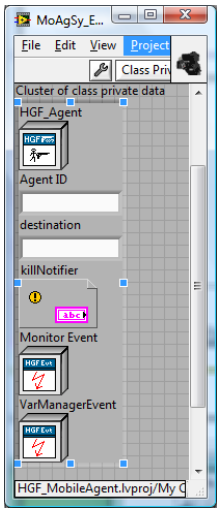

Abbildung 49: Private Attribute MoAgSy_Engine

Die MoAgSy_Engine Klasse hat direkt von Task geerbt und ist speziell für den Einsatz mit mobilen Agenten konzipiert. Eine Host Applikation startet einen Agenten, indem mit Hilfe einer Fabrik und der appendData2Variant Methode ein neuer Task mit den entsprechenden Attributen erzeugt und anschließend mit der *dispatchTask* Methode des ThreadPools in die TaskIn-Queue eingefügt wird.

In den Attributen enthält der Engine Task anschließend das Agentenobjekt inklusive der zugehörigen ID. Zusätzlich wird noch die Okkurrenz zur Beendigung des Agenten sowie die Triggerereignisse für den DSC Monitor und Variable Manager übergeben. Hierdurch kann der Task beendet werden und eine Registrierung der für die Kommunikation benötigten Shared Variable innerhalb des Tasks erfolgen. Das String-Attribut **destination** dient dazu, dem Host nach Beendigung des Tasks über ein das Ziel des Agenten zu informieren.

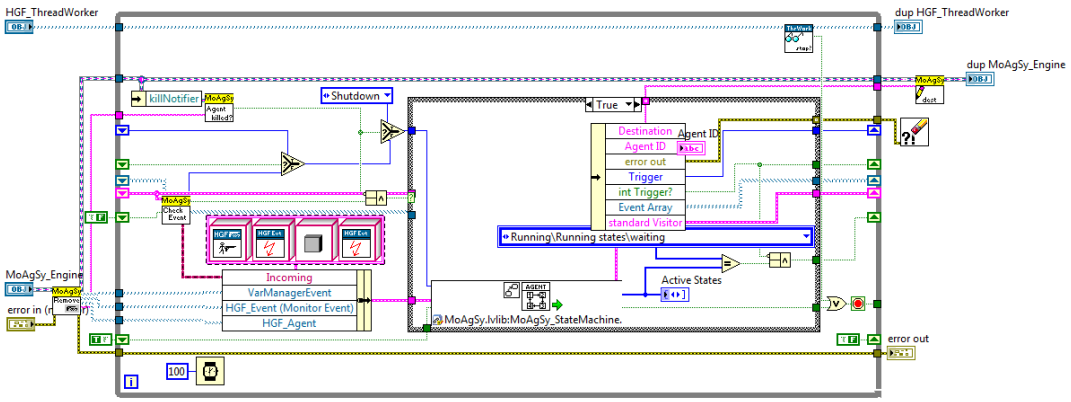

Abbildung 51: Blockdiagramm MoAgSy_Engine:action.vi

Wird ein Engine Task durch einen Worker aktiviert, so werden die benötigten Attribute ausgelesen. Die eigentliche Funktion ist innerhalb einer Schleife implementiert. Beim ersten Aufruf der Zustandsmaschine wird der Parameter **init?** auf True gesetzt wodurch die Zustandsmaschine zurückgesetzt wird. Mit Hilfe eines Shift Registers wird bei jeder weiteren Iteration ein False übergeben. An dieser Stelle kann das bereits erwähnte *first call?* VI nicht eingesetzt werden. Ein Task ist zwar eintrittsinvariant, arbeitet jedoch mit geteilten Klonen. Dies hat zur Folge, dass eine Instanz nach Beendigung seiner Ausführung erneut verwendet werden kann. In diesem Fall würde das *first call?* VI dennoch ein False zurückgeben, wodurch die Zustandsmaschine nicht zurückgesetzt werden würde.

Bei der ersten Iteration wird das Agentenobjekt sowie die DSC Triggerereignisse an die Zustandsmaschine übergeben und dort in ihren Zustandsdaten gespeichert. Die weitere Initialisierung erfolgt innerhalb der Zustandsmaschine, die anschließend die registrierten Ereignisse an den Task ausgibt. Mittels Shift Registern werden diese für die nächsten Iterationen verfügbar gemacht. Der Task übernimmt hierbei die Aufgabe, auf eingehende Events zu warten und die Stop Okkurrenz sowohl des Hosts, wie auch des Workers abzufragen. Nur im Falle eines Ereignisses wird das Zustandsdiagramm ausgeführt. So lange kein Ereignis stattgefunden hat, befindet sich der Agent also in einem Schlafmodus. Über das Einstellen eines Standardbesuchers mit einer zugehörigen Auszeit kann jedoch ein regelmäßiges Ereignis ausgelöst werden. Hierfür verfügt der Engine Task über ein SubVI, welches vor jeder Ereignisabfrage ausgeführt wird und abfragt, ob die Auszeit überschritten wurde.

