# Diplomarbeit

## Entwicklung eines OPC UA Servers für eine NC-Maschine

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines

Diplom-Ingenieurs unter der Leitung von

**Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Kittl**

(Institut für Fertigungstechnik und Hochleistungslasertechnik)

eingereicht an der Technischen Universität Wien

**Fakultät für Maschinenwesen und Betriebswissenschaften**

von

**Iman Ayatollahi**

9825423 (700)

Mommsengasse 4/10

1040 Wien

Wien, im April 2014

_____

Iman Ayatollahi

Ich habe zur Kenntnis genommen, dass ich zur Drucklegung meiner Arbeit unter der Bezeichnung

# Diplomarbeit

nur mit Bewilligung der Prüfungskommission berechtigt bin.

Ich erkläre weiters Eides statt, dass ich meine Diplomarbeit nach den anerkannten Grundsätzen für wissenschaftliche Abhandlungen selbstständig ausgeführt habe und alle verwendeten Hilfsmittel, insbesondere die zugrunde gelegte Literatur, genannt habe.

Weiters erkläre ich, dass ich dieses Diplomarbeitsthema bisher weder im In- noch Ausland (einer Beurteilerin/einen Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe und dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Wien, im April 2014

_____

Iman Ayatollahi

# Danksagung

Einleitend möchte ich mich herzlichst beim Betreuer meiner Diplomarbeit, Herrn Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Kittl, bedanken. Die ausgezeichnete Betreuung und auch die anregenden Gespräche zu diesem Thema, haben die Begeisterung geweckt, mich weiterhin in diesem spannenden Feld der Fertigungstechnik zu engagieren.

Besonderer Dank gebührt auch meinen Kollegen und Freunden Dipl.-Ing. Florian Pauker und Thomas Weiler für ihre fachliche und moralische Unterstützung während der Entwicklung der Software und der Verfassung dieser Arbeit.

Herrn Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Kastner danke ich herzlichst für die fachlichen Anmerkungen und für das Korrekturlesen der Arbeit.

An dieser Stelle möchte ich auch meinen Eltern und meinem Bruder Mag. Ehsan Ayatollahi für die finanzielle und moralische Unterstützung während meines Studiums danken. Ohne diese Unterstützung hätte ich mein Diplomstudium wahrscheinlich nicht absolvieren können.

# Kurzfassung

Der Einsatz von semantischen Kommunikationstechnologien in der Fertigung würde das Einrichten von „Machine-to-Machine“-Kommunikationen erleichtern und somit auch die Flexibilität und Erweiterbarkeit von Fertigungssystemen steigern. Flexible Fertigungszellen werden heute aber meist als Komplettpaket von Maschinenherstellern oder Systemintegratoren für eine bestimmte Art von Produktion angeboten bzw. parametrisiert und lassen sich nur mit viel Aufwand rekonfigurieren. Die Kommunikation zwischen den Komponenten einer Fertigungszelle basiert heute noch immer auf binären Ein- und Ausgängen von speicherprogrammierbaren Steuerungen (SPS) oder auf (proprietären) Bussystemen, und wird von der SPS einer Maschine oder von einem Kommunikationsmodul eines Leitrechners gesteuert.

„OPC Unified Architecture“ (OPC UA) ist eine vielversprechende standardisierte Spezifikation für die Kommunikation in der modernen Automationstechnik. Sie ist die aktuellste Spezifikation der „OPC Foundation“, und zeichnet sich durch ihre Plattformunabhängigkeit, Skalierbarkeit und Erweiterbarkeit aus. Das Besondere an dieser Kommunikationstechnologie ist, dass der Server Daten und Dienste samt ihrer Bedeutung – also semantisch – präsentiert. Mittlerweile existieren auch einige Software Development Kits (SDKs), die eine bequeme Programmierung von OPC UA Applikationen ermöglichen.

Diese Arbeit behandelt die Entwicklung eines OPC UA Servers für die Drehmaschine „Emco Concept Turn 55“, die als Komponente einer flexiblen Fertigungszelle eingesetzt werden soll. Der Server soll einem Zellrechner (OPC UA Client) Dienste und Informationen der Drehmaschine anbieten, die vor allem für die Orchestrierung des automatisierten Betriebs der Fertigungszelle relevant sind. Um einen automatischen Ablauf in der Zelle zu veranschaulichen, wurde auch ein bestehender OPC UA Client um eine einfache Ablaufsteuerung erweitert. Für diese Entwicklung wurden SDKs von „Unified Automation“ verwendet.

# Abstract

Using semantic communication technologies in production environments would facilitate setting up machine-to-machine communication and would consequently improve the flexibility and expandability of manufacturing systems. Today, flexible manufacturing cells are usually preconfigured, often available commercially off-the-shelf for a specific type of processing. Therefore a lot of effort and specialized knowledge is required for their reconfiguration. Communication between the components of a manufacturing cell is still mostly based on binary input and outputs of a programmable logic controller (PLC) or (proprietary) bus systems. Hence the communication is controlled by the PLC of a machine or by an I/O-module of a central computer supervising the cell.

OPC Unified Architecture (OPC UA) is a promising standardized specification for communication in the field of modern automation technologies. It is the most recent specification by the OPC Foundation, and stands out due to its platform-independence, scalability and extensibility. What makes OPC UA unique as a communication technology is its powerful capability of exposing semantics of provided data. In other words, the server can present data and services highly enriched with their meanings. By now, several software development kits (SDKs) exist, that facilitate easy programming of OPC UA applications.

In this thesis the development of an OPC UA Server for the lathe “Emco Concept Turn 55” is described. The lathe is planned to be used as a component of a flexible manufacturing cell. The server should therefore offer specific services and information to the cell controller, hence enabling the cell controller to orchestrate the automatic operation of tasks on the lathe. An existing OPC UA Client was also extended with a simple sequence control to demonstrate automatic operation of the cell using OPC UA communication. For this development, SDKs provided by “Unified Automation” were used.

# Inhaltsverzeichnis

# 1 Einleitung

Bei flexiblen Fertigungssystemen z.B. bei einer Roboter-Fertigungszelle werden Steuerungsaufgaben wie die Organisation des Materialflusses oder der NC-Programme meist von der Steuerung einer der Komponenten übernommen. Wenn diese Aufgaben von einem zugehörigen Zellrechner durchgeführt werden, kann mehr Flexibilität und ein bequemerer Betrieb erreicht werden [1]. Diese erhöhte Flexibilität kann hinsichtlich des Trends hin zur Variantenfertigung statt der klassischen Serienfertigung zu ökonomischerer Produktion führen. Dies trifft insbesondere für kleine und mittlere Unternehmen (KMU) zu, die dadurch ihre Ressourcen besser auslasten könnten. Die Implementierung einer Zellsteuerung ist jedoch wegen der verschiedenen Kommunikationsprotokolle und Maschinen, die zum Einsatz kommen, mit viel Aufwand verbunden. Um diese Problematik zu lösen, bedarf es einer standardisierten Kommunikationslösung. In dieser Arbeit wird die Entwicklung eines OPC UA Servers zum Zweck der Realisierung einer semantischen Kommunikation in einer Fertigungszelle beschrieben.

## 1.1 Historische Entwicklung und Stand der Technik

In den 1970ern kamen die ersten flexiblen Fertigungssysteme zum Einsatz und zwecks Datenerhebung und Fernsteuerung von Werkzeugmaschinen wurden herstellerspezifisch Kommunikationsprotokolle entwickelt. Anfangs wurden serielle Punkt zu Punkt Kommunikationen realisiert wie beispielsweise die LSV2-Prozedur, die z.B. auch in den Siemens-Steuerungen „System 8“ aus den 1980ern Verwendung fand [2]. Später erschienen dann Ethernet-Schnittstellen und TCP/IP basierte Protokolle darunter auch das MCIS RPC Sinumerik von Siemens, das unter dem Namen „Create MyInterface“ heute noch angeboten wird [3]. Bald gewann das Thema der standardisierten Kommunikation immer mehr an Bedeutung. Ende der 1980ern wurde die „Manufacturing Message Specification“ (MMS) vorgestellt und schließlich als ISO 9506 standardisiert [4]. Als eine Anwendung dieses Standards präsentierten Leitão et al. eine Integrationslösung für zwei CNC Maschinen und einen anthropomorphen Roboter [5]. Dabei wurden alle Systemkomponenten mit MAP/MMS Schnittstellenkarten ausgestattet. Jedoch wurde nur ein kleiner Satz der in ISO 9506 definierten MMS Dienste implementiert und die großen Erwartungen an MMS konnten nicht erfüllt werden.

Für einfache Zellenkonfigurationen wird heute noch immer auf einfachen binären Ein/ und Ausgabesignalen gesetzt. Beispielsweise hat der Verein Deutscher Maschinen- und Anlagenbau (VDMA) im Jahr 2011 eine Spezifikation für die automatische Be- und Entladung von Werkzeugmaschinen veröffentlicht [6]. Diese zielt darauf ab, den Aufwand für die Integration von Robotern und Werkzeugmaschinen in einer Zelle zu

reduzieren, indem man „die Schnittstelle zwischen den beteiligten Maschinen und Steuerungen signaltechnisch und funktional zu vereinheitlichen“ versucht [6]. In Deutschland wird seit 2011 durch die Initiative „Industrie 4.0“ die Realisierung von intelligenten Fabriken durch Verwendung von Konzepte wie „Cyber-Physical-Systems“ und „Internet der Dinge“ vorangetrieben. Diese ist u.a. gekennzeichnet durch flexiblere und wandlungsfähige Produktionssysteme [7]. Voraussetzung hierfür ist die Möglichkeit einer nahtlosen Kommunikation der einzelnen Geräte und Ressourcen sowohl auf Feldebene als auch auf Ebene des unternehmensweiten Managements (ERP-Systeme) und darüber hinaus (Internet). Deshalb bedarf es einer plattformunabhängigen einheitlichen (maschinen-)semantischen Kommunikationstechnologie, wie es schon heute OPC UA ist. In der Realität werden aber noch immer meist nur Treiber für die verschiedensten Systeme angeboten und Systemintegratoren haben wieder mit unzähligen verschiedenen Kommunikationsprotokollen zu kämpfen. Viele Unternehmen im Bereich der Automatisierungstechnik benutzen „moderne“ Begriffe wie „Cyber-Physical-Systems“ und „intelligente Fabrik“, um ihre derzeitigen Entwicklungen zu beschreiben, intern verwenden sie jedoch noch immer etablierte, meist proprietäre, Schnittstellen für die Maschinen-Kommunikation auf Basis von Bussystemen und binären Ein-Ausgangssignale.

OPC Unified Architecture, als ein Kommunikations- und Informationsmodellierungs-Standard könnte die vorhin erwähnte Problematik lösen, da mit ihr Daten und ihre Semantik zusammen objektorientiert modelliert und übertragen werden können. Mit OPC UA können Clients auf Informationen von Servern zugreifen, die wiederum von anderen OPC UA Servern Informationen abfragen können und Dienste (Methoden, Programme) ausführen können. So sind verkettete Server-Client Strukturen möglich, die eine semantische Kommunikation sowohl horizontal als auch vertikal in der Automatisierungspyramide ermöglichen.

## 1.2 Ausgangssituation und Rahmenbedingungen

Am Institut für Fertigungstechnik und Hochleistungslasertechnik (IFT) wird der Einsatz einer semantischen Kommunikation zwischen Komponenten einer flexiblen Fertigungszelle evaluiert. Zu diesem Zweck wurde schon eine Fertigungszelle bestehend aus folgenden Maschinen aufgebaut:

- CNC-Drehmaschine: Emco Concept Turn 55
- CNC-Fräsmaschine: Emco Concept Mill 55
- Industrieroboter: ABB IRB 120

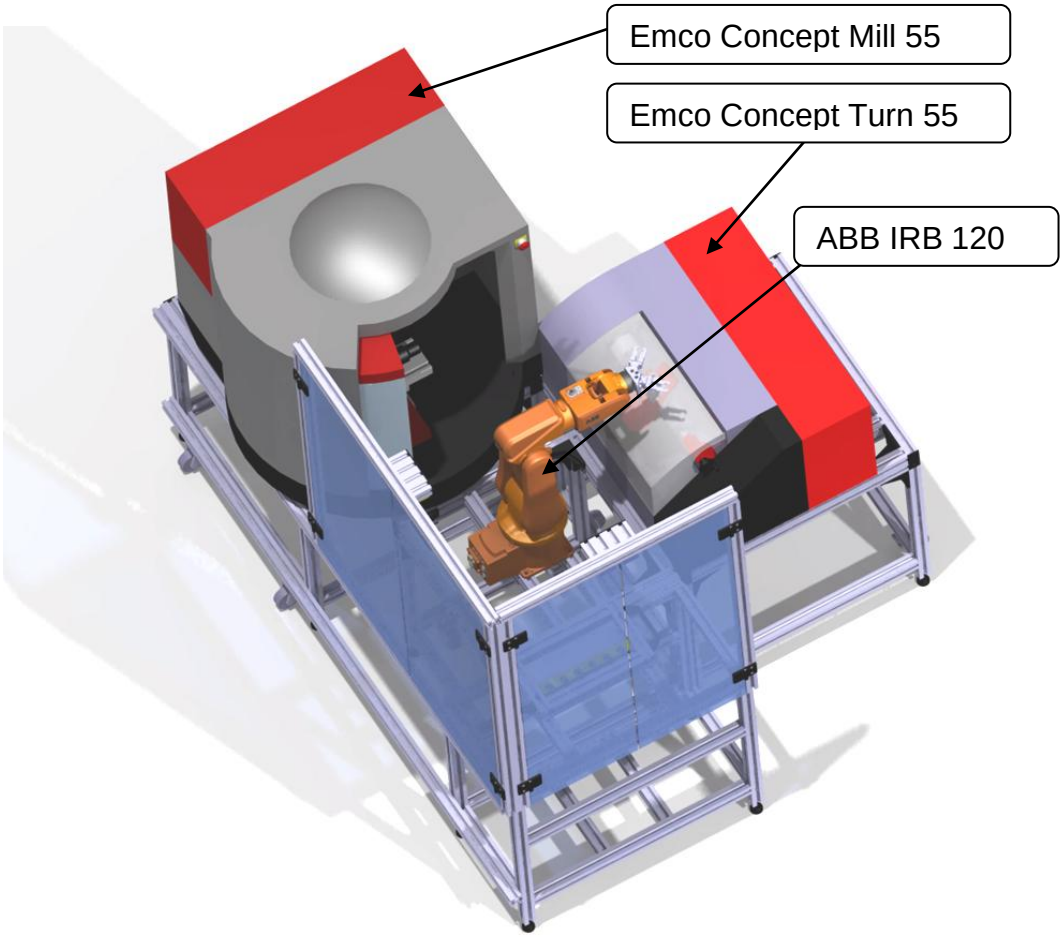


**Abbildung 1.1: 3D-Modell der Fertigungszelle am IFT [8]**

Bei dieser Lösung übernimmt ein Zellrechner aufbauend auf XML-Dateien die Orchestrierung der Abläufe in der Fertigungszelle [8]. Um die Werkzeugmaschinen der Concept-Serie fernzusteuern und aktuelle Maschinendaten zu empfangen, wurde das von Emco entwickelte proprietäre DNC-Protokoll verwendet [9]. Es wurde ein externer Softwareentwickler beauftragt einen Maschinen-Treiber zu entwickeln, um über das DNC-Protokoll, Funktionen und Daten der Emco-Concept-Maschinen für die weitere Programmierung in Microsoft Windows Systemen zur Verfügung zu stellen [10].

Diese Fertigungszelle, die diesbezüglichen Arbeiten am IFT und der schon entwickelte Maschinen-Treiber, sind die Ausgangssituation der in dieser Arbeit dokumentierten Softwareentwicklung. Gefordert wurde ein lauffähiger OPC UA Server für die Drehmaschine „Concept Turn 55“, den man im Rahmen der ViennaTec 2012 am Messestand des IFT dann auch präsentierte. Dieser musste so programmiert werden, dass er zumindest mit entsprechender Parametrisierung auch für die Fräsmaschine „Concept Mill 55“ eingesetzt werden konnte. Um Be- bzw. Entladeoperationen zu veranschaulichen, wurde auch ein OPC UA Client mit der Möglichkeit ausgestattet, eine Sequenz von Methodenaufrufe zu erstellen, die automatisch und sicher abgearbeitet werden kann.

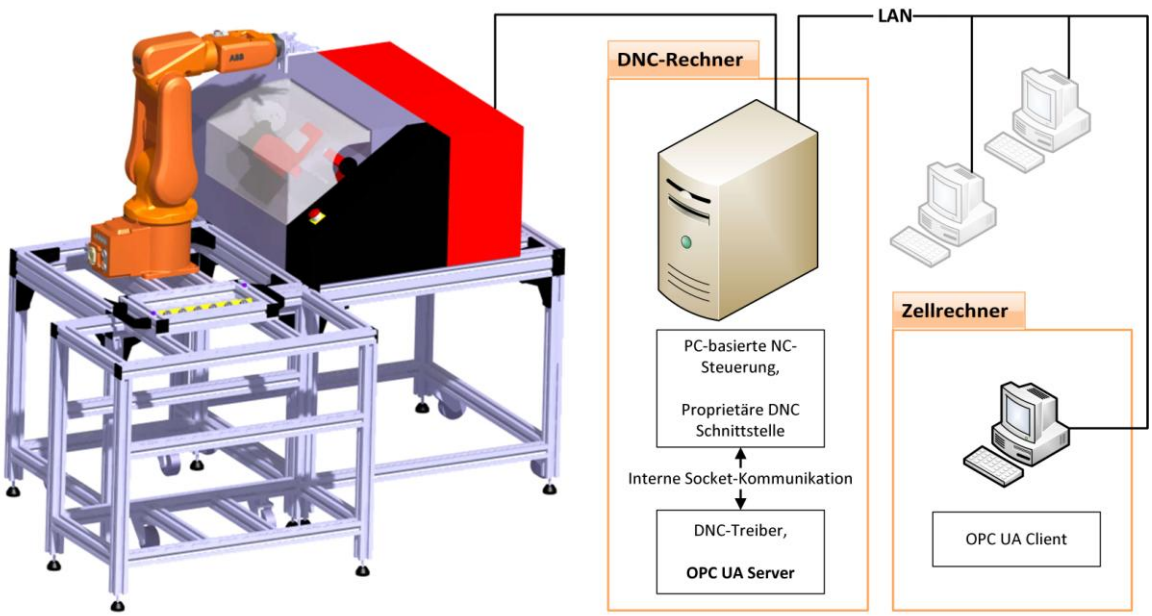


**Abbildung 1.2: Schematische Darstellung der Kommunikation zwischen Zellrechner und Werkzeugmaschine**

Der OPC UA Server und der OPC UA Client sollten auf Rechner im selben LAN laufen, wobei der Server samt DNC-Treiber auch eine Socket-Kommunikation mit dem DNC-Rechner aufbauen muss. Um den OPC UA Server logisch als Teil des Steuerrechners der Werkzeugmaschine darzustellen, wird er auf dem DNC-Rechner ausgeführt. Dadurch ergibt sich eine interne Socket-Kommunikation auf dem DNC-Rechner, wie in Abbildung 1.2 zu sehen. Später in dieser Arbeit wird öfters auf den oben dargestellten DNC-Rechner verwiesen. Der in Abbildung 1.2 dargestellte Zellrechner ist ein beliebiger Rechner, auf dem ein OPC UA Client läuft. Er wurde Zellrechner benannt, um klarzustellen, dass auf ihm die Steuerung der Fertigungszelle, somit die Orchestrierung der Abläufe jeder Komponente der Zelle über OPC UA-Kommunikation geschehen soll.

# 2 OPC Unified Architecture

Seit den 1990er Jahren steigt in der industriellen Automation die Anwendung von auf PCs basierten Automationssystemen kontinuierlich. Microsoft Windows basierte PCs werden seitdem vermehrt für Visualisierung und auch Steuerung von automatisierten Anlagen eingesetzt. Entwickler von Automatisierungssoftware wurden mit dem Problem konfrontiert, dass für Datenzugriffe in der industriellen Automation unzählige (teilweise proprietäre) Bussysteme, Protokolle und Schnittstellen verwendet wurden. Einige HMI und SCADA Softwarehersteller arbeiteten daraufhin zusammen, um eine Norm für Gerätetreiber zu bilden und somit einen standardisierten Zugriff auf Prozess- und Automatisierungsdaten für Windows-basierte PCs anzubieten. Daraus resultierte 1996 die erste OPC Spezifikation, namens „OPC Data Access“ (OPC DA), und eine Organisation namens „OPC Foundation“ wurde gegründet, die seitdem erfolgreich die OPC Spezifikationen weiterentwickelt und den Umfang dieser erweitert hat [11].[1]

Die aktuellste, 2009 veröffentlichte OPC Spezifikation heißt „OPC Unified Architcture“ (OPC UA). Sie ist plattformunabhängig und unterscheidet sich grundlegend von den vorherigen Spezifikationen. Nach der Veröffentlichung von OPC UA werden die Vorgängerspezifikationen meist als „Classic OPC“ bezeichnet.

