Institut für Parallele und Verteilte Systeme

Abteilung Anwendersoftware

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 3244

Optimierung der IST-Analyse und kontinuierliche Überwachung einer realen Produktionsumgebung

Alexander Braunstein

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer:	M. Sc. Jorge Minguez
begonnen am:	20.09.2011
beendet am:	15.03.2012
CR-Klassifikation:	C.1.1, C.2.5, H.2.8, I.2.4

Abstract

Um die Fehleranalyse in der Produktion zu automatisieren, das heißt Produktionsdaten zu analysieren und gleichzeitig den Produktionsablauf zu optimieren, ist die Implementierung eines MAPE-basierten Servicezyklus nötig. In dieser Diplomarbeit wurde der Monitoring-Teil des MAPE-Zyklus implementiert. In den Monitoring-Teil mit einbezogen ist eine Voranalyse der Produktionsdaten. Nach einer Einführung in die verwendeten Technologien wird die Konzeption und Implementierung des Monitoring beschrieben. Ein besonderes Augenmerk wird auf die Gebiete serviceorientierte Architektur, Datenstromverarbeitung, Data-Mining und semantische Daten gelegt.

To automate the fault analysis in production, that is, to analyze production data, while optimizing the production process, the implementation of a MAPE-based service-cycle is necessary. In this thesis, the monitoring part of the MAPE-cycle has been implemented. Included in the monitoring-part is a preliminary analysis of production data. After an introduction to the technologies used, the design and implementation of monitoring is described. Particular attention is paid to the areas of service-oriented architecture, data stream processing, data mining and semantic data.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1 Motivation

Fehleranalysen in der Produktion werden heutzutage immer noch manuell durchgeführt. Dies erfordert viel Zeit und Arbeit, was sich durch eine Automatisierung immens reduzieren würde. Die Benutzung von Standards und modularen Systemarchitekturen ist der Schlüssel zur Erstellung eines adaptiven Informationsmanagements in Produktionsumgebungen. Deshalb geht der Trend immer mehr weg von der traditionellen *hub-and-spoke* Architektur, bei welcher der Informationsfluss über einen zentralen Knoten (HUB – Middelware) [1] gesteuert wird, hin zur *Service Oriented Architecture (SOA)*, die mittels eines *Enterprise Service Bus (ESB)* realisiert werden kann. Dennoch ist diese Technologie nicht ausreichend für die Implementierung eines adaptiven Informationsmanagements. Hierfür ist eine Middleware notwendig, welche in der Lage sein muss einen *Enterprise Application Integration (EAI)* – Prozess auszuführen, die Produktionsdaten zu analysieren und unter Einbeziehung aller notwendigen Services den Produktionsverlauf zu optimieren. Um dies zu erreichen ist die Implementierung eines MAPE[1]-basierten Servicezyklus (Abb. 1) nötig.

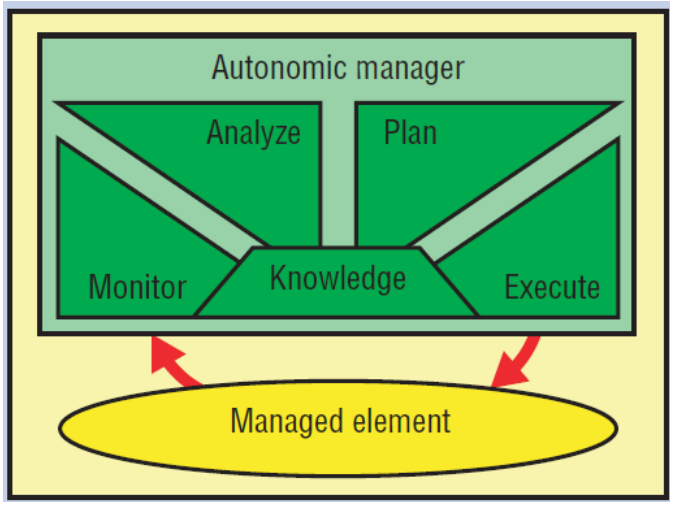

Abbildung 1: MAPE-Zyklus [15]

Ziel dieser Diplomarbeit ist es, durch die Implementierung des Monitoring-, und Voranalyseteils den MAPE-Zyklus zu schließen und ein automatisiertes