Der Engine Task wird beendet, sobald die Zustandsmaschine ihre Ausführung beendet hat. Falls eine Stop Okkurrenz gesetzt wurde, so wird durch den Task ein Shutdown Trigger an die Zustandsmaschine übergeben, woraufhin ein Agent noch in der Lage ist, Aktionen für das kontrollierte Herunterfahren auszuführen.

Zustandsmaschine

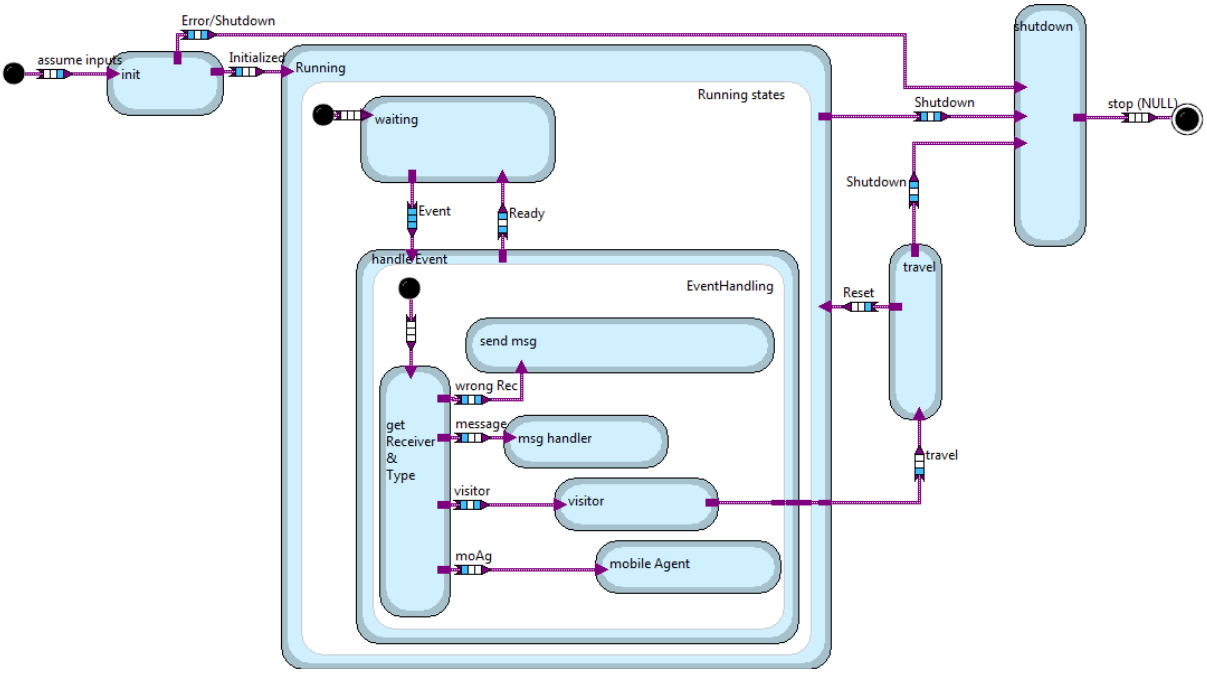

Abbildung 52: Zustandsmaschine des mobilen Agentensystems

Die Zustandsmaschine übernimmt bei der Initialisierung sowohl das Agentenobjekt als auch die Triggerereignisse für den lokalen DSC Monitor und den DSC Variablen Manager. Diese werden in den internen Zustandsdaten gespeichert. Anschließend wechselt sie in den Zustand **init**. Bei Eintritt in diesen Zustand werden zunächst die ID sowie der Standardbesucher des Agenten ausgelesen. Anschließend wird mit Hilfe der DSC Trigger eine Shared Variable für den Agenten erzeugt und beim Monitor registriert. Ist die Initialisierung nicht erfolgreich verlaufen, so wird der Agent wieder beendet. War die Initialisierung erfolgreich wechselt die Zustandsmaschine in den Zustand **running** und das registrierte Event so wie der Standardbesucher werden an den Engine Task ausgegeben, der auf eingehende Ereignisse beziehungsweise die Auszeit für den Standardbesucher wartet. Der Zustand **running** besitzt weitere Unterzustände. Bei Eintritt in diesen Zustand geht die Zustandsmaschine in den Subzustand **waiting** über. Bei eingehenden Ereignissen wird ein Ereignistrigger an die Zustandsmaschine übergeben, welcher einen Wechsel in den Subzustand handle Event bewirken kann.