## 2.1 Classic OPC [11]

OPC wird heute als eine standardisierte Schnittstelle zwischen Automationssystemen in verschiedenen Ebenen der Automatisierungspyramide eingesetzt [11]. Classic OPC umfasst neben der wohl wichtigsten Spezifikation „OPC Data Access“ (DA) auch andere Spezifikationen wie „Alarms and Events“ (A/E), „Historic Data Access“ (HDA) und XML-DA (XML-basierte Datenübertragung). Außer XML-DA basieren alle anderen Classic OPC Spezifikationen auf der von Microsoft für Windows-Betriebssysteme entwickelten „Component Object Model“ (COM) und „Distributed COM“ (DCOM) Technik. Classic OPC wurde mit XML-DA erweitert, um den Einsatz von OPC auf Nicht-Windows Plattformen sowie auf eingebetteten Systemen zu ermöglichen.

[1] Die Abkürzung OPC stand ursprünglich für „OLE (Object Linking and Embedding) for Process Control”, in Anlehnung an den von Microsoft entwickelten Objektsystem und Protokoll zum Einbetten von Objekten aus verschiedenen Anwendungen ineinander. Da die neuesten Spezifikationen plattformunabhängig sind und auch nicht nur in der Prozessleittechnik Verwendung finden, werden oft andere Begriffe verwendet für die OPC stehen könnte. Aktuell verwendet die OPC-Foundation den Slogan „**O**pen **P**roductivity & **C**onnectivity“ für die Abkürzung [12].

## 2.2 Motivation für OPC UA [11]

OPC Unified Architecture entstand aus dem Wunsch heraus, einen echten Ersatz für die existierenden COM-basierten Spezifikationen zu entwickeln, ohne Funktions- und Leistungseinbußen. Zusätzlich muss sie alle Anforderungen für plattformunabhängige Systemschnittstellen mit ausführlichem und erweiterbarem Modellierungsvermögen decken, so dass auch komplexe Systeme beschrieben werden können. Die breiten Anwendungsgebiete, wo OPC Verwendung findet, verlangen auch nach Skalierbarkeit, angefangen von eingebetteten Systemen, über SCADA und DCS bis hin zu MES und ERP-Systemen. Während Classic OPC als eine Gerätetreiber-Schnittstelle ausgelegt war, wird OPC heute als eine Systemschnittstelle verwendet. Deshalb ist die Zuverlässigkeit der Kommunikation zwischen verteilten Systemen sehr wichtig. Robustheit und Fehlertoleranz sind also wichtige Anforderungen an die Kommunikation, aber auch Redundanzen für hohe Verfügbarkeiten.

Bei Classic OPC war die Datenmodellierung sehr beschränkt, deshalb musste OPC so weiterentwickelt werden, dass ein allgemeines objektorientiertes Modell für alle OPC Daten bereitgestellt werden konnte. Dieses Modell sollte auch ein erweiterbares Typensystem einschließen, um Metainformationen anzubieten und komplexe Systeme beschreiben zu können. Die Verfügbarkeit von Methoden - bereitgestellt und klar beschrieben von Servern und aufrufbar von Clients - ist eine mächtige Besonderheit, die benötigt wurde um OPC flexibel und erweiterbar zu machen. Die Weiterentwicklung des Modellierungspotentials war somit eine wichtige Anforderung, genauso wichtig war aber auch die Unterstützung einfacher Modelle mit einfachen Konzepten.

In Tabelle 2.1 sind die Anforderungen an den Nachfolger von Classic OPC gelistet, die bei der Entwicklung von OPC UA realisiert wurden.

**Tabelle 2.1: Anforderungen an OPC UA [11]**

| Kommunikation zwischen verteilten Systemen | Datenmodellierung |
|---|---|
| • Zuverlässigkeit durch<br>  ○ Robustheit und Fehlertoleranz<br>  ○ Redundanz<br>• Plattformunabhängigkeit<br>• Skalierbarkeit<br>• Hohe Leistungsfähigkeit<br>• Internet und Firewalls<br>• Sicherheit und Zugangskontrolle<br>• Interoperabilität | • Allgemeines Modell für alle OPC Daten<br>• Objektorientiert<br>• Erweiterbares Typensystem<br>• Metainformationen<br>• Komplexe Daten und Methoden<br>• Skalierbarkeit von einfachen bis komplexen Modellen<br>• Abstraktes Basismodell<br>• Basis für andere Standard-Datenmodelle |

## 2.3 Übersicht über OPC UA

Die Kommunikation durch OPC UA funktioniert als Client-Server-Konzept in einem Netzwerk. OPC UA Server präsentieren in ihrem Adressraum ein Informationsmodell des konkreten Systems – bieten somit Dienste an – und Clients können auf diese Informationen zugreifen, anders ausgedrückt: Dienste des Servers in Anspruch nehmen. Diese Dienste sind u.a.:

- Abfrage von aktuellen oder historischen (Mess-)Daten,
- Aufruf von Funktionen, aber auch
- automatische Übermittlung von ereignisbasierten Meldungen.

Die übermittelten Daten sind mit vielen Metainformationen versehen. Als Beispiel enthält eine Variable neben dem Wert auch weitere Attribute, wie z.B. Datentyp oder Zeitstempel der Generierung des Werts, und kann auch mit Eigenschaften versehen werden, wie die physikalische Einheit und Mindest- und Höchst-Werte, usw. Der Client erhält somit bei der Abfrage der Variable auch mehr oder weniger die Bedeutung der Variable, vorausgesetzt das Informationsmodell wurde gut genug definiert. Zum Vergleich: Die Abfrage von Daten bei einer SPS ist erst möglich, wenn die Adresse der Variable in einem bestimmten Datenbaustein und auch der Datentyp der Variable vor der Abfrage Client-seitig bekannt sind. So werden bestimmte Bytes oder Bits aus dem Speicherraum übertragen – ohne deren Bedeutung. Dieser einfache Vergleich veranschaulicht die Stärke der Semantik bei OPC UA. Ein OPC UA Client muss im Vorhinein nichts über die Struktur der Informationen auf dem Server wissen, und was sie darstellen.

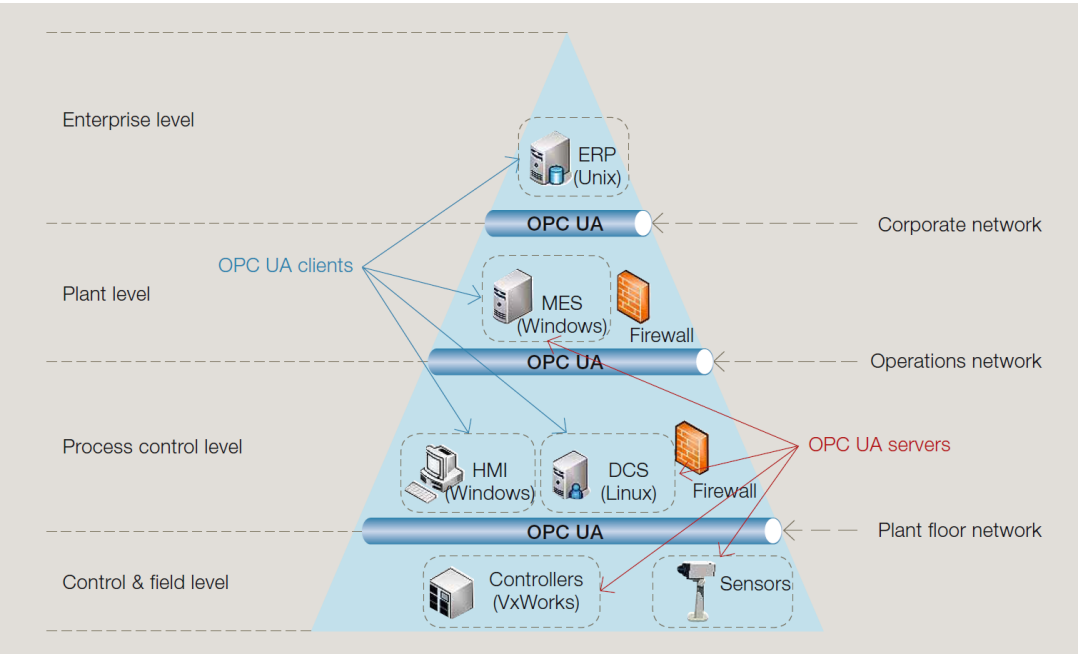


**Abbildung 2.1: Anwendungsgebiete von OPC UA dargestellt in der Automatisierungspyramide [13]**

OPC UA Server können auch als Clients agieren und erlauben somit verkettete Server-Strukturen, die z.B. in der Automatisierungspyramide semantische Daten als OPC UA Objekte von den unteren Ebenen nach oben ohne Mapping oder Übersetzung transportieren könnten (siehe auch Abbildung 2.1).

In Tabelle 2.2 sind die Teilspezifikationen von OPC UA aufgelistet. Die „International Electrotechnical Comission“ (IEC) hat schon die Teilspezifikationen 3 bis 10 in IEC 62541 als Norm aufgenommen, die Teilspezifikationen 1 und 2 sind als „Technical Report (TR)“ aufgelistet und aktuell noch keine IEC-Normen [14]. Das Deutsche Institut für Normung (DIN) hat auch bis jetzt die Teilspezifikationen 2 bis 10 als DIN EN 62541 aufgenommen [15].

**Tabelle 2.2: OPC UA Spezifikationen [11]**

| OPC Unified Automation Specifications |
|---|
| **Core Specification Parts** |
| Part 1 – Concepts |
| Part 2 – Security Model |
| Part 3 – Address Space Model |
| Part 4 – Services |
| Part 5 – Information Model |
| Part 6 – Service Mappings |
| Part 7 – Profiles |
| **Access Type Specification Parts** |
| Part 8 – Data Access |
| Part 9 – Alarms an Conditions |
| Part 10 – Programs |
| Part 11 – Historical Access |
| Part 13 – Aggregates |
| Part 12 – Discovery |

Um ein besseres Verständnis der verschiedenen Schichten von OPC UA zu erlangen, und um die Teilspezifikationen einordnen zu können, findet man in der Literatur oft das in Abbildung 2.2 dargestellte Schichtmodel. In erster Linie soll verdeutlicht werden, dass sowohl die Transportmechanismen als auch das Metamodel die Fundamente der anwendungsnäheren Schichten sind.

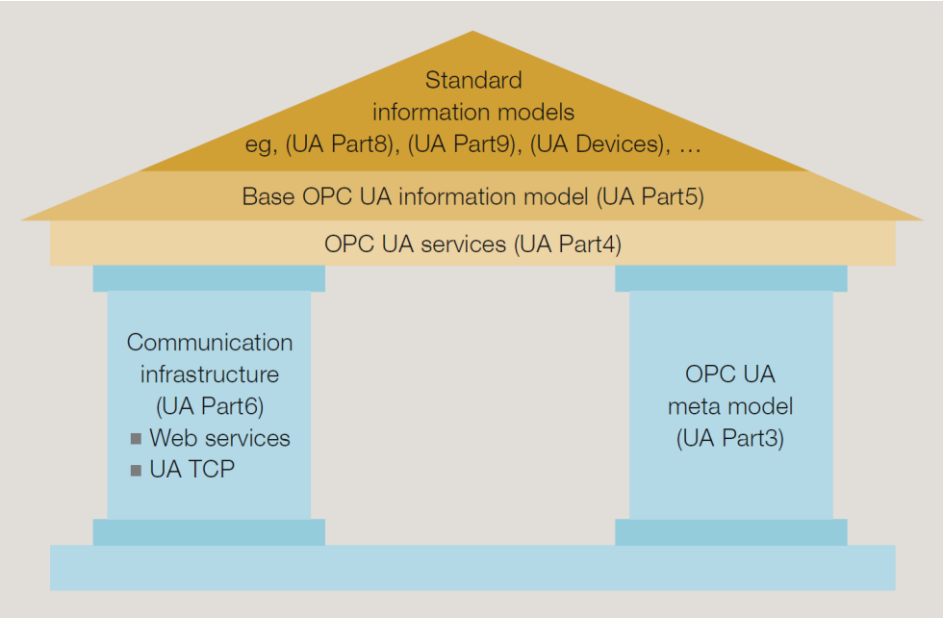


**Abbildung 2.2: Säulen von OPC UA [10] (siehe auch [11])**

Für den Datentransport sind verschiedene Mechanismen für verschiedene Anwendungsgebiete definiert bzw. optimiert worden. Dies ist deshalb notwendig, weil OPC UA für verschiedene Ebenen der Automatisationspyramide konzipiert wurde, die jeweils unterschiedliche Anforderungen an den Datentransport haben (siehe Abbildung 2.1). Beispielsweise sollte die horizontale Kommunikation auf der Feldebene nahezu echtzeitfähig sein, wohingegen auf der Unternehmensebene eine sichere Firewall-kompatible Internet-Kommunikation möglich sein sollte. Derzeit werden folgende Protokolle bzw. Internetstandards für den Datentransport unterstützt:

- Ein für OPC UA optimiertes binäres TCP-Protokoll (am leistungsfähigsten)
- Web Services
- XML
- HTTP (Firewall-freundlich)

Da das Kommunikationsmodel vom eingesetzten Protokoll unabhängig ist, kann OPC UA in Zukunft auch andere Kommunikationsprotokolle integrieren. [11]

Die zweite Säule des OPC UA – das Metamodell – „definiert die Regeln und Grundbausteine, die notwendig sind um ein Informationsmodell auszustellen.“ [11] Diese Regeln sind in der OPC UA Spezifikation „Teil 3 – Adressraummodell“ definiert. [16]. Sowohl Objekte als auch ihre Variablen und Methoden sind Knoten (Nodes) im Adressraum. Jeder Node wird durch Attribute beschrieben, und ihre Beziehung zu anderen Nodes durch Referenzen definiert (Abbildung 2.4).

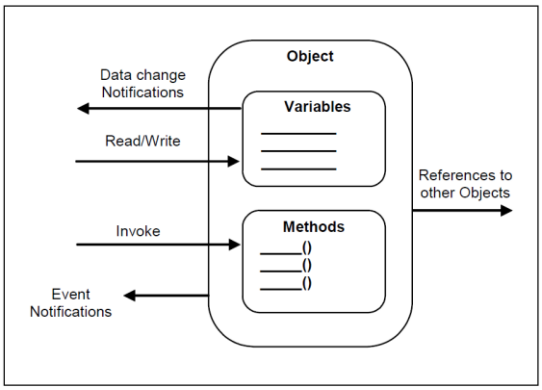


Abbildung 2.3: OPC UA Objektmodell [16]

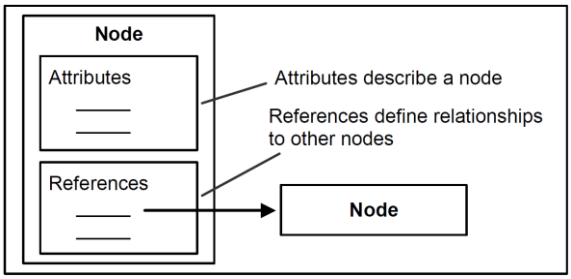


Abbildung 2.4: Node-Modell des Adressraums [16]

Eine Besonderheit von OPC UA ist die Erweiterbarkeit. Die in Abbildung 2.2 dargestellten Säulen des OPC UA, und auch die Services und das Basis-Informationsmodell kann man als die Basis von OPC UA betrachten. Auf diese Basis können auch Spezifikationen für weitere Informationsmodelle von anderen Organisationen aufgebaut werden. Diese werden auch als „Copmpanion-Standards“ definiert und erweitern somit OPC UA mit Informationsmodelle für die Verwendung in einer bestimmten Domäne. Neben „PLCopen“, das mit der OPC Foundation schon einen Companion-Standard entwickelt hat, arbeitet MTConnect zurzeit auch an der Entwicklung einer Spezifikation für OPC UA. Auf der obersten Schicht können dann Geräte-, Maschinen- oder Anlagenhersteller auch Informationsmodelle definieren, die sowohl auf die Spezifikationen ihrer Domäne (spezifiziert von Organisationen) als auch direkt auf die OPC UA Basis zurückgreifen können, siehe Abbildung 2.5.

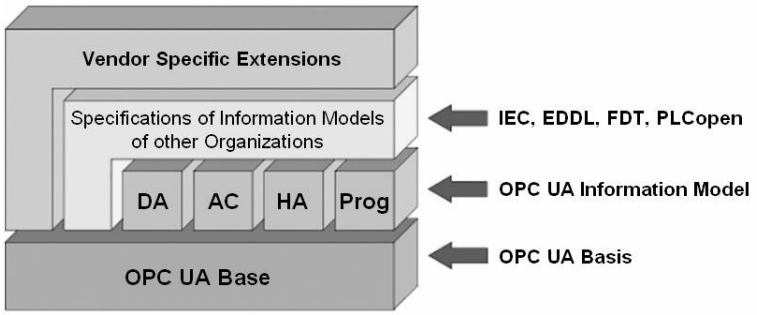


Abbildung 2.5: Schichten der OPC UA Architektur [11]

In Abbildung 2.6 sind die in der OPC UA Basis integrierten Datentypen dargestellt. Man beachte, dass abstrakte Datentypen auch definiert sind, von denen sich andere Typen im Sinne der Objektorientiertheit ableiten.

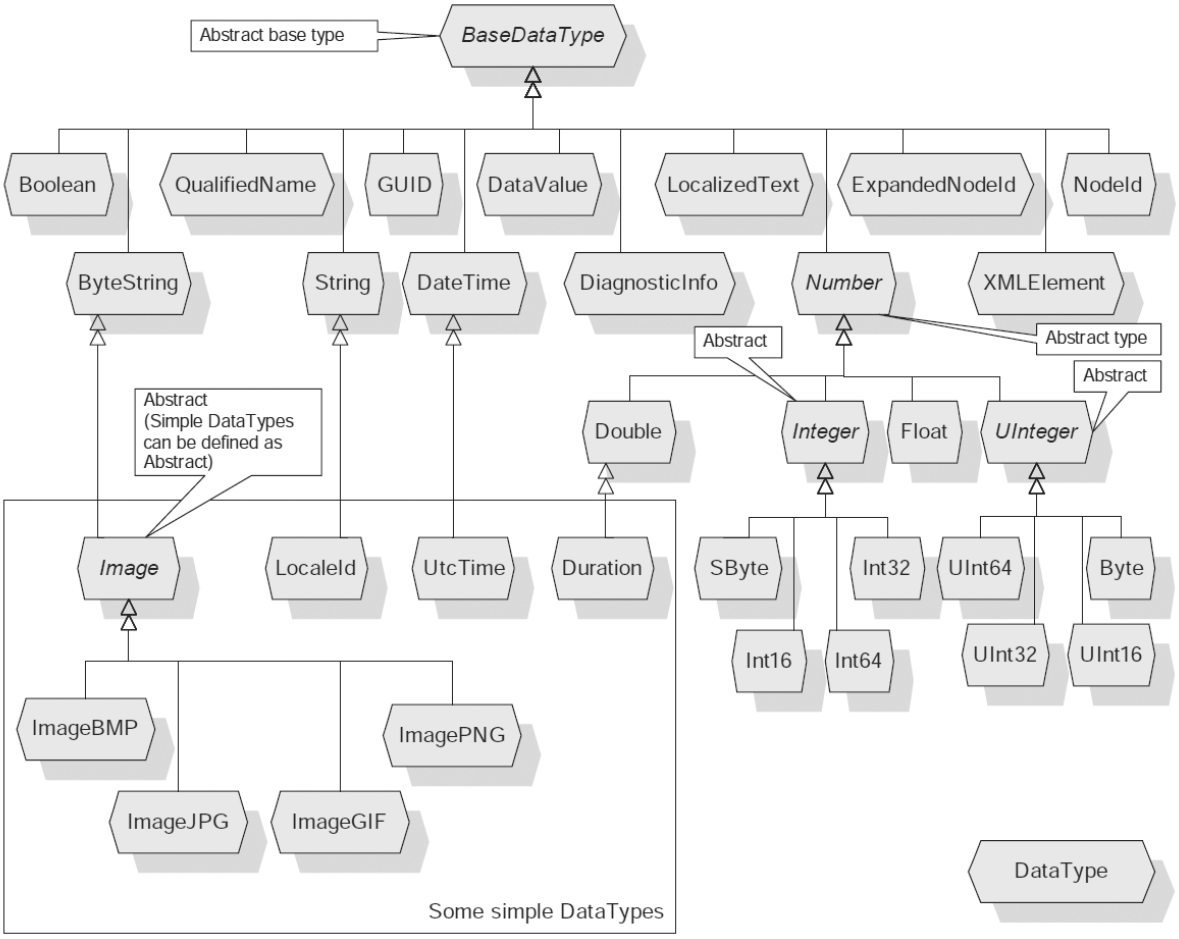


**Abbildung 2.6: Hierarchie der integrierten und simplen Datentypen [11]**

# 3 Programmierung des OPC UA Servers

Es existieren einige „Software Developement Kits“ (SDKs) die den Programmieraufwand für die Entwicklung von OPC UA Anwendungen erheblich erleichtern. Für die Programmierung des Servers wurde aus folgenden Gründen das C++ SDK/Toolkit (Version 1.3.2) von Unified Automation verwendet:

- vollständiger Funktionsumfang der OPC UA Spezifikation verfügbar
- Demoversion kostenlos online beziehbar
- C++ basiert
- Dokumentation der Klassen des SDKs und einige Beispiele online vorhanden [17]
- Informationsmodell kann schnell mit inkludiertem Modellierungswerkzeug „UaModeler“ erstellt, und das Codegerüst sauber generiert werden.
- SDK ist plattformunabhängig und laut Unified Automation auf x86(32- und 64bit) und einigen ARM Prozessoren erfolgreich getestet

## 3.1 Informationsmodell

Ein Informationsmodell ist eine abstrakte Abbildung von Objekten, die Eigenschaften oder Variablen besitzen und Beziehungen (Referenzen) zu anderen Objekten haben können, aber auch Aktionen (Methoden) zum Ausführen beinhalten. Für die Darstellung der hierarchischen Beziehungen von Objekten benutzt man Baumstrukturen. Die vorliegende Werkzeugmaschine (WZM) ist hier das Hauptobjekt, darunter werden Subsysteme der WZM abgebildet, wie in Abbildung 3.1 zu sehen. Es hängt vom Zweck des Informationsmodells und von der Verfügbarkeit von Informationen ab, welche Informationen tatsächlich durch diese Objekte abgebildet werden sollten. Der Zweck der Informationsmodellierung ist hier die Möglichkeit der semantischen Kommunikation zwischen der WZM und einer Automatisierungskomponente (Handhabungsgerät, Roboter,…) oder einer übergeordneten Zellensteuerung. Sowohl relevanten Funktionen (Methoden), die von der WZM beherrscht werden, als auch Variablen, die relevante Zustände der WZM beschreiben, sollen für einen Kommunikationspartner abgebildet werden. Später in Abbildung 3.17 ist der erstellte Adressraum der Maschine mit der Struktur des Informationsmodells während des Serverbetriebs in Detail dargestellt.