1 MAPE steht für: monitoring, analyze, plan, execution

Fehleranalysesystem zu erstellen. Die große Herausforderung zur Erstellung eines solchen Systems, ist die starke Heterogenität der Fabrikdaten in Einklang zu bringen. Als Integrationsmiddleware, zur Ausführung der EAI-Prozesse wird hier ein bereits entwickelter Manufacturing Service Bus (MSB), der den Prinzipien eines ESB folgt, verwendet. Der MSB regelt den Kontrollfluss aller Ereignisse, welche in einer Produktionsumgebung auftreten [2].
Das Monitoring füllt die Lücke zwischen der Produktion und der Analyse. Hierbei werden durch einen Push-Ansatz die Produktionsdaten zeitnah überwacht und an die Datenstromverarbeitung NEXUS DS [5] weitergeleitet. Im Anschluss daran findet innerhalb von NEXUS DS die Analyse der Produktionsdaten, mittels eines Data-Mining-Verfahrens und semantischer Daten, statt. Die Ergebnisse der Produktionsdatenanalyse werden in einem Service Repository zur Verfügung gestellt. Es wird somit versucht die Kontrollflusseigenschaften des MSB gemeinsam mit den Datenstromeigenschaften von NEXUS DS zu verwenden, um high-level Kontext-Informationen direkt in die Planungsphase der Produktion einzubringen. Diese Diplomarbeit wurde in Zusammenarbeit mit dem Institut für industrielle Fertigung und Fabrikbetrieb (IFF) der Universität Stuttgart erstellt. Das IFF hat eine physische Modellfabrik [16] entwickelt, von dessen Leitrechner die nötigen Produktionsdaten für den Analyseprozess ausgelesen werden können. Zusammen mit Zusatzinformationen lässt sich ein realer Produktionsverlauf nachstellen.

1.2 Aufbau der Arbeit

In den folgenden zwei Kapiteln werden alle zur Arbeit relevanten Themen erläutert. Im Kapitel 2 wird zunächst auf die Grundlagen aller verwendeten Technologien eingegangen. In diesem Zuge werden wichtige Technologien wie zum Beispiel die serviceorientierte Architektur, die verwendete Datenstromverarbeitung NEXUS DS sowie verschiedene *Java-Frameworks* erläutert. In den Kapiteln 3 und 4, dem Hauptteil der Diplomarbeit, wird die Konzeption und Implementierung des Monitorings und der Produktionsdatenfehleranalyse erläutert. Abgeschlossen wird die Diplomarbeit mit der Darstellung der Ergebnisse, einer Diskussion, sowie einer Zusammenfassung und einem Ausblick auf Zukünftiges im entsprechenden Forschungsgebiet.

2. Grundlagen der verwendeten Technologien

In diesem Kapitel werden die Grundlagen der Technologien beschrieben, welche bei der Implementierung des Fehleranalysesystems verwendet wurden.

2.1 Serviceorientierte Architektur (SOA)

„SOA ist ein Systemarchitektur-Konzept, das die Bereitstellung fachlicher Dienste und Funktionalitäten in Form von Services vorsieht. Ein Service ist in diesem Kontext eine Funktionalität, die über eine standardisierte Schnittstelle in Anspruch genommen werden kann [3].“

Bei der SOA steht nicht mehr die Applikation im Mittelpunkt, sondern das Hauptaugenmerk ist auf die Geschäftsprozesse gerichtet. Das wesentlichste Merkmal der Architektur ist die lose Kopplung der Komponenten. Dadurch wird eine Unabhängigkeit von den Implementierungsdetails der Dienste erreicht. Des weiteren ist eine dynamische Einbindung der Services während der Laufzeit problemlos möglich. Dies bietet einen hohen Grad an Individualität und Flexibilität [4].

Eine SOA kann durch einen Web Service realisiert werden. Grundsätzlich sind bei einer serviceorientierten Architektur drei Komponenten beteiligt (Abb. 2): Ein Verzeichnisdienst, das sogenannte Repository, ein Client (Konsument) und ein Server (Anbieter des Dienstes). Der Web Service kann über das Internet mittels HTTP, FTP oder SMTP aufgerufen werden. Jeder Web Service beschreibt seine Funktionalität über eine Schnittstellenbeschreibung mit der *Web Service Description Language (WSDL)*. Die Kommunikation zwischen Client und Server wird durch das standardisierte Nachrichtenprotokoll *Simple Object Access Protocol (SOAP)* realisiert [3]. SOAP basiert auf der *Extensible Markup Language (XML)*. Zudem kann SOAP an verschiedene Transportmechanismen gebunden werden. Weit verbreitet ist die Bindung an das *Hypertext Transfer Protocol (HTTP)*. Hier stellt der Client eine Anfrage an den Server, welcher eine entsprechende Antwort an den Client zurücksendet. Dies stellt eine synchrone Kommunikation dar. In vielen Anwendungen, wie auch bei dieser Diplomarbeit, ist jedoch eine asynchrone Kommunikation notwendig. Für diese Kommunikationsart eignet sich der *Java Messaging Service (JMS)* besonders gut. Der JMS erfüllt die wichtigsten Prinzipien einer serviceorientierten Architektur:

- Lose Kopplung der Komponenten
- Plattformunabhängige Komponenten
- Dynamische Einbindung der Komponenten

Im Kapitel 2.6 wird der *Java Messaging Service* detailliert beschrieben.