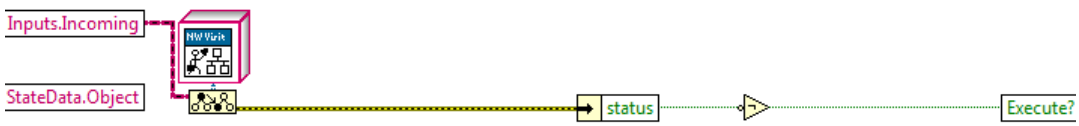

Abbildung 53: Wächteraktion der Zustandsmaschine bei einem eingehenden Ereignis

Als Wächteraktion überprüft die Zustandsmaschine, ob ein Netzwerkbesucher, welcher den neuen Wert der Shared Variablen in seinen Attributen mitbringen sollte, als Eingang anliegt. Nur dies der Fall ist findet der Übergang in den Zustand handle Event statt.

Während des Zustandsüberganges wird die Message aus dem Netzwerkbesucher rekonstruiert. Falls hierbei ein Fehler erfolgt, so wechselt die Zustandsmaschine anschließend wieder in den Subzustand **waiting**. Falls die Rekonstruktion erfolgreich verlaufen ist, o befindet sich die Zustandsmaschine nun im Subzustand **handle Event**. Hier wird zunächst der Typ und Empfänger des Message Objektes rekonstruiert und entsprechend des Typs ein Trigger ausgelöst. Falls der Empfänger nicht mit dem Agenten übereinstimmt, so wird die Nachricht ignoriert und, falls gefordert, eine Fehlerantwort gesendet. Ist der Empfänger korrekt, so wechselt die Zustandsmaschine je nach Typ in den Zustand **msg_Handler** bei eingehenden Textnachrichten, **Visitor** bei eingehenden Besuchern oder mobile Agent falls ein **mobiler Agent** empfangen wurde. Falls das Message Objekt vom Typ mobiler Agent ist, so wird eine dynamic dispatch Methode, *incoming Agent,* der Basisagentenklasse aufgerufen. Hierüber können in spezialisierten Agenten auch andere Agenten empfangen und weiter verarbeitet werden. Dies kann verwendet werden, falls zum Beispiel aufwändigere Rechenoperationen mit Hilfe eines dafür konzipierten Agenten ausgelagert werden sollen. Hat dieser „Rechenagent“ seine Berechnung erledigt, so kann er mitsamt dem Ergebnis wieder zum Ursprünglichen Agenten zurückkehren und dort weiterverarbeitet werden.

Falls das eingegangene Message Objekt vom Typ Textnachricht ist, so wird der Message Handler des Agenten, wiederum repräsentiert durch eine dynamic dispatch Methode, mit dem eingegangenen Message Objekt als Parameter aufgerufen. Neben den Standardnachrichten welche der Basisagent implementiert hat können so weitere Nachrichten durch spezialisierte Agenten verarbeitet werden.