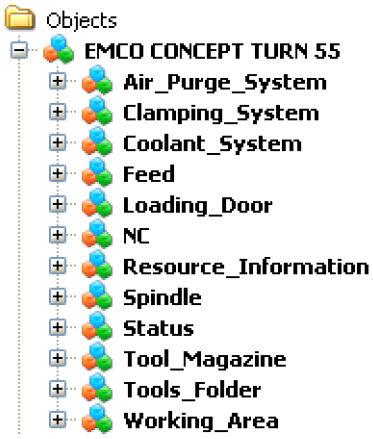


Abbildung 3.1: Das Maschinenobjekt und Unterobjekte im "Objects" Ordner

Gemäß Paradigma der objektorientierten Programmierung werden mit Hilfe des Modellierungswerkzeugs UaModeler ausgehend von der Basisklasse „BaseObjectType“ des „OPC UA Base“ (siehe auch Abbildung 2.5) neue Objekttypen definiert, auf die in diesem Kapitel genau eingegangen wird. Die nachfolgenden Abbildungen sind Screenshots von UaModeler und zeigen die erstellten Objekttypen mit ihren Variablen und Methoden, dargestellt durch Symbole. (siehe Abbildung 3.2)

| | |
|---|---|
| | Objekttyp |
| | Objekt |
| | Variable |
| | Methode |

Abbildung 3.2: Symbolerklärung der UaModeler Screenshots

Beim Erstellen des Informationsmodells in UaModeler müssen auch neu hinzugefügte Knoten (Objekttypen, Variable, Methoden) einem Namensraum zugeordnet werden, vorzugsweise in Form einer URI („Uniform Resource Identifier“), die die Namensvergabestelle der neuen Knoten identifiziert (siehe Kap. 2.8.5 in [11]). Zusätzlich zu den neu definierten Knoten enthält der Adressraum jedes OPC UA Servers natürlich auch Knoten vom OPC UA Base, z.B.: das Serverobjekt samt seinen Eigenschaften und Unterobjekte, Basisdatentypen, usw. Diese Knoten gehören zum Namensraum von OPC UA mit dem URI „http://opcfoundation.org/UA“. (siehe auch Abbildung 2.5)

Ein OPC UA Client der sich mit einem OPC UA Server verbindet, hat im Allgemeinen auch Lesezugriff auf alle Objekttypen, Datentypen und Referenztypen, sowohl jene, die von UA Basis stammen, als auch die neu definierten Objekttypen. Dies ist eine der Stärken von OPC UA; Mahnke et al. schreiben in [11]: „This basic concept of OPC UA enables an OPC UA Client to access the smallest piece of data without the need to understand the whole model exposed by complex systems.”

### 3.1.1 Clamping-, AirPurge-, CoolantSystemType und LoadingDoorType

Diese Objekttypen bilden Hilfsantriebe der Werkzeugmaschine ab. In den einfachsten automatisierten Ausführungen, sollten Objekte dieser Typen ihren aktuellen Zustand im Informationsmodell präsentieren und die Aktionen Ein- und Ausschalten bzw. Öffnen und Schließen zum Ausführen bereitstellen. Aus dieser Überlegung heraus wurden deshalb für jeden dieser Objekttypen eine OPC UA Variable definiert, die den Zustand des Objekts beschreiben soll und zwei OPC UA Methoden ohne Eingabeparameter, um den Zustand verändern zu können. Die Zustandsvariablen (`Clamp_Status`, `Door_Status`, `Purge_Status` und `Coolant_Status`) sind vom Typ String.

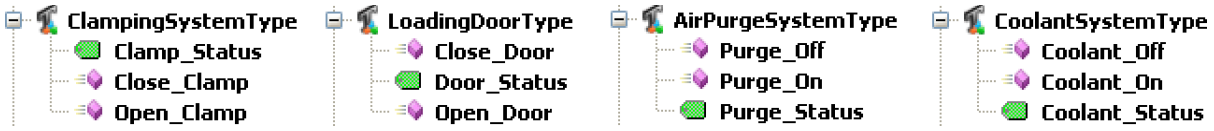


Abbildung 3.3: Objekttypen der Hilfsantriebe

`ClampingSystemType` repräsentiert die Spannvorrichtung, `LoadingDoorType` die Arbeitsraumtür, `AirPurgeSystemType` die Ausblaseinrichtung und `CoolantSystemType` das Kühlschmierstoffsystem.

Wann und wie sich der Wert der Zustandsvariablen ändert und was der Server bei Aufruf der Methoden ausführt, werden nach der Modellierung programmiert. (siehe Kap. 3.4)

Die Status-Variablen können folgende Werte annehmen: „OPEN“, „CLOSED“ und „UNDEFINED“ bzw. „ON“ und „OFF“.

### 3.1.2 FeedType und SpindleType

Die Vorschubachsen der WZM werden durch `FeedType` und der Hauptspindel durch `SpindleType` abgebildet. Informationen zu diesen Systemen der WZM stellt die DNC Schnittstelle jedoch nur begrenzt zur Verfügung. Für den Zweck der automatisierten Beschickung der WZM wären weitere Informationen auch nicht notwendig. Aus diesen Gründen, sind die hier definierten Objekttypen einfach gehalten.


Abbildung 3.4: Objekttypen FeedType und SpindleType

Wie üblich bei Steuerungen von WZM, verändern Override-Werte die durch die NC-Befehle eingestellten Vorschubgeschwindigkeiten bzw. Spindeldrehzahlen und können zwischen 0% und 120% liegen. Die OPC UA Variablen `Feed_Override`, `Spindle_Override` präsentieren diese Prozentwerte als Zahlen zwischen 0 und 120, für den Datentypen der Werte wurde Byte gewählt (siehe auch Abbildung 2.6). Neben dem Wert hat ein OPC UA Variable noch weitere Attribute. Einer dieser Attribute ist die Beschreibung („Description“) mit LocalizedText als Datentyp. Beim Modellieren wird die Beschreibung auch gesetzt, um den Umstand, dass es sich um Prozentwerte handelt, klarzustellen.

Die Methoden `Set_Feed_Override` und `Set_Spindle_Override` sind zum Setzten der Override-Werte modelliert und akzeptieren jeweils eine Zahl vom Datentyp Byte als Eingabeparameter, durch dessen Attribut „Description“ die mögliche Eingabe erklärt wird.

Die Methode `Feed_Position_Request` und die Variable `Spindle_Active_Tool` wurden im Informationsmodell implementiert, jedoch später nicht mit einer Funktionalität versehen, weil die benötigten Informationen hierfür durch den extern entwickelten DNC-Treiber (noch) nicht zugreifbar sind.

### 3.1.3 NCType

Zu diesem Objekttyp gehören zwei Variablen vom Datentyp String. `NC_Program` soll den Programmnamen des aktuell gewählten NC-Programms darstellen, und `NC_Programm_Status` kann die Werte „ACTIVE“, „RESET“ oder „STOP“ annehmen. Die Methoden `Reset_NC`, `Start_NC` und `Stop_NC` sind selbsterklärend, sie dienen zum Starten, Stoppen und Rücksetzen des angewählten NC-Programms. Sie benötigen keine Eingabeparameter und geben auch keine Werte aus. Mit der Methode `Assign_NC_Programm` soll ein auf dem DNC-Rechner vorhandenes NC-Programm gewählt werden können. Mit der Methode `Receive_NC_Program` kann ein vorhandenes NC-Programm als String gelesen und auf dem Client-Rechner gespeichert werden. Und `Transmit_NC_Program` ist die Methode, die ein NC-Programm vom Client-Rechner auf den DNC-Rechner überträgt. Die genaue Programmierung dieser Methoden wird später in dieser Arbeit behandelt.

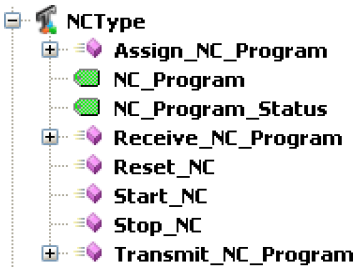


Abbildung 3.5: Objekttyp NCType

### 3.1.4 ResourceInformationType

Dieser Typ wurde definiert, um einige Informationen der Maschine im Informationsmodell verfügbar zu machen. Die Werte der Variablen sind statisch, d.h., sie verändern sich nicht im Betrieb. Obwohl keine Methoden im RessourceInformationType vorhanden sind, wurde dieser als Objekttyp und nicht als Folder-Typ definiert, weil UaModeler die Erstellung von Folder-Typen (noch) nicht unterstützt.

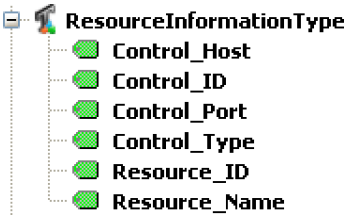


Abbildung 3.6: Objekttyp ResourceInformationType

### 3.1.5 StatusType

`StatusType` beinhaltet auch nur OPC UA Variablen als Strings und ein dynamisch generierbares Objekt namens `Message_`. Der Zustand vom Not-Aus Schalter wird in `Emergency_Stop` dargestellt. `Link_Mode` gibt an, ob die Socket-Verbindung zwischen DNC-Treiber und der Steuerung hergestellt ist. Der Wert von `Operating_Mode` kann „automatic“ oder „manual“ sein. `Machine_Status` sollte folgende Werte annehmen können:

- „ACTIVE“: Maschine führt ein NC-Programm aus, ist also produktiv
- „INACTIVE!“: Maschine führt kein NC-Programm aus und es wurde kein Alarm ausgelöst, die Maschine ist bereit zum Ausführen eines Programms, jedoch unproduktiv.
- „FAULT!“: Eine Unterbrechungsmeldung steht an, ein manuelles Eingreifen ist also notwendig.

`Machine_Status` erlaubt es also, von einem Client in der Zell- oder Leitebene die unproduktive Maschine schnell auszumachen und notwendige Aktionen zu veranlassen.

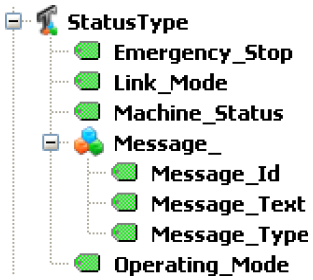


Abbildung 3.7: Objekttyp StatusType

Das Objekt `Message_` wird dynamisch vom Typ `MessageType` instanziiert, und hierarchisch unter der Instanz von StatusType angeordnet, wobei nach dem Unterstrich im Objektnamen eine Nummerierung folgt. Der Grund für diese Modellierung ist, dass man die Möglichkeit hat mehrere Meldungen von der Maschine im Informationsmodel darzustellen. Nach Quittierung einer Meldung sollte das entsprechende nummerierte Objekt automatisch gelöscht werden. „Emco Concept 55“-WZM geben über der DNC-Schnittstelle jedoch maximal nur eine Meldung aus.

### 3.1.6 ToolsFolderType und ToolType

Die Methoden `Receive_Tool_Data` und `Transfer_Tool_Data`, laden bzw. übertragen Werkzeugdaten von der Steuerung bzw. zur Steuerung. `Receive_Tool_Data` hat keine Input- und Output-Argumente, liest alle vorhandenen Werkzeugdaten von der Steuerung, erzeugt für jedes vorhandene Werkzeug ein OPC UA Objekt vom Typ `ToolType` (siehe Abbildung 3.9) mit dem Namen `Tool_XX` (`XX` ist die Werkzeugnummer), und schreibt alle Werte des Werkzeugs in die passenden Variablen des Tool-Objekts. `Receive_Tool_Data` wird beim initialisieren des Servers einmal ausgeführt, um beim Starten des Servers vorhandene Werkzeuge und Werkzeugdaten darzustellen.


Abbildung 3.8: Objekttyp ToolsFolderType

Mit der Methode `Transfer_Tool_Data` können vom OPC UA Client für ein Werkzeug neue Werkzeugdaten in die Steuerung übermittelt werden. Diese Methode hat 26 Eingabeparameter vom Typ Float, die den Werkzeugparametern der Steuerung Sinumerik 840d entsprechen, und zusätzlich noch ein Eingabeparameter

vom Typ UInt16 (unsigned Int16), der die Nummer des Werkzeugs angibt, dessen Werte überschrieben werden sollen. Die Methode `Reset_Tool_Data` wurde vollständigkeitshalber modelliert, jedoch später auf ihrer Programmierung verzichtet.


Abbildung 3.9: Objekttyp ToolType

### 3.1.7 ToolMagazineType

Die Variable `Magazine_Capacity`, gibt die Anzahl der Plätze im Werkzeugmagazin an. Je nachdem für welche WZM der „Emco Concept“-Familie der Server initialisiert wird, kann die Anzahl der Magazinplätze variieren. Dieser Wert bleibt nach der Initialisierung des OPC UA Servers konstant. Beim Initialisieren des Servers sollen auch mehrere Variablen `Tool_in_Location_XX` anhand der Anzahl der Magazinplätze erstellt und nummeriert werden. Diese Variablen sind vom Typ String und beinhalten die Namen des entsprechenden Werkzeugobjektes laut `ToolsFolder` (Instanz des `ToolFolderType`) bzw. „EMPTY“ falls der Magazinplatz unbesetzt ist.

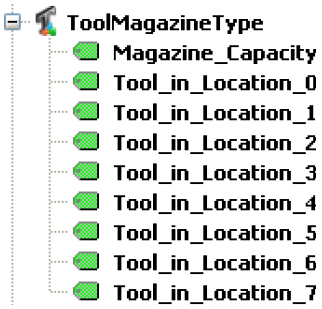


Abbildung 3.10: Objekttyp ToolMagazineType

Wegen der Einfachheit der Werkzeugverwaltung bei „Emco Concept“ WZM ist die Werkzeugnummer identisch mit der Nummer des Magazinplatzes, dem ein Werkzeug zugeordnet ist. D.h., der Wert von `Tool_in_Location_2` ist „Tool_02“ falls Magazinplatz besetzt ist, sonst „EMPTY“.

### 3.1.8 WorkingAreaType

`Loading_Unloading_Finished` und `Notify_Loading_Unloading` sind einfache Methoden ohne Eingabeparameter und die Variablen `Transport_Notification` und `Working_Area_Clear` sind vom Typ Boolean.

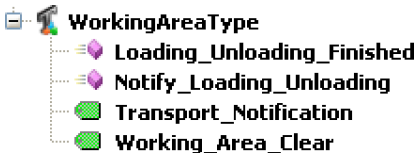


**Abbildung 3.11: Objekttyp WorkingAreaType**

Der Objekttyp `WorkingAreaType` wurde erstellt, um das Zusammenspielen eines Roboters mit der Werkzeugmaschine in der Fertigungszelle zu steuern. Um die Notwendigkeit der Variablen und Methoden dieses Objekttyps zu verstehen, muss man sich eine Be- bzw. Entladeoperation der Maschine durch einen Roboter vorstellen. Der Client des Zellrechners führt die Methode `Notify_Loading_Unloading` auf dem Server der WZM aus. So wird eine Be- bzw. Entladeoperation an der Maschine angemeldet. Der Server der WZM setzt die Variable `Transport_Notification` dann auf TRUE. Sind die Bedingungen für das Beladen bzw. Entladen der WZM gegeben (z.B. Maschinentür ist offen, …), wird die Variable `Working_Area_Clear` auf TRUE gesetzt und alle Methoden der WZM, die eine Bewegung seitens der Maschine im Arbeitsraum verursachen könnten, werden blockiert, außer `Close_Clamp` und `Open_Clamp`. Ist die Be- bzw. Entladeoperation erledigt und hat der Roboter sich schon aus dem Arbeitsraum der WZM zurückgezogen, ruft der Client des Zellrechners auf dem Server der WZM die Methode `Loading_Unloading_Finished` aus. Dadurch werden die Variablen `Transport_Notification` und `Working_Area_Clear` wieder auf FALSE gesetzt.

Weil diese Entwicklung nur den OPC UA Server der WZM behandelt, wurde nach der Modellierung auf die Programmierung der Methoden und Variablen dieses Objekttyps verzichtet. Zur Zeit der Verfassung dieser Arbeit sind für den Roboter und Zellrechner der vorhandenen Fertigungszelle noch keine OPC UA Applikationen entwickelt worden, und dieses Objekt hätte deshalb auch nicht getestet werden können.

### 3.1.9 MachineType

Schließlich wird die WZM selbst durch ein Objekttyp modelliert. Dieser enthält je eine Instanz von den oben definierten Objekttypen, außer von `MessageType` und `ToolType`, die ja während der Laufzeit dynamisch unter den Objekten `Tools_Folder` (Instanz von `ToolsFolderType`) und `Status` (Instanz von `StatusType`) instanziiert werden. Beim Modellieren des `MachineType` im UaModeler wird für den Namen der Instanzen anders als bei den Typen nicht mehr das sogenannte „UpperCamelCase“ verwendet, sondern Unterstriche zwischen den Worten eingesetzt, um die Unterscheidung zwischen Objekt und Objekttyp besser erkennen zu können (`Loading_Door` ist z.B. eine Instanz von `LoadingDoorType`). Wenn der Server in Betrieb ist, kann von einem Client aus auch die definierten Objekttypen durchsucht werden. Neben den neu definierten Objekttypen (hervorgehoben) sind auch die vom OPC UA Base in Abbildung 3.12 unter `BaseObjectType` zu sehen.

Für einen komplexeren Anwendungsfall sollte man (evtl. auch abstrakte) Objekttypen definieren und von diesen die neuen Objekttypen ableiten. So könnten z.B. die sehr ähnlich modellierten Objekttypen der Hilfsantriebe sich von einem abstrakten Objekttyp ableiten. Ein anderes Szenario wäre z.B. folgendes: Ein Hersteller von Subsystemen einer Maschine könnte entsprechend der Typisierung seiner Geräte eine hierarchische Struktur von OPC UA Objekttypen definieren und je nachdem in welche Maschine bestimmte Subsysteme physisch eingebaut sind, müssten dann beim Initialisieren des OPC UA Servers unter dem Objekttyp der Maschine nur die entsprechenden Subsysteme instanziiert werden.