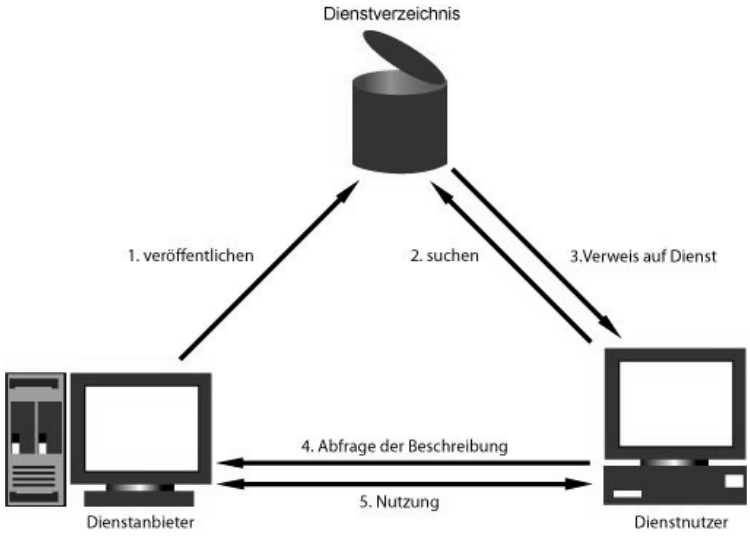

Abbildung 2: SOA-Dreieck. Hauptbeteiligte an einer Serviceorientierten Architektur [36]

2.2 Enterprise Service Bus (ESB)

Mit dem *Enterprise Service Bus* ist es möglich, die Dienste eines Unternehmens in eine Anwendungslandschaft mit Hilfe von Standards kosteneffizient und agil zu integrieren [17]. Zu den Kernaufgaben eines ESB gehören vor allem die Nachrichten-, und Netzwerkprotokolltransformation, das Routing der Nachrichten, sowie die lose Kopplung der Dienste. David A. Chappell definiert einen ESB wie folgt:

„An Enterprise Service Bus is a standard-based integration platform that combines messaging, web services, data transformation and intelligent routing in a highly distributed, event-driven Service Oriented Architecture [17].“

Auf dem Markt gibt es einige kommerzielle sowie freie Implementierungen eines *Enterprise Service Bus*. Um beispielhaft jeweils ein Produkt zu nennen: *JBoss ESB* [18] (freie Implementierung) oder von IBM *WebSphere Enterprise Service Bus* [19] (kommerzielle Implementierung).

2.2.1 Manufacturing Service Bus (MSB)

In vielen Industriezweigen fehlt immer noch ein Grundgerüst um ein Fabrikinformationssystem aufzubauen. Der MSB, entstanden durch eine Erweiterung des ESB, ist somit auch eine SOA-basierte Integrationsplattform. Der MSB verfügt über Zusatzfunktionalitäten in drei Bereichen: *Event Management*, *Factory Context* und *Change Propagation Workflow* [2]. Da in Fabrikstätten die meisten Nachrichten auf Grund eines Ereignisses entstehen, ist das *Event Management* von großer Bedeutung. Mit dem MSB kann die Lücke zwischen den Event-basierten Produktionsumgebungen und den serviceorientierten Geschäftsprozessen gefüllt werden [2]. Somit eignet sich die Anwendung des Event-basierten, serviceorientierten MSB besonders bei Produktionsunternehmen.

2.3 Data Streaming & Complex Event Processing

In diesem Abschnitt wird auf das Verarbeiten von Datenströmen eingegangen. Zudem wird das mit dem *Data Streaming* eng verbundene *Complex Event Processing* erläutert und ein Vergleich zwischen *Data Streaming* und *Complex Event Processing* gegeben.

2.3.1 Datenströme

Nach der Definition ist ein Datenstrom eine kontinuierlich übersandte Menge an Datensätze [26]. Aufgrund der Größe, der Menge und der zeitlich schnellen Abfolge, werden die Datensätze vor ihrer Bearbeitung nicht gespeichert [26].
Vergleicht man Datenströme mit den statischen Daten aus einer Datenbank, so kann man statische Daten als Tupel von Werten ansehen, welche als begrenzte Anzahl in einer Datenbank gespeichert werden. Datenströme hingegen sind zeitlich geordnete Daten, die in unbegrenzter Menge auftreten können [32]. Die Tabelle 1 stellt Datenstromsysteme den Datenbanksystemen gegenüber und beschreibt die Unterschiede. In Abbildung 3 wird die Verarbeitung von Daten in einem Datenstrom und von statischen Daten in einer Datenbank erläutert. Aus den Vergleichen lässt sich erkennen, dass die Anfragen an Datenströme die Anfragen an eine Datenbank (z.B. *Data Warehouse*) ergänzen. Durch die Filterung der Datensätze innerhalb von Datenströmen besteht die Möglichkeit, mit Hilfe von Datenströmen eine Datenbank zu füllen.