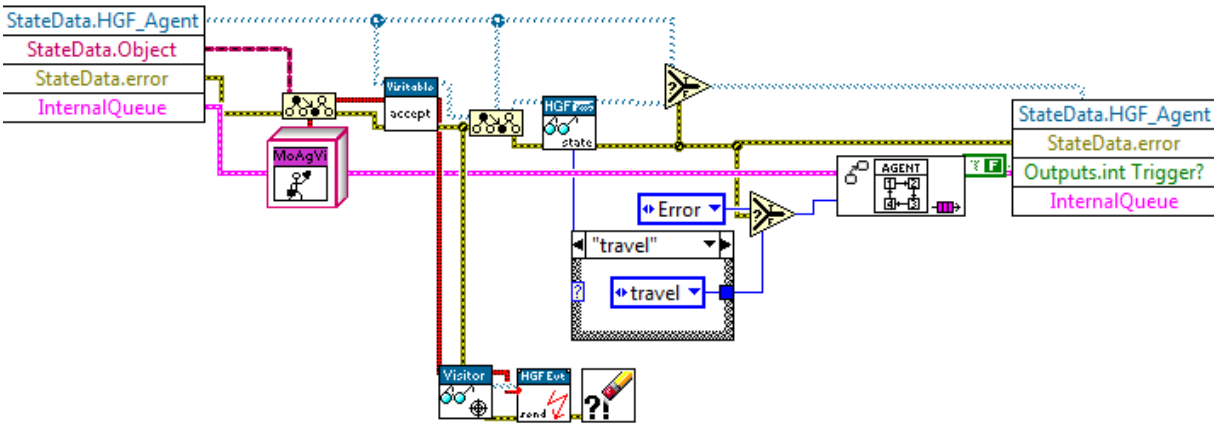

Abbildung 54: Aktion der Zustandsmaschine nach Erhalt eines Besuchers

Wurde ein Message Objekt vom Typ Besucher empfangen, so wird während des Überganges zum Subzustand **Visitor** das Objekt der Message abgefragt und in die internen Zustandsdaten übernommen. Anschließend wird die in (Abbildung 54) gezeigte Aktion ausgeführt. Hierbei wird nach einer Typenwandlung des Besuchers die *accept* Methode aufgerufen, also der Algorithmus des Besuchers auf den Agenten angewendet. Anschließend wird das **nextState?** Attribut des Agentenobjektes abgefragt und darüber der nächste Zustand für die Zustandsmaschine bestimmt. Anschließend wechselt die Zustandsmaschine in den entsprechenden Zustand.

Nachdem ein Besucher verarbeitet wurde, sind die Zustände **wait**, **travel** und **shutdown** möglich. Ein Übergang in den Zustand **wait** versetzt den Agenten wieder in den Schlafmodus bis vom Engine Task das nächste Ereignis detektiert und die Zustandsmaschine erneut gestartet wird. Der Übergang in den Zustand **shutdown** kann sowohl durch einen Stop Besucher erfolgen, als auch über den Engine Task direkt getriggert werden, was erfolgt nachdem eine der Stop Okkurrenzen gesetzt wurde. Hierin

wird die Kommunikationsschnittstelle deinitialisiert und das *deinitialize* VI des mobilen Agenten ausgeführt. Durch Überschreiben desselben kann von spezialisierten Agenten ein Algorithmus für kontrolliertes Beenden implementiert werden. Über das **nextState?** Attribut des Agenten lässt sich zusätzlich erkennen, ob der Agent selber den Übergang ausgelöst hat oder der Host hierfür Verantwortlich ist.

Nachdem ein Reisebesucher auf den Agenten angewendet wurde, geht die Zustandsmaschine in den Zustand **travel** über. Hierin wird die Migration nach bekanntem Schema (Abbildung 37) realisiert. Verläuft die Migration erfolgreich, so findet ein Übergang zum **shutdown** Zustand statt. Falls die Migration hierbei nicht erfolgreich verläuft, so entscheidet der Agent über das weitere Vorgehen. Im Fehlerfall kann die lokale Ausführung fortgesetzt, der Agent dennoch beendet oder das dynamic dispatch VI *travel error* aufgerufen werden. In diesem VI können spezialisierte Agenten eine eigenständige Routine für den Fehlerfall implementieren.

Fazit und Ausblick

Auf Grundlage der HGF Basisklassen wurde ein Prototyp für ein mobiles Agentensystem entworfen und implementiert. Dieser Prototyp zeigt, dass das generische Laden von Klassen zur Laufzeit möglich ist und Objekte, die als Entitäten behandelt werden sollen, über den bereits bestehenden ThreadPool generisch und programmatisch aktiviert werden können. Der Transport von Objekten über ein Netzwerk stellt sich mit Hilfe der *Flatten To String* und *Unflatten From String* VIs als problemlos dar, falls die Klasse am Zielort bekannt ist. Über Skalierbarkeit und Performanz kann bisher keine Aussage getroffen werden, da entsprechende Tests noch ausstehend sind.

Die Verwendung von Shared Variables hat sich an manchen Stellen als problematisch herausgestellt. Fehler in der Verarbeitung von Shared Variables haben zum Teil zu häufigen Systemabstürzen geführt, deren Ursache noch nicht genau geklärt werden konnte. Die Übertragung von der Entwicklungsumgebung auf eine Laufzeitumgebung hat sich ebenfalls als nicht trivial herausgestellt. Eine zusätzliche Kommunikationsschicht bieten die HGF_Dim Klassen, die sich in die vorhandene Ereignisstruktur einbetten lassen sollten.

Die Implementierung der GPG-Klasse stellt für einen Prototypen ein geeignetes Verfahren zur Verschlüsselung und somit Absicherung der Kommunikation dar. Das in der bisherigen Implementierung mittels Kommandozeilenfunktionen ausgeführte Verschlüsselungsverfahren ist jedoch keine Praxistaugliche Lösung. Unterschiedliche Versionen von GPG werden hierbei zwar unterstützt, da eine identische Syntax für die Kommandozeilenaufrufe verwendet wird, jedoch können unterschiedliche Sprachversionen zu Fehlern in der Interpretation der Rückgabeparameter führen. Die Erzeugung von Schlüsselpaaren bei der Initialisierung eines Agenten führt dazu, dass eine programmatische Erzeugung der Agenten sehr Zeitaufwändig werden kann.