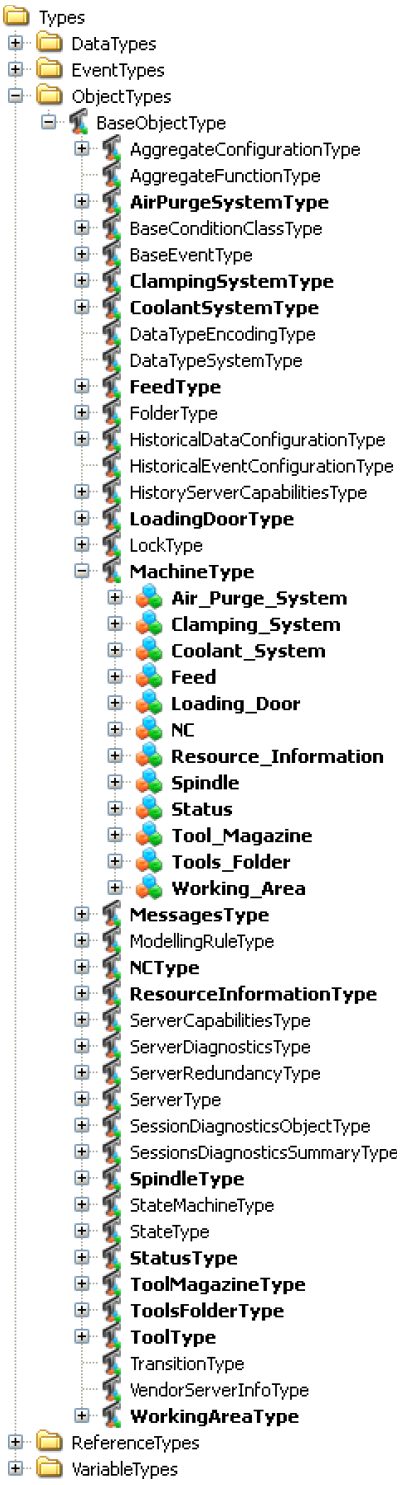


**Abbildung 3.12: MachineType neben anderen Objekttypen im Folder ObjectTypes**

## 3.2 Implementierung der Objekttypen in die Programmierumgebung

Nachdem die Modellierung in UaModeler abgeschlossen ist, erstellt UaModeler für die ausgewählte Programmiersprache (hier C++) folgende Dateien, mit Code und Präprozessor-Anweisungen:

- Für jeden definierten Objekttypen zwei Header- und zwei Quelldateien
- Eine Headerdatei mit dem Namen „*datatypes.h“, enthält u.a. die eventuell definierten Enumerations, in diesem Fall wurden keine verwendet
- Eine Headerdatei mit dem Namen „*identifiers.h“, enthält nur Präprozessor-Anweisungen, die den Bezeichnern der erstellten Nodes (Knoten) des Informationsmodells je eine nummerische NodeID zuweist.
- Eine Header- und Quelldatei mit dem Namen „*nodemanager.h“ bzw. „*nodemanager.cpp“

Zu den Dateien vom ersten Punkt: Jeder Objekttyp entspricht einer Klasse in C++. UaModeler erstellt für jeden Objekttyp eine Basisklasse und eine weitere Klasse, die die Basisklasse erbt, z.B.: `NCTypeBase` und `NCType`. Und für beide Klassen werden jeweils eine Header- und eine Quelldatei erstellt. Der Code für Methoden wird in der Quelldatei der jeweiligen Klasse (z.B.: „*nctype.cpp“) geschrieben, wobei Definitionen der Methoden mit den Input- und Outputargumenten schon generiert sind. Der Platzhalter (*) in den Dateinamen entspricht dem im UaModeler eingegebene Namensraum des Modells.

Für die Programmierung wurde die Programmierumgebung Microsoft Visual Studio gewählt. Die von UaModeler generierten Dateien werden in einer neuen Konsolenanwendung geladen. Zusätzlich müssen noch einige Dateien vom SDK in das Projekt geladen werden. Diese sind:

- opcserver.h und opcserver.cpp
- serverconfigxml.h und serverconfigxml.cpp
- shutdown.h und shutdown.cpp
- servermain.cpp

servermain.cpp wird vom „Hello World“-Beispiel des SDKs für die weitere Programmierung kopiert, nicht benötigte Codefragmente werden gelöscht. Diese Datei enthält die „main“- Funktion. Zudem müssen im Visual Studio Projekt Include-Verzeichnisse des SDKs, und die benötigten Bibliotheken des DNC-Treibers angegeben werden. Ab diesem Zeitpunkt kann das Projekt kompiliert und der Server gestartet werden, jedoch hat der Server noch keine Funktionalität, kann aber einem verbundenen Client, die mit UaModeler modellierten Objekttypen präsentieren.

## 3.3 Aufbau von servermain.cpp

Zunächst wird hier die Struktur der Startfunktion - in der Programmiersprache C++ die „main“-Funktion - behandelt. Daneben beinhaltet servermain.cpp folgende größere Funktionen bzw. Prozeduren, die in den Unterkapiteln näher betrachtet werden:

- MainInit
- MainCycle
- ShutdownSequence

In den Beispielen vom SDK wird der Serverhauptzyklus durch eine Schleife mit einer SDK-spezifischen, thread-sicheren, statischen Sleep-Funktion (member der Klasse UaThread [17]) realisiert. Bei dieser Entwicklung wurde jedoch eine event-basierte Lösung implementiert, indem ein .NET Timer (Zeitgeber) aus dem Namensraum „System.Timers“ verwendet wurde. Abbildung 3.13 stellt den Ablaufplan der „main“-Funktion von servermain.cpp dar. Auf der rechten Seite sind die Schritte nummeriert, wobei die Schritte 9 bis 11 wegen des zuvor erwähnten Timers zeitlich unabhängig vom Haupt-Thread laufen, und deshalb als ein paralleler Strang im Ablaufplan dargestellt werden.

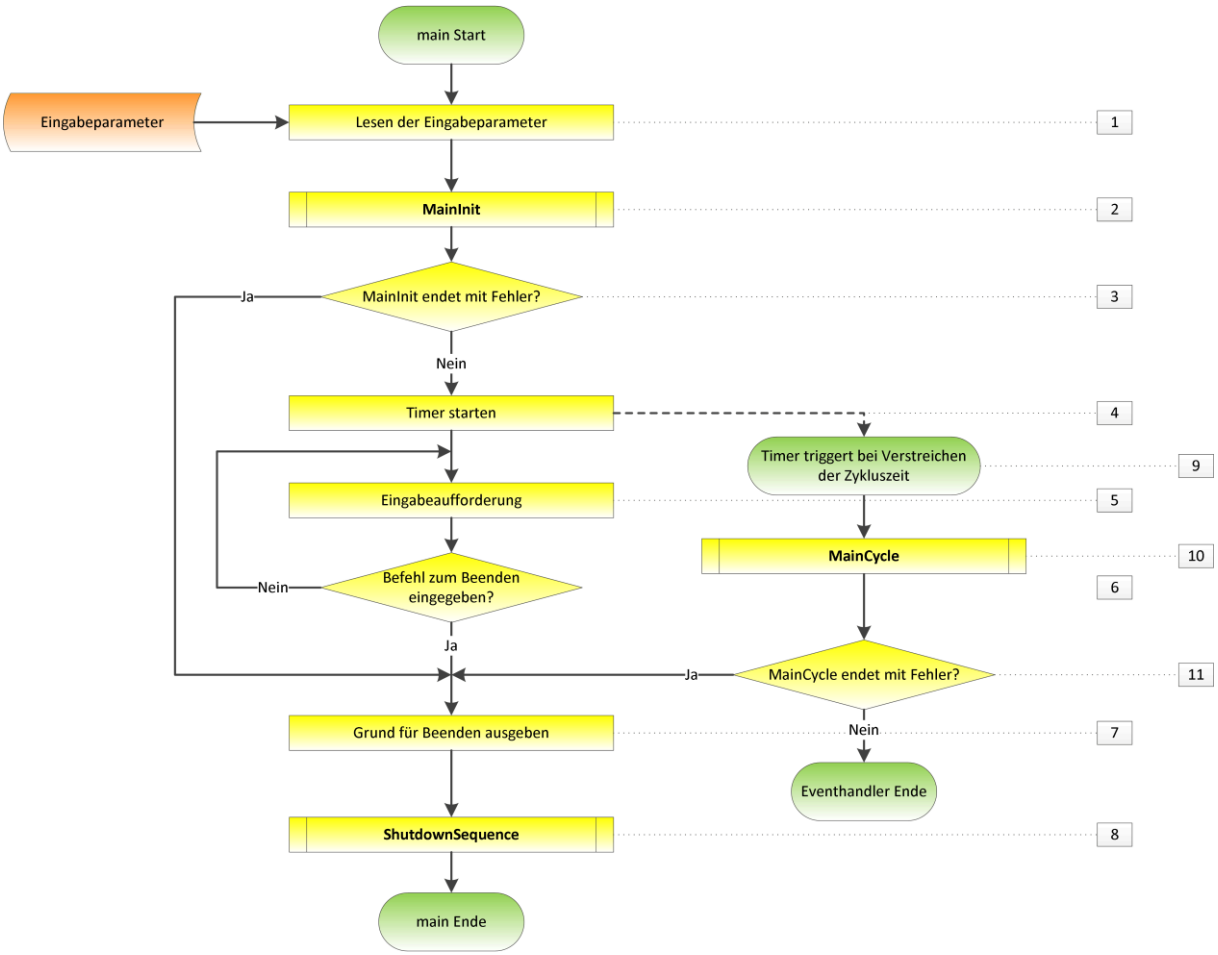


Abbildung 3.13: Ablaufplan der "main"-Funktion

Die „main“-Funktion ruft zuerst eine Initialisierungsfunktion namens MainInit auf. MainInit initialisiert u.a. den Timer mit der eingegebenen Zykluszeit. Wenn MainInit keinen Fehler ausgibt, wird der Timer gestartet (Schritt 4 in Abbildung 3.13).

Solange der Timer nicht deaktiviert wird, wird jedes Mal nach Verstreichen der eingestellten Zeit MainCycle aufgerufen. Der Hauptzyklus des Servers läuft somit nicht wie in den Programmbeispielen des SDKs in einer Schleife ab. Der sogenannte serverbasierte Timer wird als static-member einer eigens dafür erstellten Klasse deklariert. Diese Lösung gilt gemäß Microsoft Developer Network als threadsicher, siehe [18] .

Der Haupt-Thread verweilt im normalen Betrieb bei der Eingabeaufforderung (Schritt 5) bis der Benutzer an der Konsole „exit“ eingibt. Mit Eingabeparameter (Schritt 1) sind die Optionen gemeint, die man beim Ausführen der exe-Datei in der Kommandozeile (oder in einer Verknüpfungsdatei) anhängt. Ohne Eingabeparameter startet der Server mit voreingestellten Optionen. Die Zykluszeit und die IP-Adresse des DNC-Rechners gehören zu den wichtigsten Parameter, die man beim Ausführen des Programms setzen kann. Weil der OPC UA Server auf dem DNC-Rechner laufen soll, ist die voreingestellte IP-Adresse gleich 127.0.0.1 (localhost), siehe auch Abbildung 1.2.

Im Quellcode von servermain.cpp, außerhalb der „main“-Funktion werden auch global einige Variablen und Zeiger deklariert, u.a. die Zeiger `pServer`, `pNM` und `pObject` und zwei Zeiger auf Strukturen, die später in MainCycle die aktualisierten Maschinendaten vom DNC-Treiber enthalten werden.

```
1  OpcServer* pServer;
2  EmcoNS::NodeManagerEmcoNS* pNM;
3  EmcoNS::MachineType* pObject;
4
5  struct struct_DNCstate { unsigned int Door, Clamp, Coolant, Airpurge,
6                           FeedOverride, SpindleOverride;
7                           char ProgramState; const char * ProgramName;
8                           bool Emergency; char Mode;};
9
10 struct_DNCstate* DNCState_Now;
11 struct_DNCstate* DNCState_Before;
```

**Code-Snippet 1: Deklarationen der Zeiger pServer, pNM, pObject, DNCState_Now und DNCState_Before**

Dabei ist `EmcoNS` der schon im UaModeler definierte Namensraum für die projektspezifischen Knoten. `NodeManagerEmcoNS` ist eine abgeleitete Klasse von NodeManagerBase des SDKs. Die Klasse `OPCServer` wird durch die Dateien opcserver.h und opcserver.cpp auch vom SDK zur Verfügung gestellt. Die Zeiger `pServer`, `pNM` und `pObject` zeigen jeweils auf das Serverobjekt, den Nodemanager bzw. das Maschinenobjekt. `DNCState_Now` und `DNCState_Before` werden in Kapitel 3.3.2 behandelt.

### 3.3.1 Die Funktion MainInit

Der Ablaufplan dieser Funktion ist in Abbildung 3.14 dargestellt. Bei Aufruf dieser Funktion aus „main“ werden ihr die vorher erwähnten Parameter IP-Adresse des DNC-Rechners und Zykluszeit übergeben. (Schritte 1 und 2)

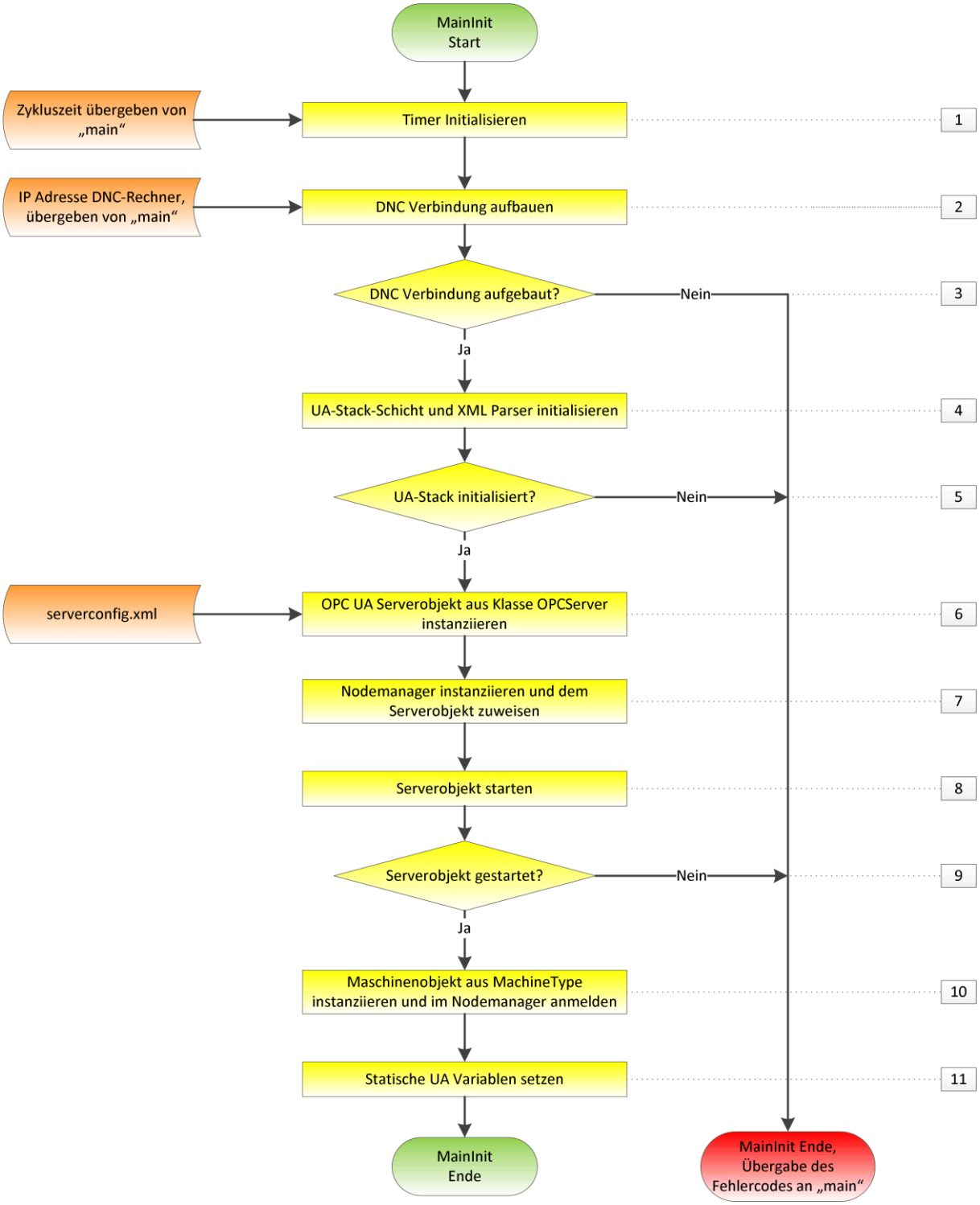


**Abbildung 3.14: Ablaufplan der Funktion MainInit**

Mit Code-Snippet 1 (entspricht Schritt 6 in Abbildung 3.14) wird das Serverobjekt instanziiert und die Server-Konfigurationsdatei geladen, dabei ist

`sConfigFileName` hier der Pfad zur Datei Serverconfig.xml die man aus den Beispielen des SDKs übernehmen kann. Diese Konfigurationsdatei enthält als XML u.a. Einstellungen für die Anmeldung von Clients, wie die unterstützten „Securitypolicies“ oder Höchstzahl der Clients, die sich mit dem Server verbinden können usw.

```
pServer = new OpcServer;
pServer->setServerConfig(sConfigFileName, szAppPath);
```

**Code-Snippet 2: Instanziierung des Serverobjekts mit Einstellungen aus ServerConfig.xml**

Weiter in Code-Snippet 3 wird eine Instanz des Nodemanagers erstellt und am zuvor instanziierten Server mit der Funktion `addNodeManager` angemeldet. Mit `pServer->start()` wird der Start des Serverobjekts initiiert, wobei die Rückgabe der Funktion überprüft wird, um sicherzustellen, dass das Serverobjekt richtig gestartet ist. (vgl. Schritte 7 bis 9 in Abbildung 3.14)

```
1  pNM = new EmcoNS::NodeManagerEmcoNS(true);
2  pServer->addNodeManager(pNM);
3
4  RetServStart = pServer->start();
5  if ( UAInitRet != 0 )
6  {
7        delete pServer;
8        printf("OPC UA could not be started! Error: %i\n", RetServStart);
9        return RetServStart;
10 }
11 else {printf("OPC UA Server started!\n");}
```

**Code-Snippet 3: Instanziierung Nodemanager und Starten des Serverobjekts**

Nun kann das Objekt der Maschine vom Typ MachineType (siehe auch Kap. 3.1.9) instanziiert werden (siehe Code-Snippet 4). Der Objekttyp MachineType wurde ja, wie oben gezeigt, im UaModeler schon definiert. Bei Untersuchung des vom UaModeler generierten Codes erkennt man, dass MachineType sich von MachineTypeBase und dies wiederum von BaseObjectType ableitet, der allgemein für jedes OPC UA Objekt verwendet wird. Die Argumente, die beim Instanziieren an den Konstruktor übergeben werden, beinhalten die Identifikation des Knotens, den Anzeigenamen des Objekts und den zugehörigen indizierten Namensraum. `defResourceName` ist eine konstante Zeichenkette mit dem Namen des Objekts und ist vorher global definiert worden als „EMCO CONCEPT TURN 55“

```
pObject = new EmcoNS::MachineType(
              UaNodeId(defResourceName, pNM->getNameSpaceIndex()),
              defResourceName, pNM->getNameSpaceIndex(), pNM);

pNM->addNodeAndReference(OpcUaId_ObjectsFolder, pObject,OpcUaId_Organizes);
```

**Code-Snippet 4: Instanziierung und Referenzieren des Maschinenobjekts**

Mit der Methode `addNodeAndReference` erstellt der Nodemanager eine OPC UA Referenz zwischen zwei Knoten. OPC UA Referenzen sind genauso wie Objekte

typisiert, die Basis-Referenztypen sind also Knoten und haben auch eine definierte Kennung (NodeId). Diese Kennung wird hier als Argument durch die Konstante `OpcUaId_Organizes` der Methode `addNodeAndReference` übergeben. Die ersten beiden Argumente sind die Quelle und das Ziel der Referenz. `OpcUaId_ObjectsFolder` ist eine Konstante, die die Knotenkennung von „ObjectsFolder“ entspricht (s. Informationsmodell). D.h., hier wird die Beziehung aufgebaut: ObjectsFolder „organisiert“ (hierarchische Referenz) das Objekt auf das `pObject` zeigt.

Die bis hier beschriebenen Codefragmente sind notwendig, um das im UaModeler erstellte Informationsmodell der Maschine (von einem Client aus) zu sehen und zu durchsuchen. In MainInit werden noch einige statische OPC UA Variablen gesetzt. Die zyklische Aktualisierung der Variablen geschieht in der Funktion MainCycle. Ein Aufruf der modellierten OPC UA Methoden würde jedoch noch keinen Effekt haben. Der Programmrumpf der OPC UA Methoden befindet sich in den entsprechenden Quelldateien, die in Kapitel 3.4 behandelt werden.