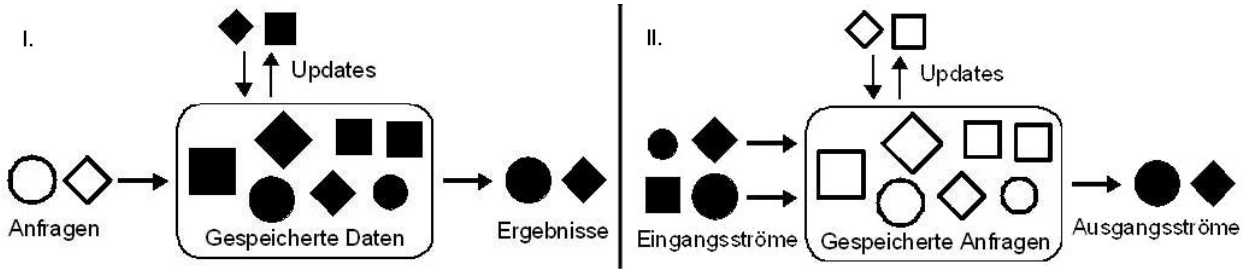

Abbildung 3: I.: Verarbeitung von statischen Daten in einer Datenbank, II.: Verarbeitung von Daten in einem Datenstrom [32]

Im Allgemeinen lassen sich zwei Arten von Datenströmen unterscheiden: Transaktionsdatenströme und Messdatenströme. Unter Transaktionsdatenströme versteht man kontinuierlich gesendete Aufzeichnungen, sogenannte Log-Daten. Messdatenströme stammen beispielsweise von Sensorwerten und finden ihre Anwendung unter anderem in der Meteorologie. Durch die Verwendung von Datenströmen wurde eine neuartige PUSH-Kommunikation möglich. Im Gegensatz zur PULL-Kommunikation, bei der ein Client nach neuen Daten anfragt, werden bei der PUSH-Kommunikation die Daten gesammelt, gefiltert und an einen bestimmten Adressaten gesendet. Die Filterung der Datenströme geschieht durch eine dauerhafte Überwachung, bei der nach bestimmten Mustern innerhalb des Datenstroms gesucht wird. Dabei werden verschiedenartige Anfragen an den Datenstrom gestellt. Analysiert man einen Datenstrom, so werden aggregierte Werte aus dem Datenstrom ermittelt. Es könnte beispielsweise eine Früherkennung für Wetterereignisse realisiert werden. Des weiteren hat man die Möglichkeit die Datensätze innerhalb eines Datenstroms mit Metadaten zu versehen. Dies bietet die Möglichkeit einer schnelleren Überwachung und Analyse des Datenstroms. Zum Beispiel könnte man anhand der Metadaten erkennen, ob dieser Datensatz für das aktuelle Vorhaben relevant ist oder nicht [26].

	Datenstromsysteme	**Datenbanksysteme**
Daten	flüchtig	dauerhaft
Anfragen	dauerhaft	flüchtig
Änderungen	(meist) auf Hinzufügen beschränkt	beliebig
Ergebnisse	eventuell angenähert	exakt
Datenzugriff	möglichst Einpass-Verfahren	beliebig
Indizierung	der Anfragen	der Daten

Tabelle 1: Vergleich: Datenstrom- und Datenbanksysteme [26]

Im folgenden Abschnitt wird die in dieser Diplomarbeit verwendete Datenstromverarbeitung NEXUS DS beschrieben.

2.3.2 Datenstromsystem NEXUS DS

NEXUS DS ist eine flexible und erweiterbare Middleware für lokale sowie verteilte Datenstromverarbeitungen, welche als Erweiterung des ursprünglichen Systems (NEXUS) für kontextsensitive Anwendungen hervorgeht [5]. Durch die Komposition von vordefinierten, standardisierten oder auch benutzerdefinierten Operatoren zu einem Ausführungsgraph wird eine Datenstromverarbeitung implementiert. Generell befinden sich in einem Datenflussgraph drei verschiedenartige Operatoren (Abb. 4).