Die Verarbeitung von Message-Objekten basiert auf der Interpretation des Typs als Parameter des Objektes. Mit Hinblick auf eine spätere Erweiterbarkeit sollte dies durch einen generischen Ansatz ersetzt werden.

Ein Agent kann bisher nur an ein explizites Ziel migrieren. Er ist nicht in der Lage, sich ein Ziel für die Migration selbständig zu suchen. Um dies zu ermöglichen ist zum Beispiel die Verwendung eines Host-Servers denkbar, bei dem sich ein Host während des Starts anmeldet und in regelmäßigen Abständen eine Liste der ausführbaren Agenten sowie der freien Kapazität für zusätzliche Agenten übermittelt.

Um einen Eindruck über den nötigen Programmieraufwand zukünftiger Projekte zu bekommen, ist die Implementierung einer Agentensoftware für Bilderfassung und Bildanalyse geplant.

Anhang

How-To: Erzeugen eines neuen Agenten

Um eine neue Bibliothek für einen Agenten zu erzeugen, muss zuerst ein neues Projekt angelegt werden. Diesem Projekt fügt man die MoAgSy_Base Bibliothek hinzu. Hierin sind die Basisklassen des Agenten, des Agentenbesuchers und der Fabrik enthalten.

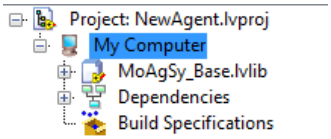

Abbildung 56: Agentenerzeugung 1 - neues Projekt

Anschließend wird eine neue Bibliothek mit den benötigten Klassen darin erstellt. Im Beispiel sieht man einen Agenten **NewAgent**, zwei Besucher **NewAgent_Visitor1** und **NewAgent_Visitor2,** sowie die dazugehörige Fabrik **NewAgent_Factory**. Das Benennungsmuster für die Fabrik, also

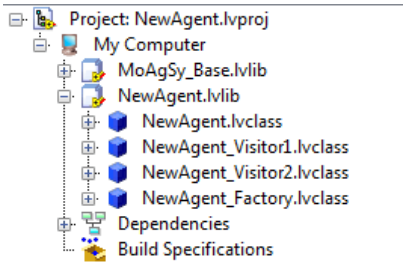

Abbildung 55: Agentenerzeugung 2 - Bibliothek

„**Klassenname_Factory**“ ist hierbei verpflichtend, da über sie die Klassen später in den Speicher geladen werden, wonach sie für eine generische Verarbeitung zur Verfügung stehen. Für die Restlichen Klassen und Methoden empfiehlt es sich, jede Klasse innerhalb eines separaten Ordners zu speichern. Dies erhöht zum Einen die Übersichtlichkeit, zum Anderen werden dynamic dispatch Methoden mit identischem Namen versehen, was innerhalb eines gemeinschaftlichen Ordners nicht möglich ist.

Anschließend werden die Vererbungsprozeduren über die jeweiligen Klasseneigenschaften (Rechtsklick auf die Klasse->Properties) festgelegt, wonach die Implementierung der Methoden und Attribute erfolgt. In diesem Fall erhält der Agent ein Numeric Attribut, das bei der Initialisierung festgelegt werden kann. Als Schnittstellen besitzt der Agent zwei öffentlichen VIs, eine Lese- und eine Schreibmethode für das Numeric-Attribut. Der erste Besucher enthält, im überschriebenen *action* VI, die Abfrage des Numeric-Attributes, welches er anschließend an den Anwender ausgibt (Abbildung 59). Der zweite Besucher hat ebenfalls ein Numeric, welches er bei der Initialisierung übergeben bekommt. In seiner *action* Methode liest er das Agentenattribut aus, addiert seinen eigenen Wert hinzu und schreibt dies als neuen Wert in das Agentenattribut (Abbildung 58). Wichtig ist hierbei, dass jeder Besucher den Agententyp, auf den er angewendet werden soll, kennen muss. Über eine Typenwandlung auf den entsprechenden spezialisierten Agenten wird es erst möglich, die

gewünschte Methode aufzurufen. Die Fabrik überschreibt das *classes* VI und enthält hierin jeweils ein Objekt der restlichen Klassen (Abbildung 57).