### 3.3.2 Die Funktion MainCycle

MainCycle wird – wie im Ablaufplan von „main“ zu sehen (Abbildung 3.13) – durch den Eventhandler des Timers gestartet. Deshalb wird der Eventhandler (obwohl nicht Teil von MainCycle) hier in Code-Snippet 5 angeführt und behandelt:

```
1  void OnTimedEvent( Object^ source, ElapsedEventArgs^ e )
2  {
3        iRetMainCycle = MainCycle();
4        if (iRetMainCycle != 0)
5        {     iBadCycles++;
6              if (iBadCycles >= defMaxBadCycles)
7              {
8                    printf("Error: Too many Cycles unsuccessfull!\n");
9                    printf("Initiating Shutdown!\n");
10                   Shutdown_Sequence();
11             }
12       }
13 }
```

**Code-Snippet 5: Eventhandler des Timers**

Der Eventhandler ruft also die Funktion MainCycle auf und überprüft ihren Rückgabewert. Wenn MainCycle zu oft einen Fehler ausgibt – z.B. wenn im ersten Schritt von MainCylce das Lesen vom DNC öfters erfolglos war – wird eine Fehlermeldung ausgegeben und Shutdown_Sequence aufgerufen. `defMaxBadCycles` ist ein konstanter Integer und ist global in servermain.cpp definiert.

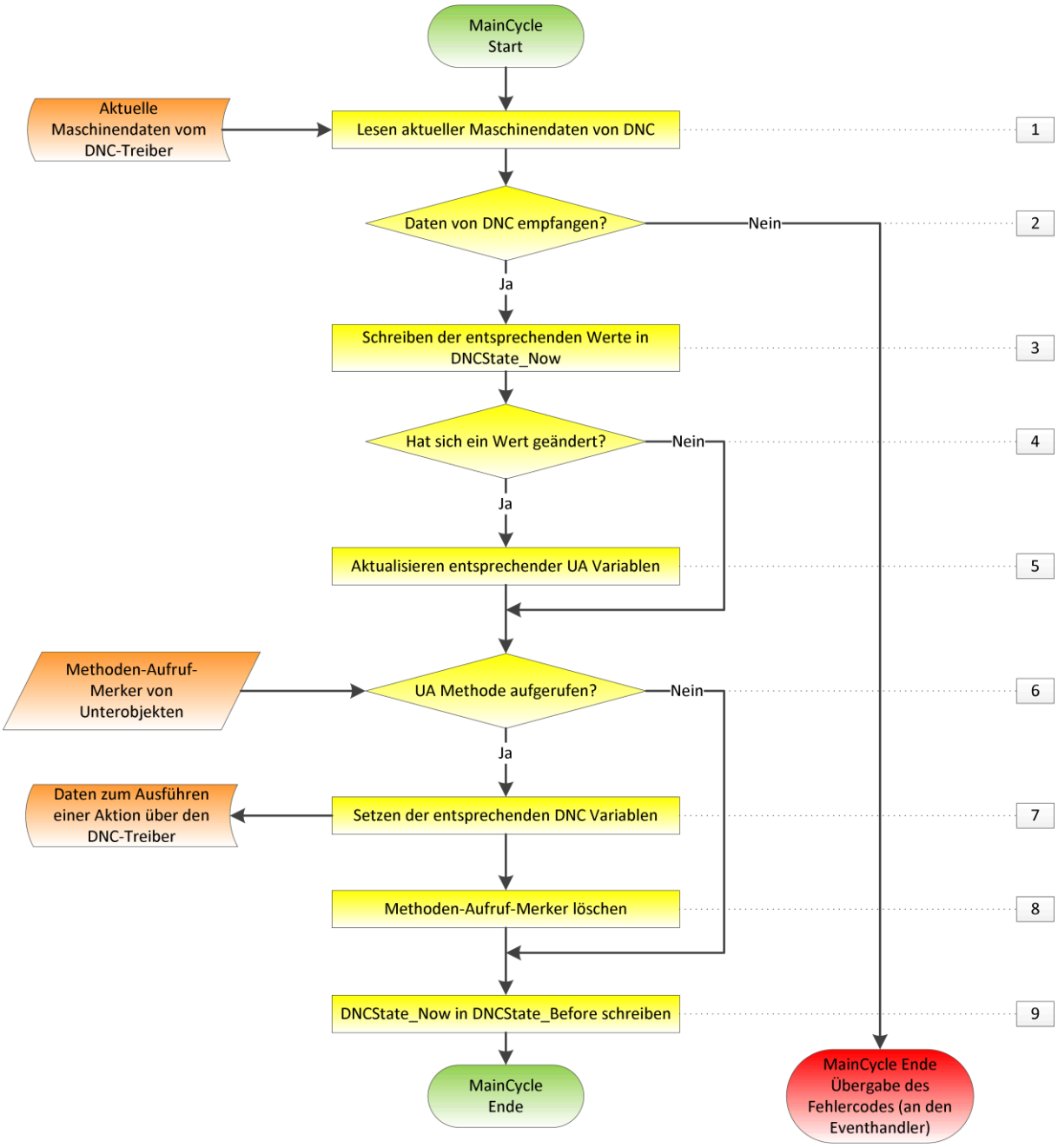


**Abbildung 3.15: Ablaufplan der Funktion MainCycle**

Der Ablaufplan von MainCycle ist in Abbildung 3.15 dargestellt. Im ersten Schritt werden relevante Daten über die DNC-Verbindung mit Hilfe der zuvor instanziierten Objekte `turn55` und `turn55State` (Typendefinitionen dazu sind nicht Teil dieser Arbeit, siehe [10]) abgefragt. Liefert der DNC-Treiber diese Maschinendaten nicht, so wird MainCycle abgebrochen und -1 zurückgegeben (siehe Code-Snippet 6). Die aktuellen Maschinendaten werden dann in Schritt 3 durch Aufruf von Memberfunktionen von `turn55State` eins zu eins in den entsprechenden Variablen in die Datenstruktur auf die `DNCState_Now` zeigt (deklariert in Code-Snippet 1) kopiert. Hier sei noch angemerkt, dass einige Funktionen von `turn55State` andere Namen haben als die entsprechenden Variablen der `DNCState_Now` Struktur.

```
DNCGetStateResult = turn55->getState(turn55State.Get());
if (DNCGetStateResult != 0) {return -1;}

DNCState_Now->Door = turn55State->Door();
DNCState_Now->Clamp = turn55State->Clamp();
DNCState_Now->Coolant = turn55State->Cool();
DNCState_Now->Airpurge = turn55State->Blow();
DNCState_Now->FeedOverride = turn55State->Feed();
DNCState_Now->SpindleOverride = turn55State->Spindle();
DNCState_Now->ProgramState = turn55State->ProgramState();
DNCState_Now->ProgramName = turn55State->ProgramName();
DNCState_Now->Emergency = turn55State->Emergency();
DNCState_Now->Mode = turn55State->Mode();
```

**Code-Snippet 6: Abfrage aktueller Daten durch Aufruf von Funktionen des DNC-Treibers**

Am Ende von MainCycle werden alle Werte der Datenstruktur `DNCState_Now` als Ganzes in `DNCState_Before` geschrieben, und beim nächsten Zyklus wird `DNCState_Now` wieder mit aktualisierten Werten von `turn55State` beschrieben. In den meisten Zyklen (z.B. bei 100ms Zykluszeit) werden sich diese Werte nicht ändern. Deshalb müssen sich auch die OPC UA Variablen nicht in jedem Zyklus aktualisieren, sondern nur bei Änderungen der Werte gegenüber dem letzten Zyklus. Die Überprüfung, ob sich ein Wert verändert hat, wird in Schritt 4 (Abbildung 3.15: Ablaufplan der Funktion MainCycle) für jedes Subobjekt der Maschine durchgeführt. Beispielhaft wird hier der Code für das Setzen der Door_Status Variable vom Objekt Loading_Door in Code-Snippet 7 erklärt. Hier sind die Konstanten `OPEN` und `CLOSE` Präprozessor-Definitionen, die respektive Integer-Zahlen 0 und 1 entsprechen, und `Stringdef_CLOSE`, `Stringdef_OPEN` und `Stringdef_UNDEFINED` respektive den Strings „OPEN“, „CLOSE“ und „UNDEFINED“.

```
if (DNCState_Now->Door != DNCState_Before->Door)
{
  if (DNCState_Now->Door == CLOSE) {              // CLOSE = 1
      pObject->getLoading_Door()->setDoor_Status(Stringdef_CLOSE);
      pObject->getClamping_System()->bCheck_DoorIsClosed = true;
      pObject->getNC()->bCheck_DoorIsClosed = true;
      pObject->getClamping_System()->bDoorIsUndef = false; }
  else if (DNCState_Now->Door == OPEN) {          // OPEN = 0
      pObject->getLoading_Door()->setDoor_Status(Stringdef_OPEN);
      pObject->getClamping_System()->bCheck_DoorIsClosed = false;
      pObject->getNC()->bCheck_DoorIsClosed = false;
      pObject->getClamping_System()->bDoorIsUndef = false; }
  else {
      pObject->getLoading_Door()->setDoor_Status(Stringdef_UNDEFINED);
      pObject->getClamping_System()->bCheck_DoorIsClosed = false;
      pObject->getNC()->bCheck_DoorIsClosed = false;
      pObject->getClamping_System()->bDoorIsUndef = true; }
}
```

**Code-Snippet 7: Schreiben auf die OPC UA Variable Door_Status von Subobjekt Loading_Door**

Es werden in diesem Block auch einige boolesche Variable anderer Objekte gesetzt, z.B.: `pObject->getNC()->bCheck_DoorIsClosed`. Dies ist notwendig, damit bei Aufruf von bestimmten OPC UA Methoden dieser Objekte, überprüft werden kann,

ob sie ausgeführt werden dürfen (siehe Kap. 3.4). Der Code für die Schritte 6 bis 8 aus dem Ablaufplan von MainCycle ist für einige Objekte in Code-Snippet 8 angeführt. Der Begriff „Methoden-Aufruf-Merker“ im Ablaufplan entspricht hier z.B. die Integer-Variable `tempstatus` des Objekts Loading_Door (Zeilen 3 bis 11). `tempstatus` kann Werte zwischen 0 und 3 annehmen und wird bei Aufruf der OPC UA Methoden Open_Door und Close_Door entsprechend auf 0 oder 1 (gleich OPEN bzw. CLOSE) gesetzt. Wird z.B. die Methode Open_Door aufgerufen, wird `tempstatus` gleich 0, und im MainCycle werden mit `setAuxDoor(0)` (Member-Funktion des Objekts turn55) die entsprechenden Aktionen fürs Öffnen der Maschinentür über den DNC-Rechner eingeleitet. Bis der Endzustand (offene Tür) erreicht wird, vergehen einige Sekunden. MainCylce wird aber nicht angehalten und läuft somit in dieser Zeit bis zu 20-mal in der Sekunde ab. `tempstatus` wird deshalb auf 3 gesetzt, damit bei den nachfolgenden Zyklen nicht immer wieder `setAuxDoor` gesetzt wird. Anders ausgedrückt: `tempstatus` ist in fast allen Zyklen gleich 3, nur in der Zeit vom entsprechenden Methoden-Aufruf bis zum Setzen von `setAuxDoor` hat sie einen anderen Wert (0 oder 1), und diese Zeit ist maximal nur ein Zyklus lang.

```
1  // ==================== Setting DNC Variables ======================
2  // Loading_Door
3  if (pObject->getLoading_Door()->tempstatus == OPEN)    // OPEN = 0
4  {
5      turn55->setAuxDoor(0);
6      pObject->getLoading_Door()->tempstatus = 3;
7  }
8  if (pObject->getLoading_Door()->tempstatus == CLOSE)   // CLOSE = 1
9  {
10     turn55->setAuxDoor(1);
11     pObject->getLoading_Door()->tempstatus = 3;
12 }
13 // Clamping_System
14 ...
15 // Air_Purge_System
16 ...
17 // Coolant_System
18 ...
19 // Set_Feed_Override
20 if (pObject->getFeed()->FeedOverrideSet == TRUE)
21 {
22     turn55->setOverrideFeed(pObject->getFeed()->FeedOverrideTemp);
23     pObject->getFeed()->FeedOverrideSet = FALSE;
24 }
25 // Set_Spindle_Override
26 ...
27 // Assign_NC_Program
28 if ((pObject->getNC()->ncprogramtemp != "") &&
29      (pObject->getNC()->ncprogexists))
30 {
31     turn55->setProgram((UaString("$MF") +
32                 pObject->getNC()->ncprogramtemp).toUtf8());
33     pObject->getNC()->ncprogramtemp = "";
34 }
35 // Reset_NC
36 ...
```

```
37 // Stop_NC
38 if (pObject->getNC()->ncprogstattemp == 'S')
39 {
40     turn55->setProgramState('S');
41     pObject->getNC()->ncprogstattemp = ' ';
42 }
43 // Start_NC
44 ...
```

Code-Snippet 8: Setzen der DNC-Variablen mit Hilfe der „Methoden-Aufruf-Merker“

Der entsprechende Code für die Subobjekte Clamping_System, Air_Purge_System, Coolant_System sind (fast) identisch. Ähnlich ist auch der entsprechende Code für das Subobjekt NC bei seinen Methoden Start_NC, Stop_NC und Reset_NC (siehe auch 3.1.3).

### 3.3.3 Die Prozedur Shutdown_Sequence

Shutdown_Sequence hat keinen Rückgabewert und ist deshalb per Definition keine Funktion sondern eine Prozedur. Sie leitet lediglich das ordnungsgemäße Herunterfahren des Servers ein, in dem einzelne Objekte gestoppt und aus dem Speicher wieder gelöscht werden (siehe Abbildung 3.16) Nachdem diese Prozedur abgearbeitet ist, sind wir schon am Ende der „main“-Funktion (siehe Abbildung 3.13) und das Programm endet.

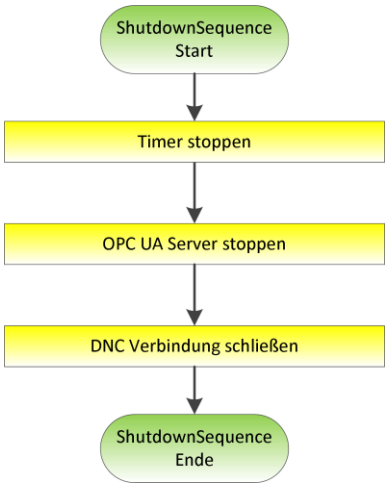


Abbildung 3.16: Ablaufplan der Prozedur Shutdown_Sequence

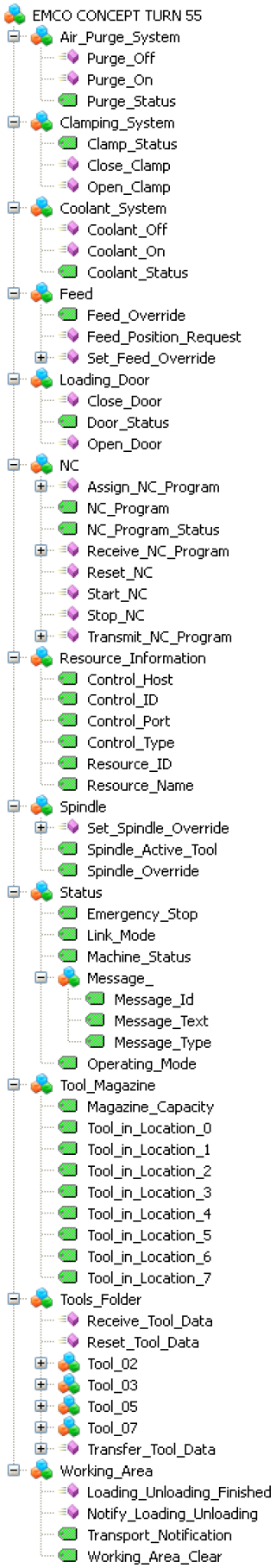


**Abbildung 3.17: Der Adressraum der modellierten WZM im laufenden Betrieb**

## 3.4 Programmablauf der Methoden

### 3.4.1 Methoden der Hilfsantriebe (Auxiliary)

Unter den Hilfantrieben bzw. Auxiliary werden hier die Subobjekte Loading_Door, Clamping_System, Air_Purge_System und Coolant_System verstanden. Wie schon in Kapitel 3.1.1 beschrieben, enthalten diese Objekte jeweils zwei Methoden, deren Namen schon selbsterklärend sind. Jedes dieser Objekte enthält auch eine Integer-Variable namens `tempstatus`, die der Funktion MainCycle in Servermain, die gewünschte Statusänderung des Objekts übermittelt. Die einzelnen `tempstatus`-Variablen sind also die sogenannten „Methoden-Aufruf-Merker“, die in Ablaufplan von MainCycle in Schritt 6 abgerufen werden (siehe Kapitel 3.3.2). Der Aufruf jeder dieser Methoden verändert den Wert der Variable `tempstatus` falls die Bedingungen für die Durchführung der Aktion erfüllt sind. Die erste Bedingung ist stets die Überprüfung, ob die Status-Variable des Objekts schon gleich dem Wert ist, der durch den Methodenaufruf erreicht werden soll; z.B. wird beim Aufruf von Open_Door zuerst überprüft, ob die Beladetür schon geöffnet ist. Da die Methoden dieser Kategorie alle ähnlich sind, sind hier nur der Ablaufplan (Abbildung 3.18) und das Code-Snippet (Code-Snippet 9) der Methode Open_Door dargestellt.

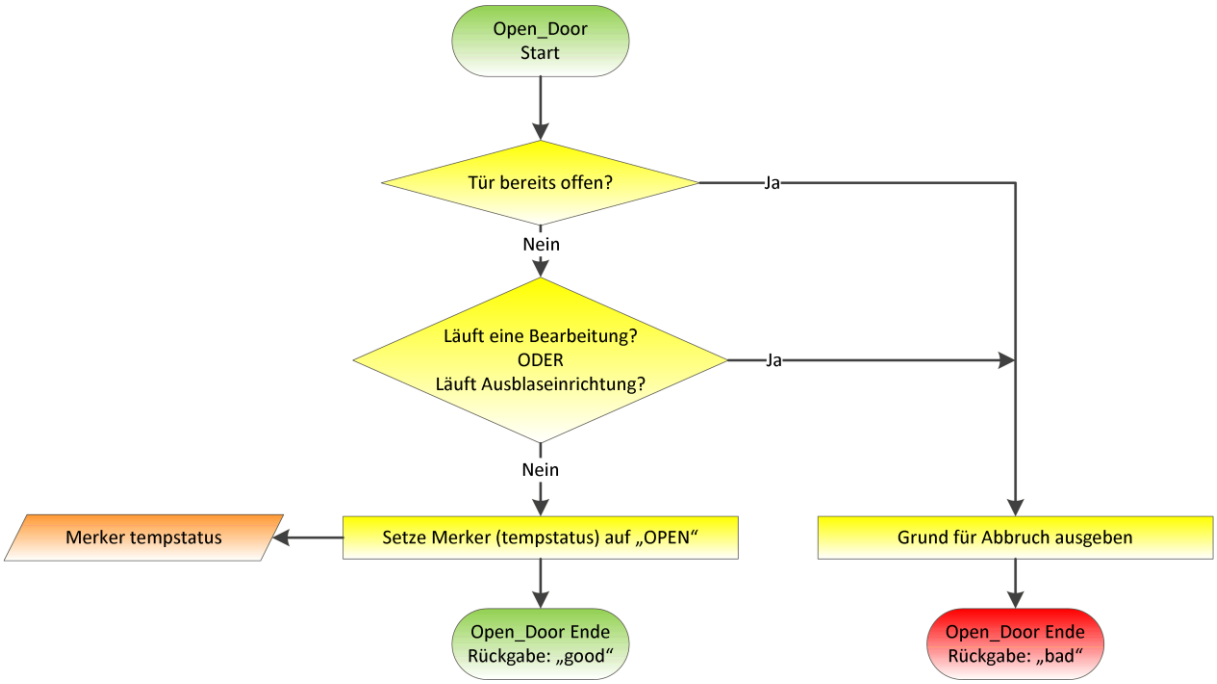


**Abbildung 3.18: Ablaufplan der Methode Open_Door**

```
1  UaStatus LoadingDoorType::Open_Door(const ServiceContext&
2                                                  serviceContext)
3  {
4     UaStatus ret;
5     if (getDoor_Status() == Stringdef_OPEN)
6     {
7       printf("Error: Loading Door is already opened!\n");
8       ret = OpcUa Bad;
9     }
10    else
11    {
12       if (this->bNCProgIsActive || this->bPurgeIsOn)
13       {
14         printf("Error: Open_Door call denied, NC Program is active or
15                 purge is on!\n");
16         ret = OpcUa_Bad;
17       }
18       else
19       {
20         this->tempstatus = OPEN;
21         printf("Open_Door called!\n");
22         ret = OpcUa_Good;
23       }
24    }
25    return ret;
26 }
```

**Code-Snippet 9: Methode Open_Door**

Die Bedingungen für die restlichen Methoden dieser Kategorie sind bei jedem Objekt unterschiedlich und sind gekoppelt an Status-Variablen anderer Objekte. Die Logik, die den Methodenaufruf unterbrechen soll, ist jeweils in den folgenden Tabellen aufgelistet.