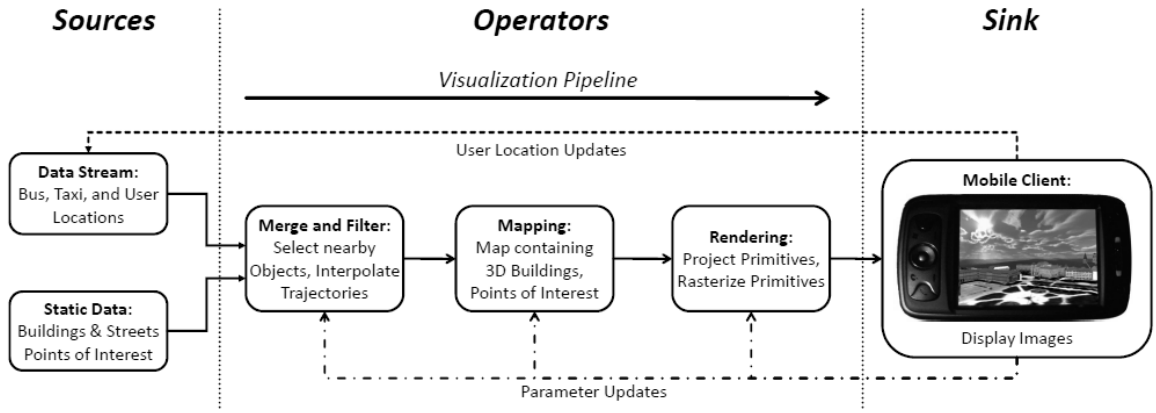

Abbildung 4: Operator Graph of an Interactive and Location-Aware Visualization Pipeline for Mobile ClientDevices [5]

Es existiert immer ein Quelloperator (*Source*), welcher für die Selektion der Rohdaten verantwortlich ist. Diese Rohdaten werden an einen oder mehrere Verarbeitungsoperatoren weitergeleitet. Das Ergebnis der Verarbeitungsoperatoren wird schließlich an einen Senkenoperator (*Sink*) weitergeleitet, der das Ende der Datenstromverarbeitung darstellt. Die Konfiguration des Ausführungsgraphs lässt sich bequem über einen *GUI-Editor* vornehmen. Anschließend lässt sich der Ausführungsgraph als XML-basiertes *AWML-File* exportieren. Durch die *Augmented World Modeling Language (AWML)* wird die *Augmented World* beschrieben. Eine *Augmented World* ist ein Modell der realen Welt, das reale und virtuelle Objekte enthält [38]. Aus dem AWML-Schema ist der Aufbau eines generischen NEXUS-Objektes zu erkennen (Abb. 5).