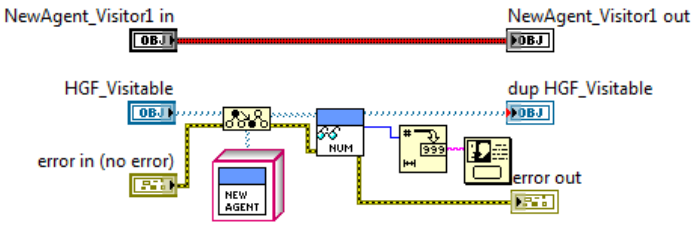

Abbildung 59: Agentenerzeugung 3 - Blockdiagramm NewAgent_Visitor1:action.vi

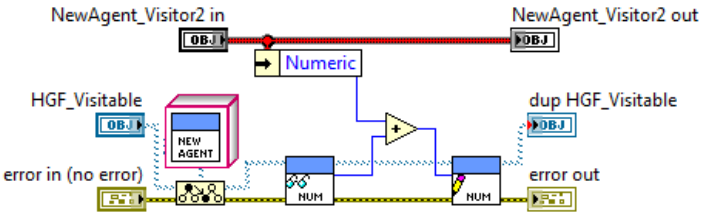

Abbildung 58: Agentenerzeugung 4 - Blockdiagramm NewAgent_Visitor2:action.vi

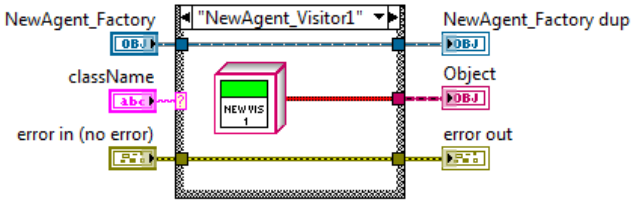

Abbildung 57: Agentenerzeugung 5 - Blockdiagramm NewAgent_Factory:classes.vi für classname="NewAgent_Visitor1"

Sind alle Methoden implementiert, sollte das Projekt ähnlich Abbildung 60 aussehen.

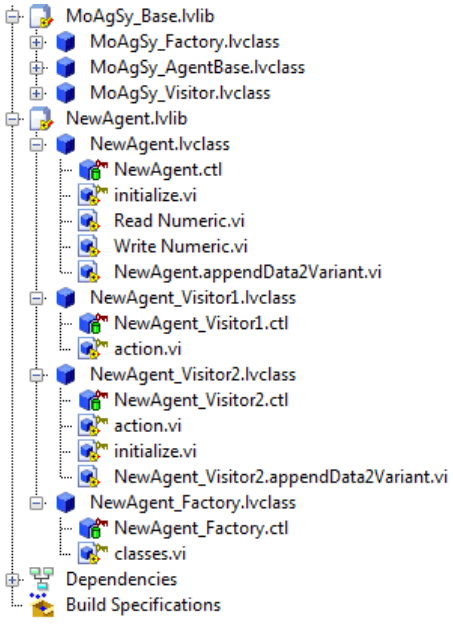

Abbildung 60: Agentenerzeugung 6 - Projekt nach Implementierung der Methoden

An dieser Stelle ist der neue Agent schon Einsatzbereit. Um eine spätere Verwendung auch auf anderen Hosts zu vereinfachen, wird die Bibliothek zu einer Zip-Datei gepackt. Über Rechtsklick auf „Build Specifications“→new→Zip-File lässt sich diese erstellen. Im Folgenden werden die Einstellungen für diese Zip-Datei vorgenommen.

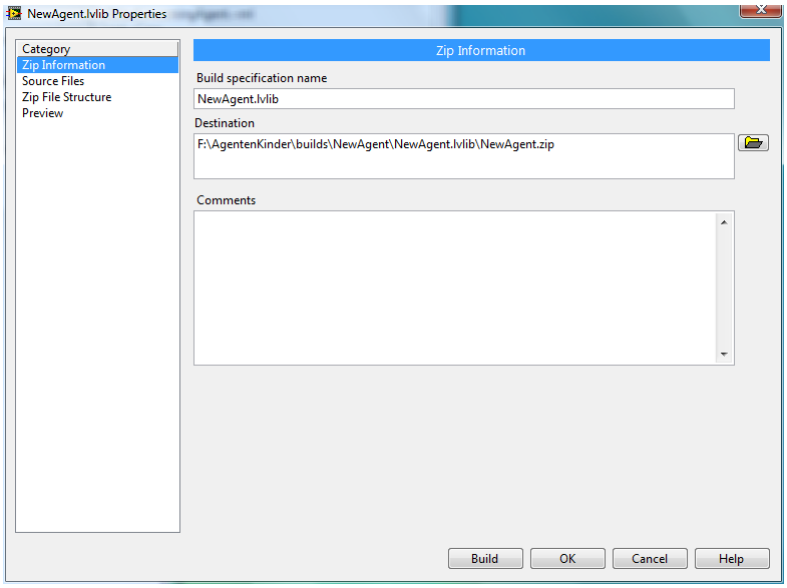

Abbildung 61: Agentenerzeugung 7 – Zip Einstellungen – Zip Information

In den Zip-Informationen wird der Name der Zip Datei sowie der Speicherort festgelegt. Der Zip-Dateiname sollte hierbei dem Klassennamen entsprechen.