**Tabelle 3.1: Bedingungen für das Unterbrechen der Methoden des Objekts Loading_Door**

| **Objekt: Loading_Door** | | | |
|---|---|---|---|
| **Methode: Open_Door** | | | |
| **abhängig vom Objekt** | **Unterbrechen wenn** | | **Begründung** |
| NC | Bearbeitung läuft | NC_Program_Status = "ACTIVE" | Beladetür muss stets geschlossen sein, während Bearbeitung |
| Air_Purge_System | Ausblaseinrichtung an | Purge_Status = "ON" | Ausgeblasene Späne könnten Bediener verletzen |
| **Methode: Close_Door** | | | |
| **abhängig vom Objekt** | **Unterbrechen wenn** | | **Begründung** |
| Clamping_System | Spannmittel offen | Clamp_Status = "OPEN" | Kontrolle, ob Werkstück gespannt ist, bevor Tür geschlossen wird. |
| Air_Purge_System | Ausblaseinrichtung an | Purge_Status = "ON" | Instabilität des DNC-Treibers, unbekannt |

**Tabelle 3.2: Bedingungen für das Unterbrechen der Methoden des Objekts Clamping_System**

| **Objekt: Clamping_System** | | | |
|---|---|---|---|
| **Methode: Open_Clamp** | | | |
| **abhängig vom Objekt** | **Unterbrechen wenn** | | **Begründung** |
| NC | Bearbeitung läuft | NC_Status = "ACTIVE" | Werkstück muss stets gespannt sein während Bearbeitung |
| Loading_Door | Beladetür zu | Door_Status = "CLOSED" | Solange Tür nicht offen, muss Werkstück gespannt bleiben |
| Loading_Door | Status Beladetür unbek. | Door_Status = "UNDEFINED" | Solange Tür nicht offen, muss Werkstück gespannt bleiben |
| **Methode: Close_Clamp** | | | |
| **abhängig vom Objekt** | **Unterbrechen wenn** | | **Begründung** |
| Loading_Door | Beladetür zu | Door_Status = "CLOSED " | Sinnlos bei geschlossener Beladetüre |
| Loading_Door | Status Beladetür unbek. | Door_Status = "UNDEFINED" | Sinnlos bei geschlossener Beladetüre |

**Tabelle 3.3: Bedingungen für das Unterbrechen der Methoden des Objekts Air_Purge_System**

| **Objekt: Air_Purge_System** | | | |
|---|---|---|---|
| **Methode: Purge_On** | | | |
| **abhängig vom Objekt** | **Unterbrechen wenn** | | **Begründung** |
| NC | Bearbeitung läuft | NC_Program_Status = "ACTIVE" | Um Bearbeitung nicht zu stören, weil Ausblasen kann Teil eines NC-Programms sein |
| **Methode: Purge_Off** | | | |
| **abhängig vom Objekt** | **Unterbrechen wenn** | | **Begründung** |
| NC | Bearbeitung läuft | NC_Program_Status = " ACTIVE " | Um Bearbeitung nicht zu stören, weil Ausblasen kann Teil eines NC-Programms sein |

## 3.4.2 Methoden zur Änderung der Override-Werte

Die Methoden Set_Feed_Override des Objekts Feed und Set_Spindle_Override des Objekts Spindle sind hier die einfachsten Methoden mit einer Eingabe. Wie schon in Kapitel 3.1.2 beschrieben, ist das Eingabeargument als Byte (abgeleitet vom abstrakten Typ UInteger, siehe Abbildung 2.6) definiert. Demnach wird ein OPC UA Client bei Aufruf dieser Methoden schon die mögliche Eingabe auf ganze Zahlen zwischen 0 und 255 beschränken. Bei der Programmierung musste deshalb nur kontrolliert werden, ob der eingegebene Wert zwischen 0 und 120 liegt, wie in Abbildung 3.19 und Code-Snippet 10 zu sehen. Hier wird - falls der eingegebene Wert passt - die boolesche Variable `FeedOverrideSet` auf `TRUE` gesetzt und der eingegebene Wert in `FeedOverrideTemp` kopiert. Beide Variablen sind „public members“ des Objekttyps Feed und können in MainCycle abgefragt werden. Ist

`FeedOverrideSet TRUE`, wird in MainCycle `FeedOverrideTemp` gelesen und der Wert durch die entsprechende Setzen-Funktion des DNC-Objekts turn55 an die Maschine übermittelt (siehe Code-Snippet 8, Zeilen 19-24).

Set_Spindle_Override läuft genauso ab, lediglich die Namen der Variablen sind andere.

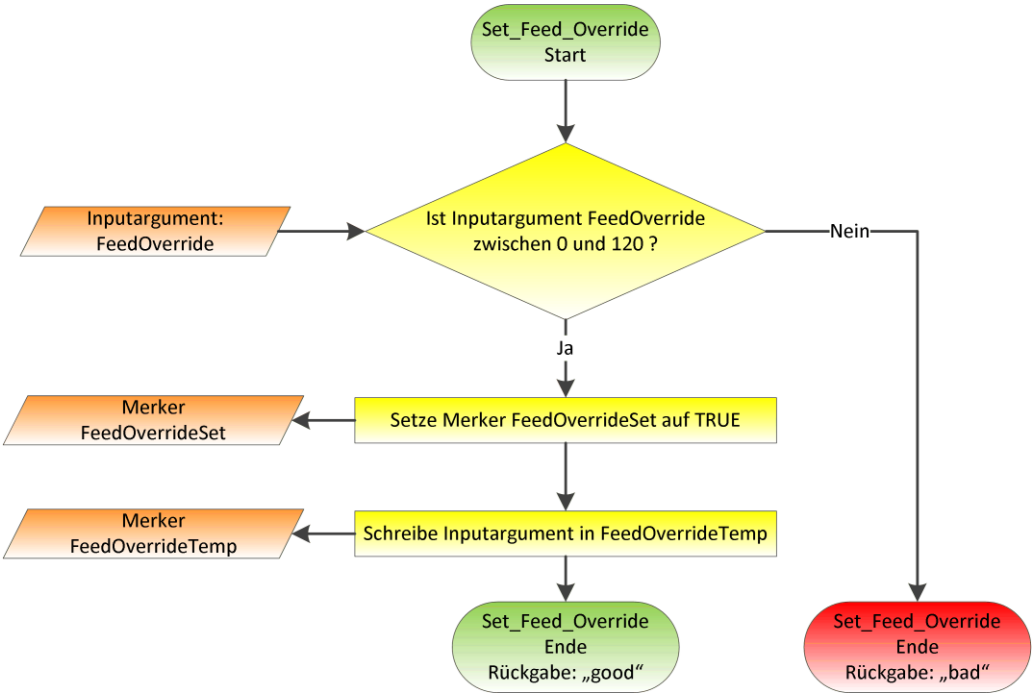


Abbildung 3.19: Ablaufplan der Methode Set_Feed_Override

```
1  UaStatus FeedType::Set_Feed_Override(const ServiceContext&
2                                                  serviceContext,
3                                          /*IN*/OpcUa_Byte  FeedOverride)
4  {
5        if ((FeedOverride >= 0) && (FeedOverride <=120))
6        {
7              this->FeedOverrideSet = TRUE;
8              this->FeedOverrideTemp = FeedOverride;
9              return OpcUa_Good;
10       }
11       else return OpcUa_Bad;
12 }
```

Code-Snippet 10: Methode Set_Feed_Override

## 3.4.3 Methoden des Objekttyps NCType

### 3.4.3.1 Assign_NC_Program

Mit dieser Methode kann ein auf dem DNC-Rechner vorhandenes NC-Hauptprogramm (*.MPF) in die Steuerung geladen werden. Wie in Abbildung 3.20

ersichtlich, kann ein NC-Programm nur dann zugewiesen werden, falls gerade kein NC Program läuft, d.h., falls die Variable NC_Programm_Status nicht „ACTIVE“ ist. Diese Methode kann auch dazu genutzt werden, um eine Liste der vorhandenen Hauptprogramme auf der Steuerung zu erzeugen. Dafür muss eine leere Zeichenkette eingegeben werden. Die Ausgabe ist ein String, der zeilenweise alle Dateien des Hauptprogrammordners beinhaltet. Der Pfad dieses Ordners ist als eine konstante Zeichenkette definiert. Wenn im Eingabefeld ein Dateiname eingegeben wird, die nicht existiert, wird als Ausgabe eine Fehlermeldung ausgegeben. Diese Methode gibt als Rückgabe-Code auch dann „good“ aus, weil sie – als Ausgabeargument – einen String und damit ein Ergebnis ausgibt.

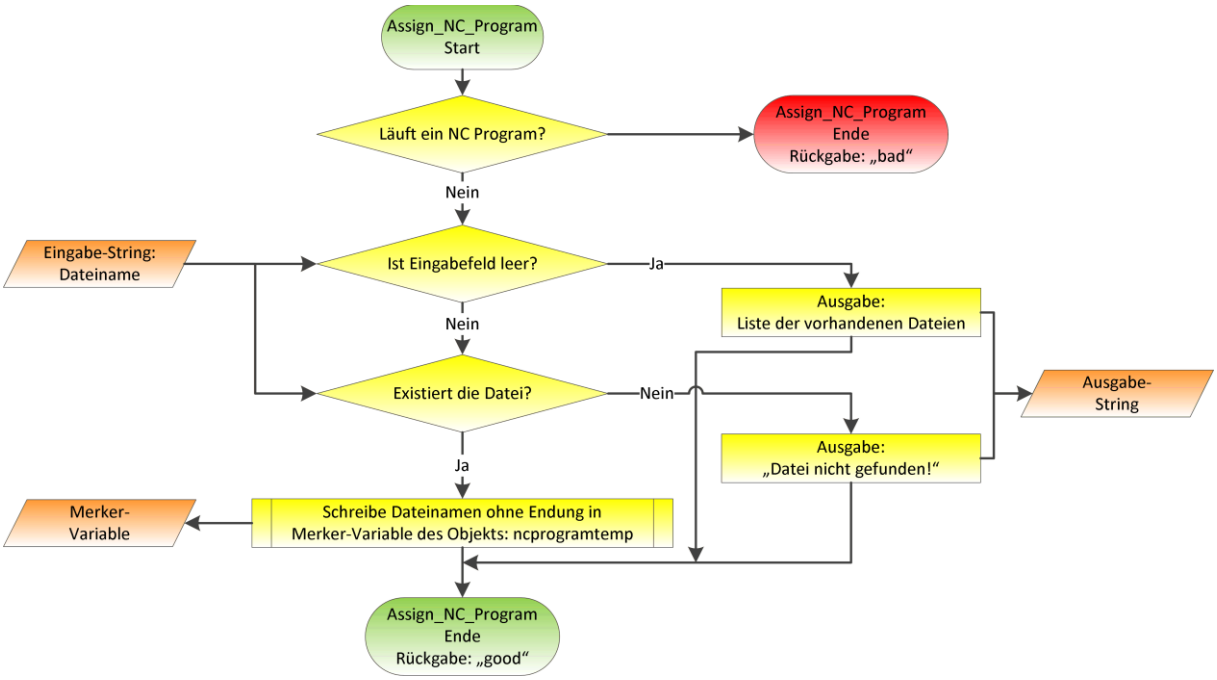


**Abbildung 3.20: Ablaufplan der Methode Assign_NC_Program**

Wenn die eingegeben Datei existiert, wird in einer Merker-Variable („public member“ des Objekts NC) die Zeichenkette mit dem Dateinamen geschrieben und auch eine boolesche Variable gesetzt. MainCylce ruft dann die passende Funktion des DNC-Treibers auf und übergibt ihr den Dateinamen.

Im Code-Abschnitt dieser Methode werden – nach der Überprüfung der Bedingungen – hauptsächlich String-Operationen durchgeführt. Es wird darauf verzichtet hier den Code anzuführen. Abbildung 3.21 zeigt einen Screenshot vom OPC UA Client „UaExpert“ von Unified Automation. Dieser Standard-Client öffnet bei Anwahl einer Methode ein Windows-Form mit Eingabe- und Ausgabe-Felder (wenn bei der Methode Argumente vorhanden sind). Wenn die Datentypen der Argumente Strings sind, kann das entsprechende Feld auch mehrzeilig als Fenster dargestellt werden. Der Screenshot zeigt die Ausgabe bei einem Methodenaufruf ohne Eingabeargument, die - wie oben beschrieben - die vorhandenen Dateien auflistet. So könnte der Name eines NC-Programms gleich vom Ausgabe- ins Eingabefeld

kopiert werden, und somit die Methode mit dem Dateinamen als Eingabe erneut aufgerufen werden.

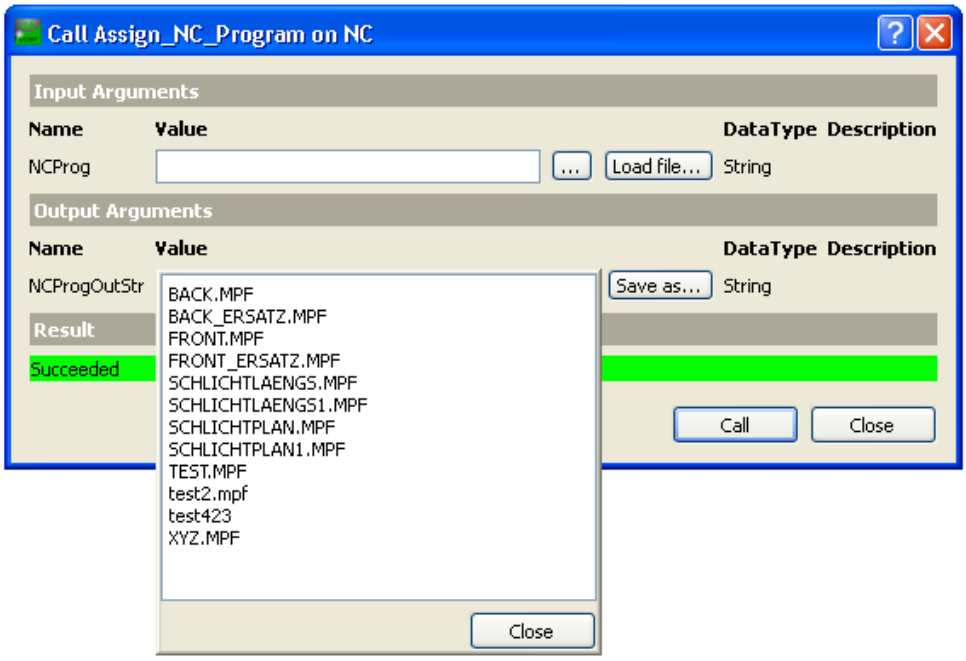


**Abbildung 3.21: Screenshot vom Methodenaufruf "Assign_NC_Program" des OPC UA Clients "UaExpert" von Unified Automation, Aufruf ohne Eingabe**

### 3.4.3.2 Receive_NC_Program

Receive_NC_Program wurde programmiert, um ein NC-Programm von einem OPC UA Client zum DNC-Rechner zu senden. Der OPC UA Server empfängt dabei eine Zeichenkette mit dem gesamten Inhalt der zu übertragenden Datei, und erstellt im MPF-Ordner des DNC-Rechners eine Datei mit demselben Namen und Inhalt. Wie auch in Abbildung 3.22 zu sehen, wird die Methode unterbrochen, falls auf dem DNC-Rechner schon eine Datei mit demselben Namen existiert.

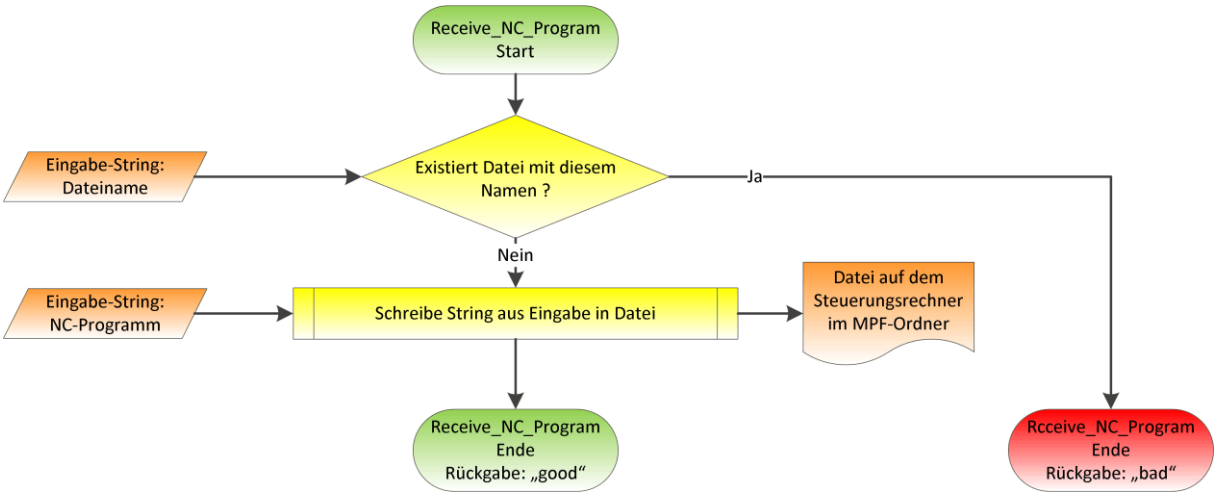


**Abbildung 3.22: Ablaufplan der Methode "Receive_NC_Program"**

Der Code dieser Methode ist unten in Code-Snippet 11 angeführt. Man beachte die Zeilen 17 bis 19. `UaDir` ist eine Klasse des SDKs, die einige Funktionen für Datei-Operationen beinhaltet, siehe auch [17]. Die Funktion `toNativeSeparators()` der Klasse `UaDir` überprüft einen Pfad und wechselt bei Bedarf die Trennzeichen im Pfad auf die nativen Trennzeichen des Betriebssystems. Diese Funktion kann verwendet werden, um mit dieser SDK bequemer plattform-unabhängige Applikationen zu programmieren.

Das Schreiben in eine neue Datei erfolgt durch Verwendung von Klassen der Standard C++ Bibliothek (std), wie in den Zeilen 23 bis 26 zu sehen.

```
1  UaStatus NCType::Receive_NC_Program(const ServiceContext&
2          serviceContext, /*IN*/const UaString&  FileNameToRec,
3          /*IN*/const UaString&  FileToRec)
4  {
5    UaStatus ret;
6    string tempstring1 = "";
7    UaDir * spath = new UaDir(NCPROGPATH);
8    // NCPROGPATH = "//Turn-pc/PRG/MPF.DIR/"
9    if (spath->exists(UaUniString(FileNameToRec.toUtf16())))
10   {
11     printf("A file with the same name already exists!\n");
12     ret = OpcUa_Bad;
13   }
14   else
15   {
16     //writing path with native seperators to tempstring1:
17     tempstring1 =
18         UaString(UaDir::toNativeSeparators(NCPROGPATH).
19         toUtf16()).toUtf8();
20     //appending tempstring1 with file name:
21     tempstring1.append(UaString(UaUniString(FileNameToRec.
22         toUtf16()).toUtf16()).toUtf8());
23     ofstream * pFile =
24         new ofstream(tempstring1.data(),ofstream::binary);
25     pFile->write(FileToRec.toUtf8(),FileToRec.length());
26     pFile->close();
27     printf("\n""%s"" created. \n",tempstring1.data());
28     ret = OpcUa_Good;
29   }
30   spath->~UaDir();
31   return ret;
32 }
```

**Code-Snippet 11: Methode Receive_NC_Program**

### 3.4.3.3 Transmit_NC_Program

Transmit_NC_Program kann aufgerufen werden um ein auf dem DNC-Rechner vorhandenes NC-Programm auf einem OPC UA Client zu öffnen bzw. abzuspeichern. Wie bei Assign_NC_Program kann hier eine Liste der vorhandenen .MPF-Dateien auf dem DNC-Rechner angezeigt werden, falls das Eingabefeld der Methode beim Aufruf leer gelassen wird. Wird dagegen im Eingabefeld der Name

einer vorhandenen Datei eingegeben, liest der Server die Datei und überträgt sie als eine Zeichenkette zum OPC UA Client.