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.nexus.uni-stuttgart.de/2.0/AWML"
   xmlns:nsas="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
   xmlns="http://www.w3.org/2001/XMLSchema" xmlns:awml="http://www.nexus.uni-
   stuttgart.de/2.0/AWML" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
   <annotation>
      <documentation>This is the schema for AWML (Augmented World Modeling
         Language). It defines, how a generic nexusobject is build. It
         does /not/ define, which elements are allowed in which nexusobject
         (since XML is not capable of that). For this information, see the
         Nexus Standard Class Schema (nscs)
      </documentation>
   </annotation>
   <import namespace="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
      schemaLocation="NexusStandardAttributeSchema.xsd"/>
   <element name="awml">
      <complexType>
         <sequence>
            <element name="nexusobject" minOccurs="0" maxOccurs="unbounded">
               <complexType>
                  <sequence>
                     <element ref="nsas:NexusAttribute" minOccurs="0"
                        maxOccurs="unbounded">
                           <annotation>
                              <documentation>unordered list of arbitrary
                                 Nexus attributes
                              </documentation>
                           </annotation>
                     </element>
                     <element name="reverseRefs"
                        type="nsas:NexusReverseReferencesType" minOccurs="0">
                        <annotation>
                           <documentation>a list of reverse references. This
                              is only a preview how reverse references could
                              look like in the next version of AWML!
                           </documentation>
                        </annotation>
                     </element>
                  </sequence>
               </complexType>
            </element>
         </sequence>
      </complexType>
   </element>
</schema>
```

Abbildung 5: AWML-Schema. Datei: NexusAwmlSchema.xsd

Durch die Implementierung der Datenstromverarbeitung als Graph (zusammengesetzt aus den Komponenten bzw. Operatoren), lässt sich die Flexibilität und Erweiterbarkeit von NEXUS DS verdeutlichen. Somit ist ein Austauschen bzw. Hinzufügen einzelner

Operatoren – ohne Veränderung der kompletten Datenstromverarbeitung – problemlos möglich. Das Hinzufügen eines Operators ist sogar während der Laufzeit durchführbar. Aus dem Grundgerüst von Quell-, Verarbeitungs-, und Senkenoperator lassen sich die beiden wesentlichen Schritte einer Datenstromverarbeitung herauslesen:

1. Selektion der Rohdaten
2. Verarbeitung der Daten durch eine klar definierte Menge an Operatoren

Da bei dieser Arbeit die vom Leitrechner kommenden Produktionsdaten (Quelldaten) einen möglichen, unendlichen Datenfluss darstellen, ist zur Bearbeitung der Daten eine Datenstromverarbeitung nötig [5]. Die Hauptgründe, warum hier die Datenstromverarbeitung NEXUS DS verwendet wurde, liegen auf der Hand:

1. *Erweiterbarkeit der Operatoren:* Durch die Eigenschaften von NEXUS DS ist es möglich semantische Annotationen zu den Operatoren hinzuzufügen. Zudem ist ein ausgezeichnetes Monitoring der Produktionsdaten durch die Implementierung neuer, benutzerdefinierter Operatoren realisierbar.
2. *Verwendung von strukturierten und unstrukturierten Daten:* Auf Grund der großen Datenheterogenität in der Produktion ist es wichtig jegliche Datentypen für die Stromverarbeitung verwenden zu können. Mit NEXUS DS lassen sich für die unterschiedlichen Fabrikdatenflüsse die jeweiligen Operatoren implementieren.
3. *Abstraktion der Laufzeiteinschränkungen:* Bestimmte Operatoren erfordern spezielle Hardwareanforderungen. Um die Datenstromverarbeitung von den Hardwareanforderungen zu entkoppeln, besitzt jeder Operator Metadaten, welche die Hardwareanforderungen festlegen. Somit muss man sich beispielsweise für Operatoren mit Echtzeitanforderungen keine Gedanken über die erforderliche Hardware machen.

2.3.3 Complex Event Processing

Um das *Complex Event Processing (CEP)* zu verstehen, ist zunächst eine Definition eines Events notwendig. Nach David Luckman [27] ist ein Event folgendermaßen definiert:

„An event is an object that is a record of an activity in a system. The event signifies the activity. An event may be related to other events."

Der Definition zur Folge besitzt ein Event eine

- Form: Ein Event ist eine Art Objekt, welches Informationen zum Ort des Auftretens, zum Entstehungszeitpunkt oder auch zum Verursacher besitzt. Solch ein Event könnte als String oder Klasse repräsentiert werden.
- Bedeutung: Die Bedeutung eines Events entspricht der Aktivität. Informationen wie diese Aktivität konkret aussieht ist meistens in der Form enthalten.
- Bedingtheit: Die Bedingtheit beschreibt die Abhängigkeit verschiedener Aktivitäten [29].

Ein komplexes Event entsteht aus mehreren einfachen Events, welche gleichzeitig in einer bestimmten Reihenfolge oder in einem zeitlichen Abstand auftreten [29]. Gemäß dem *Event Processing Glossary* [28] wird ein *Complex Event* wie folgt definiert:

„Complex Event: An event that is an abstraction of other events called its members."

Ein Beispiel für ein *Complex Event* aus dem Alltag wäre:

Wenn in einem Zimmer die Raumtemperatur steigt (Event 1) und kurz danach der Rauchmelder aktiviert wird (Event 2), so kann man annehmen, dass es in diesem Raum brennt (*Complex Event*).

Werden komplexe Events in irgendeiner Art und Weise verarbeitet, so spricht man von *Complex Event Processing*. Hierfür gibt es wieder eine Definition gemäß dem *Event Processing Glossary* [28]:

„Computing that performs operations on complex events, including reading, creating, transforming, or abstracting them."

Im Allgemeinen kann man sagen, dass es sich bei *Complex Event Processing* um eine systematische und automatische Verarbeitung von Datenströmen handelt. Den prinzipiellen Ablauf des CEP zeigt Abbildung 6. Die Events werden zunächst nach zeitlichen und kausalen Bedingungen gefiltert. Anschließend werden mehrere einfache Events zu einem komplexen Event zusammengefügt. Im nächsten Schritt wird das komplexe Event, abhängig von dessen Inhalt, weitergeleitet. Letztlich wird durch das Event ein Alarm ausgelöst bzw. ein Service aufgerufen. Große Anwendung findet das CEP in drei Bereichen [30]:

- *Business Activity Monitoring*: Hier gilt es anhand der Geschäftsprozesse frühzeitig Veränderungen zu erkennen und dementsprechend darauf zu reagieren.
- *Sensorik*: Beispielsweise senden Sensor-Netzwerke Messdaten, welche zur Überwachung einer Maschine nötig sind. Um die Messfehler zu reduzieren werden Messdaten verschiedener Sensoren kombiniert.
- *Marktanalyse*: Aktienkurse können ebenfalls als Ereignisse aufgefasst werden. Somit kann durch die Verarbeitung der Ereignisse eine Trendanalyse durchgeführt werden.

Es werden zwei Arten von CEP unterscheiden. Es ist zu schauen, ob die komplexen Ereignisse als bereits bekannte Muster der Eventströme erkannt werden, oder ob unbekannte Muster in den Eventströmen als komplexes Ereignis spezifiziert werden. Im ersten Fall geschieht dies durch speziell entwickelte Anfragesprachen für Eventströme. Bei unbekannten Mustern werden Techniken wie *Data-Mining* oder maschinelles Lernen eingesetzt [30].

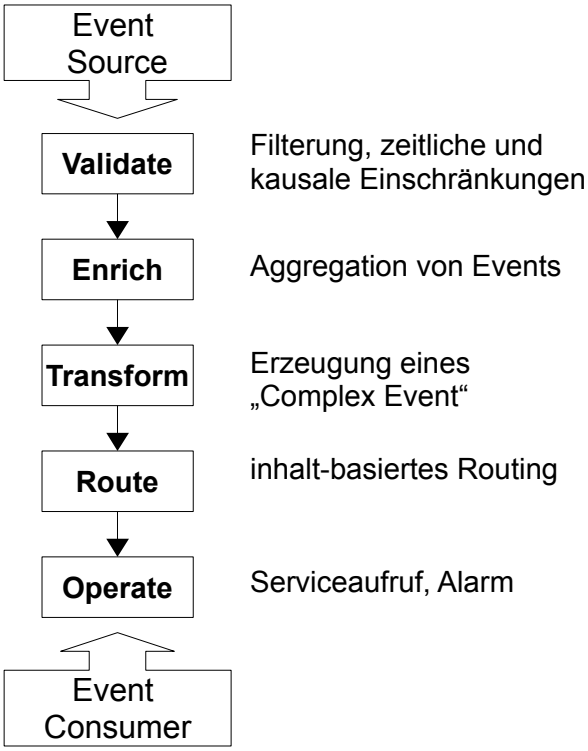

Abbildung 6: Complex Event Processing (CEP) [31]

2.4 Lernfabrik

Ein Forschungsprojekt des Instituts für industrielle Fertigung und Fabrikbetrieb (IFF) der Universität Stuttgart ist der Aufbau einer Lernfabrik (Abb. 7) für *Advanced Industrial Engineering*, welches von Prof. Westkämper (Universität Stuttgart) wie folgt definiert wurde:

„Das Industrial Engineering der Zukunft dient der schnellen und effizienten Anpassung der Produktionsstrukturen der Unternehmen in allen Ebenen der Planung und operiert zielorientiert in kurz-, mittel- und langfristigen Zeiträumen. Das Industrial Engineering ist in die Prozesse der Planung und Gestaltung sowohl strategisch als auch operativ eingebunden [6].“

Somit bietet die Erweiterung des *Industrial Engineering* die Möglichkeit einer ständigen Optimierung der aktuellen Fabrikkonfiguration. Mit Hilfe der Lernfabrik sollen die Erkenntnisse der Wandlungsfähigkeit einer Fabrik und die dafür erforderlichen Grundlagen geschult werden.

Diese physische Lernfabrik des IFF, zusammen mit dem Leitrechner der Produktionsstraße, ist Grundlage und Testumgebung für das in dieser Diplomarbeit entwickelte Fehleranalyseverfahren.