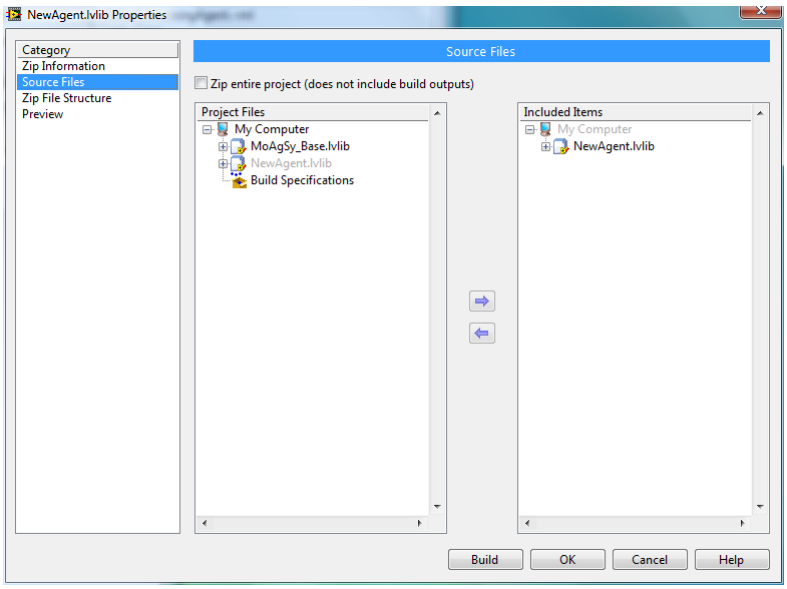

Abbildung 62: Agentenerzeugung 8 – Zip Einstellungen - Sourcefiles

In den Quelldateien wird NUR die neu erstellte Bibliothek inkludiert.

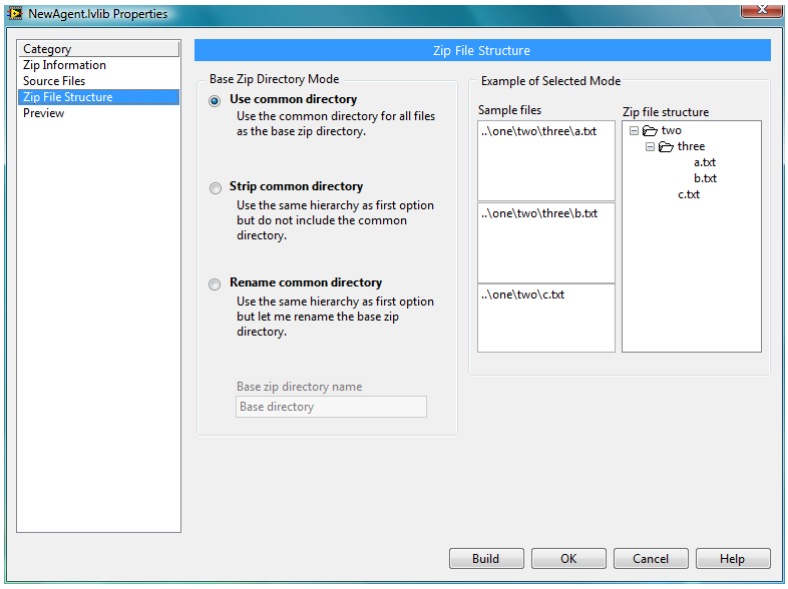

Abbildung 63: Agentenerzeugung 9 - Zip Einstellungen - Zip File Structure

Als Dateistruktur wird „common directory“ gewählt.