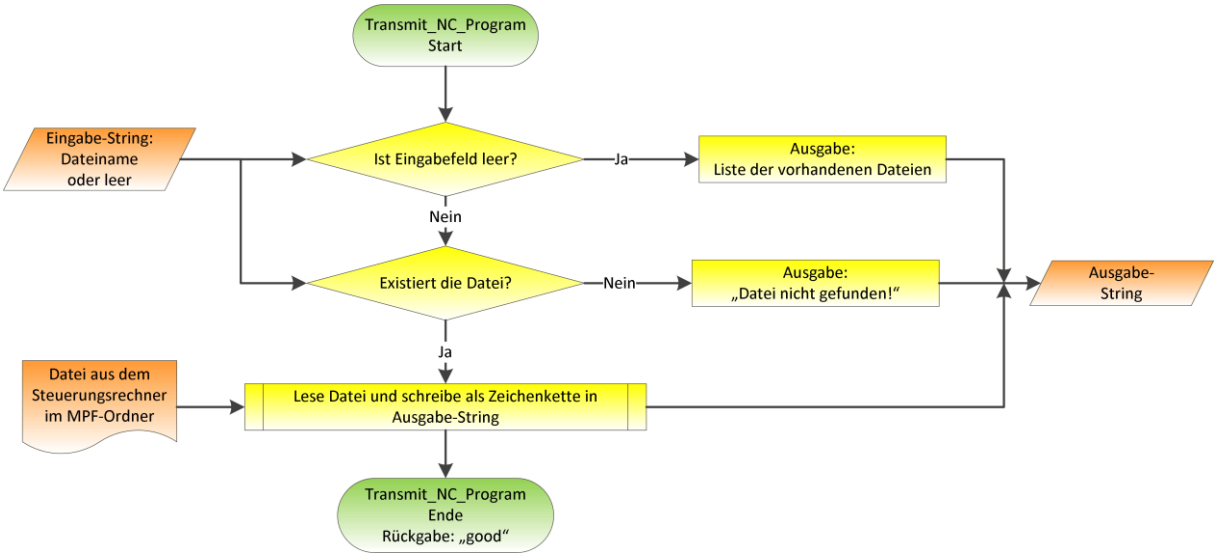


**Abbildung 3.23: Ablaufplan der Methode "Transmit_NC_Program"**

Abbildung 3.24 zeigt ein Screenshot von einem OPC UA Client nach Aufruf dieser Methode mit dem Dateinamen eines vorhandenen NC-Programms als Eingabeparameter. Im Ausgabefeld ist das NC-Programm zu sehen und kann mit dem Button „Save as…“ Client-seitig in eine Datei gespeichert werden.

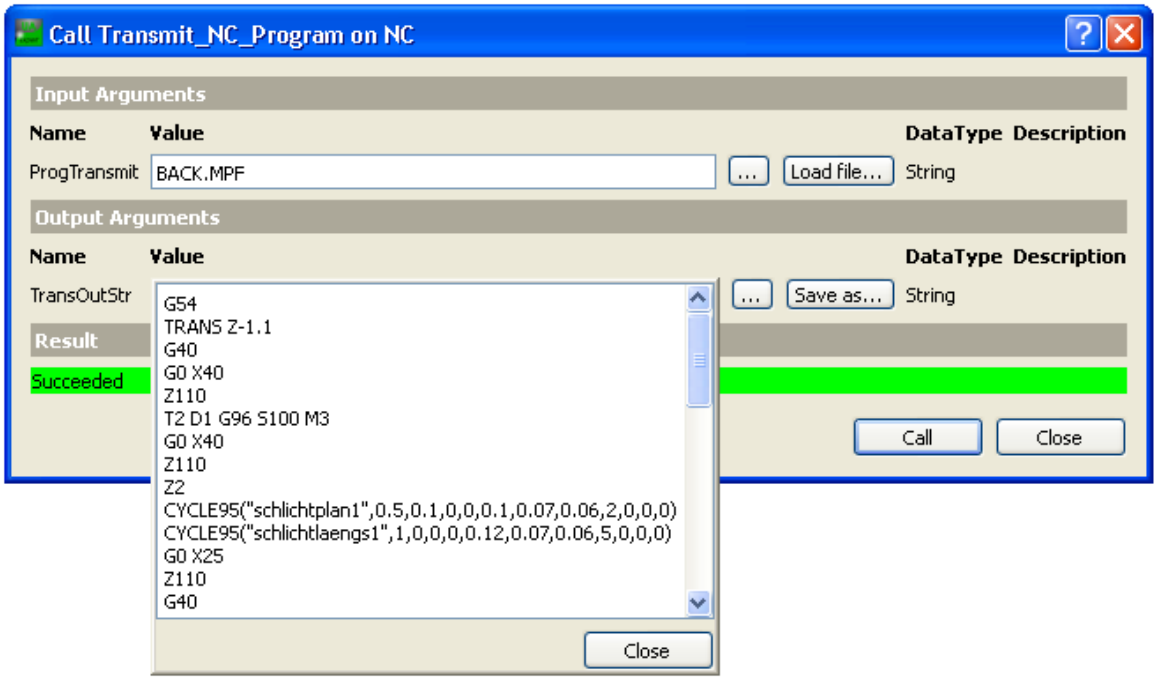


**Abbildung 3.24: Screenshot vom Methodenaufruf "Transmit_NC_Program" des OPC UA Clients "UaExpert" von Unified Automation, Aufruf mit Eingabe eines vorhandenen Dateinamen**

Da der Server auf ein Ergebnis des Methodenaufrufs wartet, könnte es sein, dass bei Übertragung von großen Dateien der Server zu lange nicht reagiert, und somit ein Timeout auslöst. Dies ist ein typisches Problem bei synchronen Methodenaufrufen. In diesem Kontext sollte ein Methode deshalb – anders als hier umgesetzt – so programmiert werden, dass er eine Aktion (wenn zulässig) einleitet, das Resultat der Aktion aber nicht abwartet. Das Resultat sollte eher durch eine ereignisbasierte Meldung generiert werden. Später in Kap. 5 wird auf OPC UA Programme als Alternativen zu asynchronen OPC UA Methoden eingegangen.

#### 3.4.3.4 Start_NC, Stop_NC und Reset_NC

Diese drei Methoden sind wie die Methoden der Hilfsantriebe (vgl. Kap. 3.4.1) sehr einfach. Die Methoden-Aufruf-Merker sind hier aber vom Typ „char“. Solange die Methode nicht aufgerufen wird, ist der Wert des Merkers `ncprogstattemp` ein Leerzeichen. Wenn Start_NC, Stop_NC oder Reset_NC erfolgreich aufgerufen werden, so nimmt der Merker jeweils die Werte `'L','S'` bzw. `'R'` an, siehe auch Abbildung 3.25.

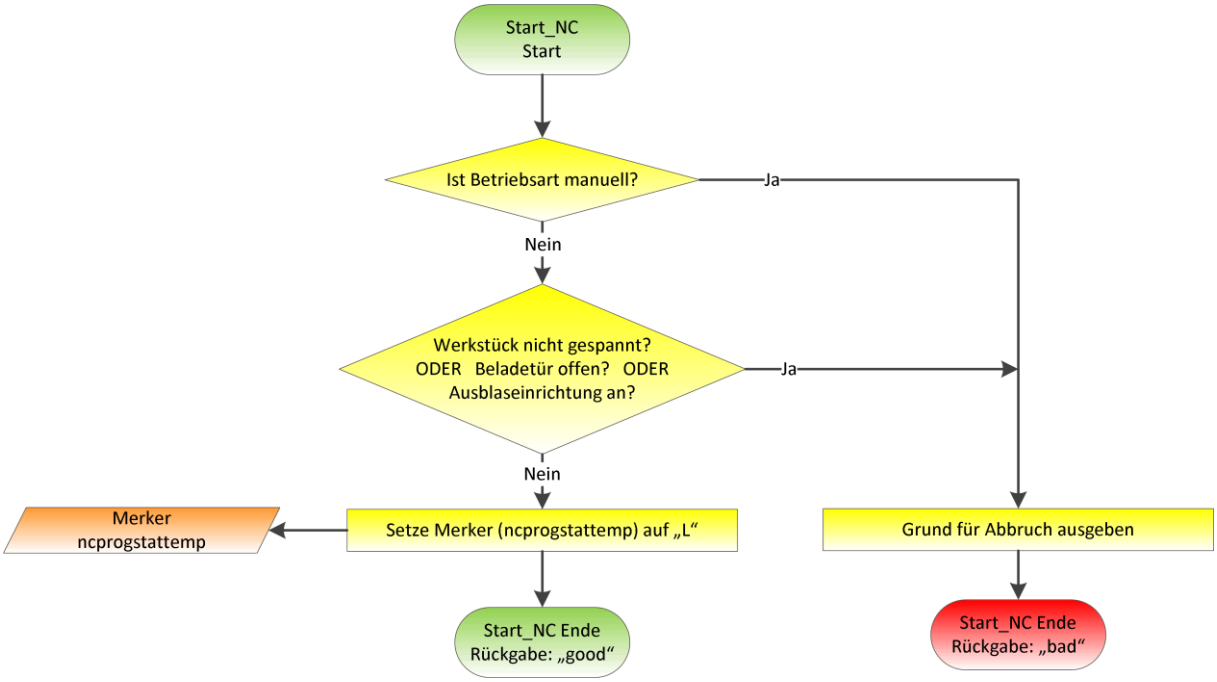


**Abbildung 3.25: Ablaufplan der Methode "Start_NC"**

In MainCycle wird dieser Merker überprüft. Wenn sein Wert nicht ein Leerzeichen ist, wird die entsprechende Funktion des DNC-Objekts aufgerufen und der Merker wieder zurückgesetzt auf den Wert des Leerzeichens. In Code-Snippet 8 ist in den Zeilen 37 bis 42 der entsprechende Code für den Methoden-Aufruf-Merker der Methode Stop_NC angeführt. Die Bedingungen, die zum Unterbrechen der Methoden führen sollten, sind unten in Tabelle 3.4 beschrieben. Die Methode Stop_NC sollte immer ausgeführt werden, und ist deshalb in der Tabelle nicht vorhanden.

**Tabelle 3.4: Bedingungen für das Unterbrechen der Methoden des Objekts NC**

| **Objekt: NC** | | | |
|---|---|---|---|
| **Methode: Start_NC** | | | |
| **abhängig vom Objekt** | **Unterbrechen wenn** | | **Begründung** |
| Status | Betriebsart manuell | Operating_Mode = "MANUAL" | Wenn Maschine in Betriebsart manuell, darf Bearbeitung nicht durch Fernsteuerung starten |
| Clamping_System | Spannmittel offen | Clamp_Status = "OPEN" | Solange Werkstück nicht gespannt, darf Bearbeitung nicht starten |
| Air_Purge_System | Ausblaseinrichtung an | Purge_Status = "ON" | Instabilität des DNC-Treibers, unbekannt |
| Loading_Door | Beladetür offen | Door_Status = "OPEN" | Solange Beladetür offen, darf Bearbeitung nicht starten |
| **Methode: Reset_NC** | | | |
| **abhängig vom Objekt** | **Unterbrechen wenn** | | **Begründung** |
| Status | Betriebsart manuell | Operating_Mode = "MANUAL" | Wenn Maschine in Betriebsart manuell, sollte Programm nicht durch Fernsteuerung zurückgesetzt werden können |
| Clamping_System | Spannmittel offen | Clamp_Status = "OPEN" | Instabilität des DNC-Treibers, unbekannt |
| Loading_Door | Beladetür offen | Door_Status = "OPEN" | Instabilität des DNC-Treibers, unbekannt |

```
1  UaStatus NCType::Start_NC(const ServiceContext&  serviceContext)
2  {
3    UaStatus ret;
4    if (this->bOperatingModeIsManual)
5    {
6      printf("Error: Operating_Mode is manual!\n");
7      ret=OpcUa Bad;
8    }
9    else
10   {
11     if (!this->bCheck_ClampIsClosed || // OR
12         !this->bCheck_DoorIsClosed ||  // OR
13         !this->bCheck_PurgeIsOff)
14     {
15        printf("Error: Current Auxiliary states don't allow
16                Start_NC!\n");
17        ret=OpcUa_Bad;
18     }
19     else
20     {
21        this->ncprogstattemp='L';
22        ret=OpcUa_Good;
23      }
24   }// end of first if-else
25   return ret;
26 }
```

**Code-Snippet 12: Methode Start_NC**

### 3.4.4 Methoden des Objekttyps Tools_Folder

Die Methoden Receive_Tool_Data und Transfer_Tool_Data sind in dem Quellcode der Klasse Tools_Folder programmiert, jedoch geschieht die Ausführung in MainCycle des Servermain. Wegen der langen Verzögerungszeit vom Aufruf der DNC-Funktion bis zur vollständigen Übermittlung der Daten, sind sie typische Fälle für asynchrone Methodenaufrufe. Nachfolgend wird die Methode Receive_Tool_Data genauer beschrieben.

Der Aufruf selbst setzt nur einen Merker, der in MainCylce (Code-Snippet 13 Zeile 3-4) überprüft wird, diese ist `flag_ReceiveToolData` (bzw. `flag_TransferToolData` für `Transfer_Tool_Data`). Mit der Funktion `turn55->getTools()` des DNC-Maschinen-Objekts wird das Herunterladen der vorhandenen Werkzeugdaten eingeleitet und die Variable `nStateToolDownload` inkrementiert. Solange der Datentransfer läuft, ist der Wert dieser Variable gleich 1 und in jedem Zyklus des Servers führt der `switch` zum `case 1`. Wenn der Transfer abgeschlossen ist, führt der `switch` zum `case 2;` die Werkzeugdaten sind dann am Server vorhanden. Diese können über die Funktion `turn55->getTool(nTool, lstTool)` (nicht zu verwechseln mit `getTool`**`s`**) aufgerufen werden (Zeile 30). Diese Funktion wird in eine Schleife 256-mal aufgerufen (d.h., 256 Werkzeug-Objekte können dynamisch generiert werden). Falls ein Werkzeug vorhanden ist, wird für dieses Werkzeug ein neues OPC UA Objekt „Tool_xx“ (xx für die Nummer des Werkzeugs) erzeugt, das die geladenen Werkzeugparameter jeweils als OPC UA Variablen enthält. (siehe auch Kap. 3.1.6 ToolsFolderType und ToolType) Die Zeilen 50 bis 55 erzeugen das jeweilige OPC UA Objekt, die Zeilen 56 bis 60 setzen entsprechende Werte für den BrowseName und DisplayName des Objekts. Schließlich setzen die Zeilen 62 bis 65 die Referenzen der Objekte, so dass sie hierarchisch unter dem Objekt „Tools_Folder“ registriert werden. Die Variablen der Tool_xx Objekte werden in den Zeilen 71 bis 78 gesetzt. Der Code wurde hier im Sinne der besseren Lesbarkeit abgekürzt und der abgekürzte Code dann zeilenweise nummeriert.

Wenn dies abgeschlossen ist, wird `nStateToolDownload` wieder inkrementiert und `der switch` schaltet zum `default case`, in dem der Methoden-Aufruf-Merker `flag_ReceiveToolData` auf `false` gesetzt wird (Zeile 91); somit wird der ganze Code-Block bei den nächsten Zyklen übersprungen, bis wieder die Methode aufgerufen wird.

Die Funktion zum Herunterladen von Werkzeugdaten ist eigentlich ein typischer Fall für die Implementierung als ein OPC UA Programm; man vergleiche auch Seite 118 bzw. Kap. 4.8 in [11].

Bei der Methode Transfer_Tool_Data (auch asynchron) müssen die Werkzeugnummer und alle Parameter beim Methoden-Aufruf eingegeben werden. Diese Daten werden dann vom Server über die entsprechende DNC-Funktion in den NC-Kern übertragen. Auf das Anführen des Codes wird hier verzichtet. Obwohl die Methode Reset_Tool_Data in ToolsFolderType modelliert wurde (siehe Kapitel 3.1.6), wurde sie nicht programmiert, weil ihre Funktion mit Transfer_Tool_Data auch realisiert werden kann.

```
// ========= DOWNLOAD TOOL DATA ============================

if (!bStopDownload ||
    (pObject->getTools_Folder()->flag_ReceiveToolData == true))
{
 switch(nStateToolDownload)
 {
  case 0:
  {
   printf("Downloading tools data...");
   dncResult_t  res2 = turn55->getTools();
   assert(res2 == DNC_TRANSFER_DOWNLOAD);
   nStateToolDownload++;
  }break;
  case 1:
  {
   printf(".");
   dncResult_t  res2 = turn55->getToolsTransferState();
   if (res2 == DNC_OK) {  nStateToolDownload++;  }
   else
   { assert(res2 != DNC_MACHINE_TRANSFER_ABORT);
     ...
   }
  }break;
  case 2:
  {
   std::list<toolEntry_t> lstTool;
   for (int nTool = 0; nTool<=255; nTool++)
   {
    dncResult_t  res2 = turn55->getTool(nTool, lstTool);
    if (res2 == DNC_OK)
    {
     if (!lstTool.empty() && !bToolAlreadyAdded[nTool])
     //means if tool exists on DNC and not already added to OpcUA
     {
      UaStatus Tool_addNodeAndReference_Status;
      printf("\nTool %i exists on DNC",nTool);
      std::list<toolEntry_t>::iterator it = lstTool.begin();
      char tempstr [3];
      unsigned char j = (*it).uToolId;
      // j is the value of current ToolId from DNC =>
      //                                    (*it).uToolId = nTool = j
      // creating the Tool instances for existing Tools from DNC:
      sprintf(tempstr,"%i",j);
      //the next line constructs for every existing tool on DNC an UA
      //object, adds hierarchy in beginning of NodeId
      //and ToolId (from DNC) at the end of NodeId, and gets
      //NodeConfig and SharedMutex from object Tools_Folder
      pTool[j] = new EmcoNS::ToolType(
          UaNodeId(UaString("%1.Tool_%2")
          .arg(pObject->getTools_Folder()->getUaNode()->
```

```
52            nodeId().toString()).arg(UaString(tempstr)), 2),
53            pObject->getTools_Folder()->s_pTool,
54            pObject->getTools_Folder()->getNodeConfig(),
55            pObject->getTools_Folder()->getSharedMutex());
56
57        pTool[j]->setBrowseName(UaQualifiedName(UaString("Tool_%1")
58            .arg(UaString(tempstr),2,UaChar('0')),2));
59
60        pTool[j]->setDisplayName(UaLocalizedText("",UaString("Tool_%1")
61            .arg(UaString(tempstr),2,UaChar('0'))));
62
63        Tool_addNodeAndReference_Status =
64            pObject->getTools_Folder()->getNodeConfig()->
65            addNodeAndReference(pObject->getTools_Folder(),
66            pTool[j], OpcUaId_HasComponent);
67
68        bToolAlreadyAdded[nTool]=1;
69        unsigned char i = 0;
70        while(it != lstTool.end())
71        {
72         if (i==0) pTool[j]->setP00_...((*it).fValue);
73         if (i==1) pTool[j]->setP01_Werkzeugtyp((*it).fValue);
74         ...
75         if (i==25) pTool[j]->setP25_DP_25((*it).fValue);
76         ++it; i++;
77        }
78       }
79      }
80      else
81      {
82        assert(res2 != DNC_MACHINE_TRANSFER_ABORT);
83        ...
84      }
85     }  // end of for (nTool<=255) block
86     nStateToolDownload++;
87    }break; // end of case 2
88    default:
89    {
90     printf("\nTools data download successful.\r\n");
91     pObject->getTools_Folder()->flag_ReceiveToolData = false;
92     nStateToolDownload = 0;
93     bStopDownload = true;
94    }break;
95   } // end of switch block
96  } //end of if(!bStoptransfer)
97  // ===========================================================
```

**Code-Snippet 13: Herunterladen der Werkzeugdaten und dynamische Instanziierung der Werkzeug-Objekte in MainCycle**

# 4 Der adaptierte OPC UA Client

Um die Kommunikation zwischen einem Zellrechner und der Werkzeugmaschine über OPC UA zu veranschaulichen, wurde ein vorhandener OPC UA Client umprogrammiert. Der Client ist im .Net SDK von Unified Automation enthalten als ein Beispiel für OPC UA Client Anwendungen, und ist als eine Windows-Applikation programmiert. Durch Verschieben der vorhandenen Objekte in der Windows-Form wurde Platz geschaffen für eine Bedienoberfläche zur Erstellung einer Ablaufsteuerung, siehe rot umrandeter Bereich in Abbildung 4.1.

Abbildung 4.1 zeigt einen Screenshot dieses Clients, zusammen mit einer Grafcet-Darstellung einer „einprogrammierten“ Sequenz. Diese Abbildung soll die Einfachheit der „Programmierung“ der automatisch abzulaufenden Schritte der Komponenten einer Fertigungszelle veranschaulichen. Es sei angemerkt, dass der Client in dieser Abbildung nur mit dem OPC UA Server der Drehmaschine verbunden ist. Für Abläufe in einer Fertigungszelle müsste er jedoch im Adressraum die Objekte aller Komponenten (z.B. die der Werkzeugmaschine und von einem Roboter) ansprechen können. Der Client müsste also dann mit einem Server der Zelle (statt der Maschine) verbunden werden.