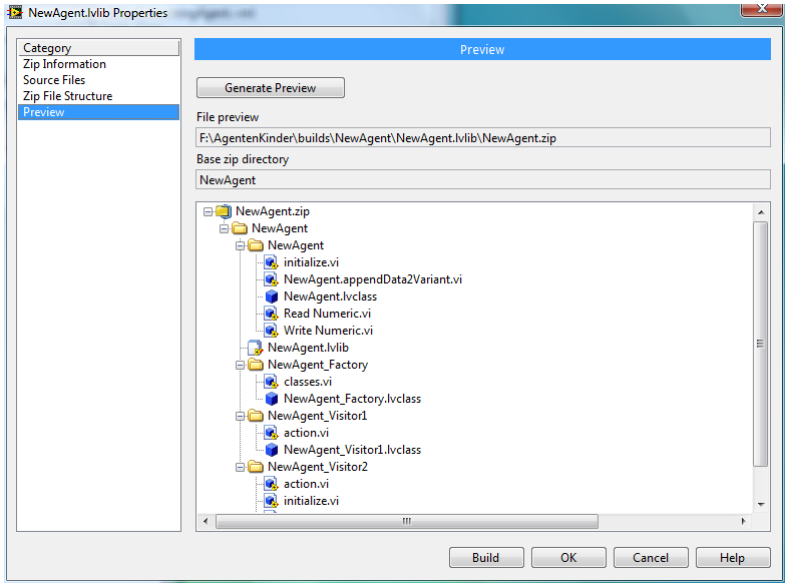

Abbildung 64: Agentenerzeugung 10 - Zip Einstellungen - Preview

Die Vorschau zeigt noch einmal die Ordnerstruktur. Falls alle Einstellungen nun korrekt sind, kann die Zip-Datei erstellt werden.

Um den neu erstellten Agenten auf einem Host verfügbar zu machen, wird die soeben erstellte Zip-Datei im Klassenverzeichnis des Hosts entpackt.

Literaturverzeichnis

195.93.158.26. (22. Februar 2010). *Visitor*. Abgerufen am 09. März 2010 von de.wikipedia.org: http://de.wikipedia.org/w/index.php?title=Visitor&oldid=70997236

70.178.173.90. (2. November 2009). *Thread Pool Pattern*. Abgerufen am 09. März 2010 von en.wikipedia.org: http://en.wikipedia.org/w/index.php?title=Thread_pool_pattern&oldid=323550249

94.245.199.57. (11. März 2010). *Race Condition*. Abgerufen am 1. April 2010 von de.wikipedia.org: http://de.wikipedia.org/w/index.php?title=Race_Condition&oldid=70262967

Ayrton89. (07. Februar 2010). *Fabrikmethode*. Abgerufen am 09. März 2010 von de.wikipedia.org: http://de.wikipedia.org/w/index.php?title=Fabrikmethode&oldid=70352930

Beck, D., Brand, H., & Kurz, N. (2005). Die LabVIEW DIM Schnittstelle. In *Virtuelle Instrumente in der Praxis - Begleitband zum Kongress VIP 2005* (S. 20-26). Heidelberg: Hüthig GmbH & Co. KG.

Brand, D. D. (10. März 2010). *CS Framework*. Abgerufen am 01. April 2010 von http://wiki.gsi.de/cgi-bin/view/CSframework/WebHome

Brand, H. (18. Februar 2005). *LabVIEW Techniken*. Abgerufen am 25. März 2008 von http://www-wnt.gsi.de/lvug/techniken.htm

Brand, H. (31. März 2009). *Wiki.gsi.de.* Abgerufen am 25. 02 2010 von http://wiki.gsi.de/pub/NIUser/HGFBaseClassLibrary/HGF_BaseClasses.ppt

Cern. (21. April 1997). *DIM*. Abgerufen am 18. März 2010 von DIM Informstionsseite: http://dim.web.cern.ch/dim/

Free Software Foundation. (01. Februar 2007). *http://www.gnupg.org/*. Abgerufen am 04. April 2010 von http://www.gnupg.org/

http://www.gsi.de. (09. 12 2009). Abgerufen am 05. 03 2010 von GSI: http://www.gsi.de/portrait/ueberblick.html

Http://www.ni.com. (26. Oktober 2009). *LVOOP Descisions behind the Design*. Abgerufen am 1. April 2010 von http://www.ni.com: http://zone.ni.com/devzone/cda/tut/p/id/3574

Mattern, F. (kein Datum). Abgerufen am 09. März 2010 von Mobile Agenten, Fachbereich Informatik der Technischen Universität Darmstadt: http://www.vs.inf.ethz.ch/publ/papers/mobags.html

Silmaril. (28. März 2010). *LabVIEW*. Abgerufen am 01. April 2010 von de.wikipedia.org: http://de.wikipedia.org/w/index.php?title=LabVIEW&oldid=72457477

www.ni.com. (2. Juli 2008). *National Instruments*. Abgerufen am 2. April 2010 von What is VI Server: http://digital.ni.com/public.nsf/allkb/FBD546A08048EA5D86256E8E0027040B

Abbildungsverzeichnis

UML Diagramme erstellt mit Endevo UML Modeller 1.2:
http://www.flander.com/

Screenshots erstellt mit Free Screen Capturer:
http://free-screen-capturer.softonic.de/

Sonstige Diagramme erstellt mit Dia:
http://projects.gnome.org/dia/