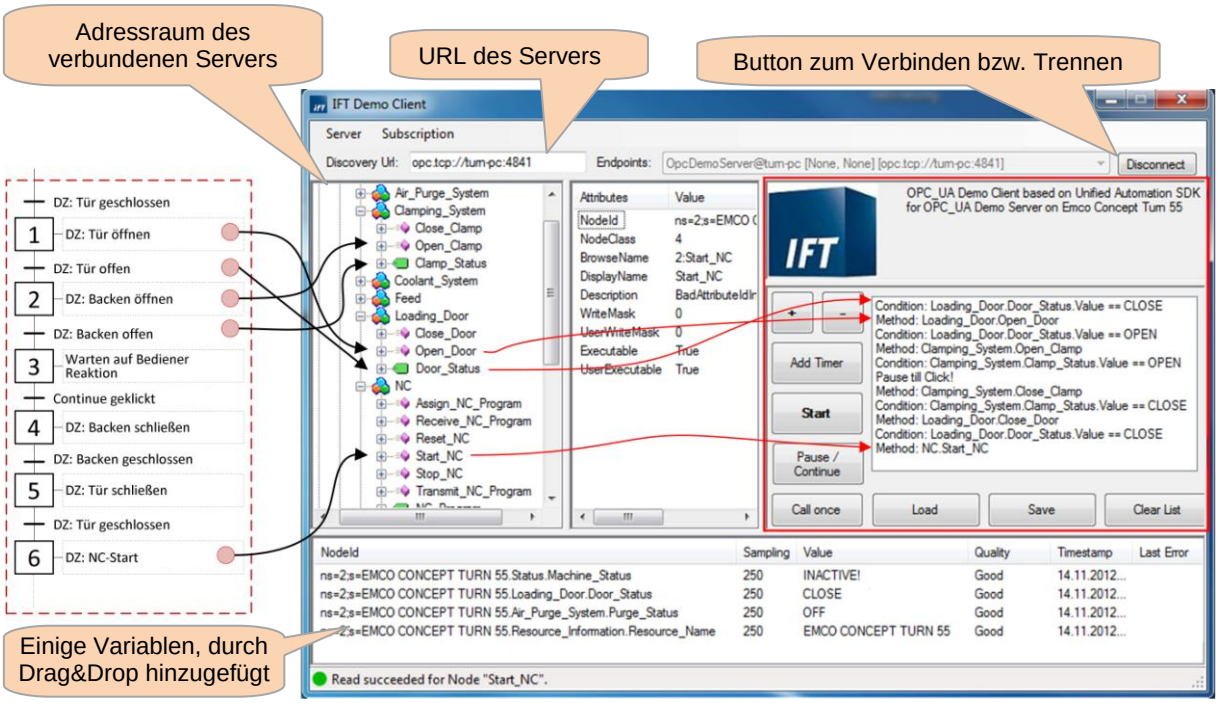


**Abbildung 4.1: Screenshot vom adapteriten OPC UA Client**

Bei Anwählen einer Methode bzw. Variable aus dem Adressraum der Maschine kann diese durch einen Click auf dem „+“-Button in das rechte Listen-Feld aufgenommen werden. Wurde eine Methode in die Liste aufgenommen, wird eine Zeile mit `"Method: "` und die entsprechende „NodeID“ als Text eingetragen. Bei Methoden, die eine Eingabe erfordern, öffnet sich ein kleines Fenster, wo der Bediener den

gewünschten Eingabeparameter eingeben kann (siehe Abbildung 4.2 und Abbildung 4.3). In diesem Fall wird dann am Ende der jeweiligen Zeile noch der String `".InputArgument: "` + der entsprechende Wert hinzugefügt.

Wird hingegen eine Variable in die Liste aufgenommen, wird diese als Übergangsbedingung aufgenommen, die Zeile enthält dann `"Condition: "` + entsprechende NodeID + `".Value == "` + entsprechender Wert gegen ihn die Bedingung überprüft wird. Dieser Wert wird wieder durch ein kleines Dialogfenster angegeben (siehe Abbildung 4.4).

Es besteht auch die Möglichkeit in der Sequenz durch den Button „Add Timer“ eine bestimmte Wartezeit, aber auch eine Pause (z.B. für Interaktion mit einem Bediener) einzubauen (siehe Abbildung 4.5). Ist die Sequenz erstellt, kann sie dann auch als Text-Datei abgespeichert und später wieder geladen werden.

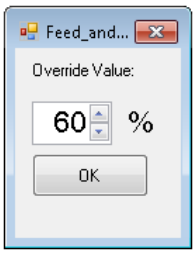


**Abbildung 4.2: Screenshot des Dialogfensters zur Eingabe des gewünschten Feed_Override-Wertes als Schritt in der Sequenz**

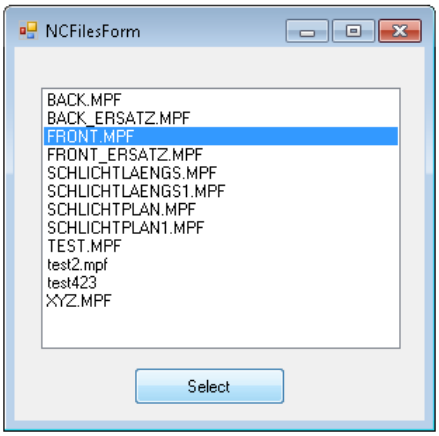


**Abbildung 4.3: Screenshot des Dialogfensters zur Auswahl eines NC_Programs (Methode: Assign_NC_Program) als Schritt in der Sequenz**

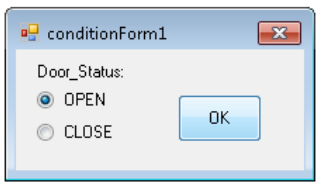


**Abbildung 4.4: Screenshot des Dialogfensters zur Auswahl der gewünschten Übergangsbedingung, hier für die Variable "Door_Status"**

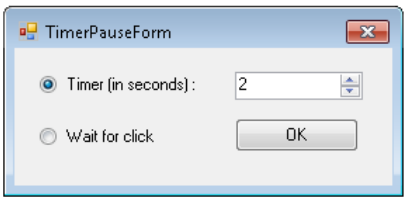


**Abbildung 4.5: Screenshot des Dialogfensters "TimerPauseForm" zum Hinzufügen einer Verzögerung oder Pause in der Sequenz**

Nachdem die Sequenz definiert wurde, kann die automatische Abarbeitung durch den Button „Start“ gestartet werden. So wird Schritt für Schritt jede Zeile aus der Sequenz abgearbeitet. Stellt der Schritt eine Aktion dar, wird die entsprechende OPC UA Methode eventuell mit den eingegebenen Eingabeparametern aufgerufen und der Client geht zum nächsten Schritt über. Ist nun der Schritt eine Übergangsbedingung, dann liest der Client regelmäßig den Wert der entsprechenden OPC UA Variable, bis dieser den gewünschten Wert hat, und geht wiederum zum nächsten Schritt der Sequenz über.

Diese Ablaufsteuerung wurde in sehr kurzer Zeit entwickelt, um einen möglichen Einsatz des OPC UA Servers der Drehmaschine zu demonstrieren, und war ursprünglich nicht Teil der Aufgabenstellung für diese Diplomarbeit. Deshalb stellt dieser OPC UA Client nur einen groben Entwurf dar und ist nicht ausgereift. Es wird deshalb hier darauf verzichtet genauer auf die Programmierung einzugehen. Um die Stärken von OPC UA zu demonstrieren, sollten in OPC UA Applikationen z.B. auch Events und Alarme implementiert werden. So sollten z.B. Übergangsbedingungen durch Anmelden des Clients an OPC UA Events realisiert werden.

# 5 Fazit und Ausblick für zukünftige Entwicklungen

Als Endergebnis wurde eine Drehmaschine des Typs „Emco Concept Turn 55“ mit dem entwickelten OPC UA Server am Institut für Fertigungstechnik und Hochleistungslasertechnik (IFT) ausgestattet. Der entwickelte OPC UA Server wurde auf der Messe „Vienna-Tec 2012“ am Stand vom Institut für Fertigungstechnik und Hochleistungslasertechnik (IFT) präsentiert, und somit der Zeitplan für die Entwicklung eingehalten. Der lauffähige Server wurde für die Drehmaschine „Emco Concept Turn 55“ konfiguriert und lief auch sehr stabil. Im Zusammenhang mit dem adaptierten OPC UA Client wurden auch einfache Abläufe an der Drehmaschine simuliert. Vor allem Mitarbeiter der Entwicklungsabteilung der österreichischen Herstellerfirma der Werkzeugmaschine – EMCO group – zeigten sich sehr interessiert an der Umsetzung von OPC UA auf der „Emco Concept Turn 55“.

Seit der Erstpräsentation auf der Messe „Vienna-Tec 2012“ wurde der Server weiterentwickelt. Beispielsweise wurden exemplarisch einige Objekte der Maschine mit OPC UA Events und „Historical Data Access“ erweitert. Somit ist „der Beweis, dass per Netzwerkkabel zu jedem beliebigen OPC UA Client eine Verbindung hergestellt werden kann, erbracht.“ [19]

Durch die Abhängigkeit vom DNC-Treiber ist jedoch eine Weiterentwicklung nur noch beschränkt möglich. Der Erfolg dieser Entwicklung warf bei Emco und am IFT die Frage auf, ob die Möglichkeit bestünde, auf dem DNC-Treiber als Middleware zu verzichten und ein OPC UA Server direkt auf dem NC-Kern der Maschine aufzusetzen.

Es ist durchaus denkbar, dass in (ferner) Zukunft Maschinen, ihre Komponenten und auch Geräte durch eine semantische Kommunikationssprache wie OPC UA miteinander und natürlich auch mit dem Menschen zweckgemäß kommunizieren könnten.

## 5.1 Vorschläge für zukünftige Entwicklungen

### 5.1.1 Verwendung von OPC UA Programme statt OPC UA Methoden

Zwischenzeitlich wurde auch eine Version des Servers mit synchronen Methodenaufrufen der Automatisierungskomponenten getestet, z.B. für die Methode Open_Door. Dies erwies sich aber als nicht zweckmäßig, weil bis zum vollständigen Ausführen der Aktion, der OPC UA Server blockiert wurde. Für weitere Entwicklungen muss die Verwendung von OPC UA Programmen, spezifiziert im Teil 10 der OPC UA Spezifikation, eruiert werden. OPC UA Programme laufen asynchron ab. Nachdem ein OPC UA Programm gestartet wurde, könnte der Server regelmäßig

den Zustand dieses laufenden Programms beobachten und ein Event generieren wenn es abgeschlossen ist bzw. eine definierte Zeit überschritten hat.

### 5.1.2 Einsatz von Events und „Alarms and Conditions“

Zustandsänderungen der Maschine könnten einem Client durch OPC UA Events übermittelt werden. So könnte auch das erfolgreiche Ausführen einer aufgerufenen asynchronen Methode Client-seitig überprüft werden. Im Laufe dieser Entwicklung wurde auch der Einsatz von Events getestet. Die Implementierung von Events ist beim verwendeten SDK jedoch im Zusammenhang mit dem Modellierungstool nicht sauber gelöst, und bedarf größerer Änderungen in den von UaModeler generierten Code-Abschnitten.

Die in der OPC UA Spezifikation Teil 9 beschriebenen „Alarms and Conditions“ sind spezielle Events, die dem Client Zustandsänderungen übermitteln oder Meldungen zum Quittieren generieren können. Quittierbare Alarme oder Zustandsänderungen könnten eingesetzt werden um eine Interaktion eines Maschinenbedieners aufzufordern z.B. bei einer Fehlermeldung oder wenn kritische Zustände eintreten.

### 5.1.3 Zertifikate und Netzsicherheit

Wegen der Nutzung von Internetstandards und die mögliche Einbindung eines OPC UA Servers im Internet sollte bei jeder OPC UA Applikation die Netzsicherheit gewährleistet sein. In der OPC UA Teilspezifikation 2 sind die auf Zertifikaten basierenden Sicherheitsprotokolle beschrieben. In der Steuerungs- oder Prozessleitebene, wo die Werkzeugmaschine gesteuert wird, nimmt die Netzsicherheit theoretisch eine untergeordnete Bedeutung an, weil Netzwerke auf diesen Ebenen entweder physisch oder logisch durch Firewalls vom Internet getrennt sind. Jedoch kann ein interner, nicht autorisierter Zugriff auf den OPC UA Server einer Maschine zu Unterbrechungen in der Produktion in der Fabrik oder gar zu Gefahren für MitarbeiterInnen in maschinennähe führen. Deshalb sollten OPC UA Server bei einer Verbindungsanfrage eines Clients stets ein bestimmtes Zertifikat fordern, um die Verbindung nur mit autorisierten Clients bzw. Benutzern zuzulassen.

### 5.1.4 Domänenmodellierung, Entwicklung von generischen Servern

Bei der Informationsmodellierung wurde auch schnell klar, dass der Bedarf eines standardisierten Modells – eines Domänenmodells – für die Domäne der flexiblen Fertigungszellen besteht. Künftige Entwicklungen sollten dies berücksichtigen und sich vorrangig damit beschäftigen erst ein sogenannten Companion-Standard für diese Domäne zu entwickeln. Im Falle von Werkzeugmaschinen arbeitet zurzeit das „MTConnect Institut“ an der Veröffentlichung der „MTConnect-OPC UA Companion Specification“, die schon als „Release Candidate“ MTConnect-Mitgliedern zur

Verfügung steht. MTConnect ist ein Standard für die Abfrage von Maschinendaten bei NC-Maschinen. Der „Companion Specification“ soll ein Mapping zwischen MTConnect und OPC UA ermöglichen.

Einige Werkzeugmaschinenhersteller bzw. NC-Steuerungsentwickler bieten schon Classic OPC zum Lesen von Maschinendaten an. Die Implementierung von OPC UA läuft aber schleppend voran. Grund dafür könnte der zusätzliche Entwicklungsaufwand sein, oder gar die geringe Nachfrage. Ein Mapping von bestehenden Classic OPC Applikationen auf OPC UA scheint auch nicht zielführend, weil die vielen Vorteile von UA so nicht umgesetzt werden können.

Die Entwicklung eines allgemeinen OPC UA Servers für Werkzeugmaschinen könnte den Entwicklungsaufwand reduzieren. Hersteller müssten dann für bestehende Maschinen einen Treiber – ähnlich wie hier der DNC-Treiber – anbieten, der Informationen aus ihren Maschinen den Knoten des wohl definierten allgemeinen Informationsmodells zuweist.

Denkbar ist das folgende Szenario:

Die Automatisierungskomponenten von Maschinen (z.B. die Arbeitsraumtür) werden als OPC UA Objekttypen definiert. Im Sinne der Objektorientiertheit müssten alle Komponenten einer Maschine von definierten OPC UA Objekttypen instanziiert werden, je nachdem welche Komponenten in einem bestimmten Maschinentyp eingebaut werden. Genauso wie das physische Zusammenbauen von Komponenten zu einer Maschine, werden diese Komponenten-Instanzen zum „Zusammenbauen“ des Objekttyps der Maschine (Maschinentyp) verwendet. Da nun Objekttypen einem global definierten (Internet-weiten) Namensraum in Form eines URIs zugewiesen sein können, könnte jeder Objekttyp weltweit eindeutig definiert werden. So, dass bei Inbetriebnahme einer Maschine ausgestattet mit einem generischen OPC UA Server, dieser sich mit einem OPC UA Server der Herstellerfirma verbindet und für diese bestimmte Maschine alle Informationen in Form von OPC UA Objekttypen bezieht.

# 6 Literaturverzeichnis


[1] Iman Ayatollahi, Burkhard Kittl, Florian Pauker, und Martin Hackhofer, „Prototype OPC UA Server for Remote Control of Machine Tools“, in *Proceedings of International Conference on Innovative Technologies*, Budapest, 2013, S. 73–76.

[2] N. N., „Sinumerik System 8, DNC-Schnittstelle, Beschreibung“. Siemens AG, 1983.

[3] N. N., „Motion Control Information System, SINUMERIK 840D/840Di/810D, Rechnerkopplung RPC SINUMERIK, Funktionshandbuch“. Siemens AG, 2005.

[4] International Organization for Standardization, *ISO/IEC 9506-2 : Industrial Automation Systems : Manufacturing Message Specification. Part 2, Protocol Specification*. ISO, 1990.

[5] P. J. Leitão, J. M. Machado, und J. R. Lopes, „A Manufacturing Cell Integration Solution“, in *Proceedings of 2nd IEEE/ECLA/IFIP International Conference on Architectures and Design Methods for Balance Automation Systems,* Lisboa, 1996.

[6] Verband Deutscher Maschinen und Anlagenbau e.V, „VDMA Einheitsblatt 34180, Datenschnittstelle für automatisierte Fertigungssysteme“, Juli 2011.

[7] Promotionsgruppe Kommunikation der Forschungsunion Wirtschaft - Wissenschaft, „Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0“, 2013. [Online]. Verfügbar unter: http://www.bmbf.de/pubRD/Umsetzungsempfehlungen_Industrie4_0.pdf. [Zugegriffen: 17-Feb-2014].

[8] Florian Pauker und Christian Kühlmayer, „Plug & Produce , Aufbau einer automatisierten Fertigungszelle für die Dreh- und Fräsbearbeitung“. 26-März-2012.

[9] N. N., „Beschreibung der DNC Schnittstelle (Binär Modus)“. Emco TEE Softwareentwicklung, 2009.

[10] Moser, Benjamin, „Dokumentation, DNC Schnittstellenadapter“. 16-Dez-2011.

[11] W. Mahnke, S.-H. Leitner, und M. Damm, *OPC Unified Architecture*, 1. Aufl. Springer Verlag, 2009.

[12] „About OPC - What is OPC?“ [Online]. Verfügbar unter: http://opcfoundation.org/Default.aspx/01_about/01_whatis.asp?MID=AboutOPC. [Zugegriffen: 11-März-2014].

[13] U. Enste und W. Mahnke, „OPC Unified Architecture - The future standard for communication and information modeling in automation“, ABB Review 07/2011, ABB, S. 397–404, Juli 2011.

[14] „IEC Webstore | Welcome“, *IEC Webstore*, 06-März-2014. [Online]. Verfügbar unter: http://webstore.iec.ch/. [Zugegriffen: 10-März-2014].

[15] „Deutsches Institut für Normung : Startseite DE“. [Online]. Verfügbar unter: http://www.din.de/cmd?level=tpl-home&contextid=din. [Zugegriffen: 10-März-2014].

[16] DIN Deutsches Institut für Normung e. V., „OPC Unified Architecture – Teil 3: Adressraummodell (IEC 62541-3:2010); Englische Fassung EN 62541-3:2010“. Stand-2012.

[17] „UA Server SDK C++ Bundle: Unified Automation OPC UA SDK API Reference.“ [Online]. Verfügbar unter: http://documentation.unified-automation.com/uasdkcpp/1.3.2/. [Zugegriffen: 07-März-2014].

[18] „Timer-Klasse (System.Timers)“. [Online]. Verfügbar unter: http://msdn.microsoft.com/de-de/library/System.Timers.Timer. [Zugegriffen: 10-März-2014].

[19] D. Pohselt, „Maschinenbau: Die perfekte Werkzeugmaschine ...“, *INDUSTRIEMAGAZIN*. [Online]. Verfügbar unter: http://www.industriemagazin.at/a/maschinenbau-die-perfekte-werkzeugmaschine. [Zugegriffen: 13-März-2014].

# 7 Abbildungsverzeichnis

# 8 Tabellenverzeichnis

# 9 Code-Snippet-Verzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| bzw. | beziehungsweise |
| ca. | circa |
| COM | Component Objekt Model |
| d.h. | das heißt |
| DNC | Direct Numeric Control |
| DCOM | Distributed COM |
| DCS | Distributed Control System |
| ERP | Enterprise Resource Planing |
| etc. | et cetera |
| exkl. | exklusive |
| XML | Extensible Markup Language |
| HTTP | Hypertext Transport Protocol |
| i.d.R | in der Regel |
| inkl. | Inklusive |
| IEC | International Electrotechnical Commission |
| IP | Internet Protocol |
| KMU | Kleine und mittlere Unternehmen |
| lt. | laut |
| MPF | Main Program File |
| MMS | Manufacturing Messaging Specification |
| OPC | OLE for Process Control (siehe auch Fußnote auf Seite 11) |
| OPC UA | OPC Unified Architecture (siehe auch Fußnote auf Seite 11) |
| PLC | Programmable Logic Controller |
| RPC | Remote Procedure Call |
| SDK | Software Development Kit |
| SPS | Speicherprogrammierbare Steuerung |
| SCADA | Supervisory Control and Data Acquisition |
| TR | Technical Report |
| TCP | Transmission Control Protocol |
| et al. | und andere |
| usw. | und so weiter |
| URI | Uniform Resource Identifier |
| u.a. | unter anderem |
| VDMA | Verband Deutscher Maschinen- und Anlagenbau |
| vgl. | vergleiche |
| WZM | Werkzeugmaschine |
| z.B. | zum Beispiel |