Abbildung 7: Physische Lernfabrik des IFF an der Universität Stuttgart

2.4.1 Leitrechner

Der Leitrechner ist die Steuerungseinheit der Lernfabrik. Hier werden unter anderem in einer Datenbank alle relevanten Informationen rund um die Produktion gespeichert. Für das Fehleranalyseverfahren sind folgende Datenbankinformationen notwendig:

- Tabelle *dbo.Log*: In der Log-Tabelle der Leitrechnerdatenbank befinden sich die meisten, relevanten Informationen. Im Folgenden werden die Informationen der einzelnen Tabellenspalten erklärt:

  TaskNo: Name des Auftrages
  OperationNo: ID einer Operation innerhalb des Produktionsverlaufs
  ModuleName: Name der bearbeitenden Station
  Status: Hier sind nur Datensätze mit dem Wert 2 relevant.
  QuantityOK: Anzahl der korrekt produzierten Stücke
  TimeStamp: Zeitstempel der Datensätze

- Tabelle *dbo.TaskList:* Aus der Auftragstabelle werden folgende Informationen benötigt:

  Quantity: Gesamtanzahl der Produkte bzgl. des Auftrags
  ProcessName: Produktvariante bzgl. des Auftrags
  StartScheduled: Startzeitpunkt des Auftrags

- Tabelle *dbo.TempProcess:* Aus dieser Tabelle wird die Prozessreihenfolge bzgl. der Produktvariante ausgelesen.

- Tabelle *dbo.ModuleProperty*: In der Spalte *repairInstruct* wird vermerkt, ob an dieser Station Reparaturanweisungen angezeigt werden können.

Der Leitrechner basiert auf dem relationalen Datenbankmanagementsystem von Microsoft „SQL Server“.

2.5 Java Database Connectivity (JDBC)

Um aus Java heraus eine Verbindung zum MSSQL Server aufzubauen wird die Datenbankschnittstelle JDBC verwendet. JDBC hat die Aufgabe eine Datenbankverbindung herzustellen und die Ergebnisse von SQL Anfragen in Java brauchbar zu machen. Im hier entwickelten Projekt wurde der Microsoft SQL Server JDBC Driver 3.0[2] benutzt. Dies ist ein Treiber vom JDBC-Typ 4. Eine Definition der verschiedenen JDBC-Typen findet man auf der Website von IBM [20].

2.6 Java Messaging Service (JMS)

Der JMS ist ein Nachrichtensystem, welches eine Kommunikation zwischen Sender und Empfänger ermöglicht. Da der Empfänger auch antworten kann, ist ein Rollentausch zwischen Sender und Empfänger möglich. Mit dem JMS findet eine asynchrone Nachrichtenübertragung statt. Das heißt, Sender und Empfänger müssen nicht ständig verfügbar sein. Versendete bzw. noch nicht abgeholte Nachrichten werden vom JMS-Server verwaltet. Der JMS-Server befindet sich somit zwischen Sender und Empfänger und dient als Vermittler bzw. sogenannte *Message Oriented Middleware (MOM)* (Abb. 8).

2 http://www.microsoft.com/download/en/details.aspx?id=21599

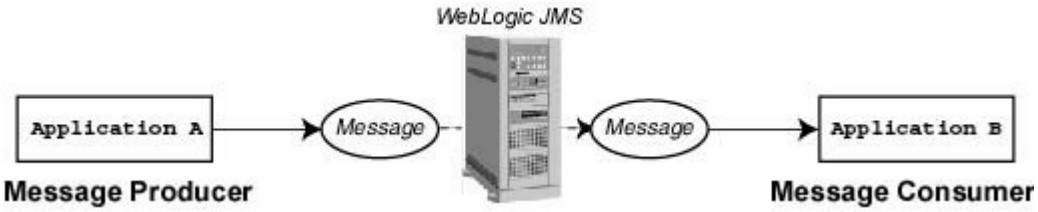

Abbildung 8: Java Messaging Service

Beim Java Messaging Service gibt es grundsätzlich zwei verschiedene Nachrichtenkonzepte: Die 1-zu-1–Übertragung und die 1-zu-n–Übertragung.
Bei der 1-zu-1–Übertragung gibt es genau einen Empfänger, genau einen Sender und genau einen Nachrichtenkanal. Die Nachrichten werden innerhalb der *JMS-Middleware* in einer Warteschlange nach dem FIFO-Prinzip (first in first out) gespeichert. Die Nachrichten bleiben während ihrer Lebenszeit oder bis zur Abholung durch den Empfänger in der Warteschlange gespeichert. Sind mehrere Empfänger für diese Warteschlange registriert, erhält derjenige die Nachricht, der sie als erstes abholt.
Die 1-zu-n–Übertragung wird auch *Topic* genannt. Hier veröffentlicht der Sender eine Nachricht, welche für mehrere Empfänger bestimmt ist. Alle registrierten Empfänger erhalten diese für sie bestimmte Nachricht. Auch bei diesem Nachrichtenkonzept ist die Nachricht während ihrer Lebenszeit gültig.
Nach dem erfolgreichen Konfigurieren des *JMS-Servers*, müssen folgende Schritte durchgeführt werden um eine Nachricht über JMS zu versenden:

1. Eine Verbindung zum Server herstellen (Erstellen eines *InitialContext*)
2. Suchen einer *Connection Factory* (*LookUp Connection Factory*)
3. Erstellen einer Verbindung (*Connection*)
4. Session anlegen
5. Sender anlegen
6. Empfänger anlegen
7. Nachricht anlegen

Diese Schritte finden sich im Quellcode des *Source-Operators* wieder (Kapitel 4.2).

Die bekanntesten Vertreiber von *JMS-Middlewares* sind Glassfish [21] und JBoss [22]. In dieser Arbeit wurde der JMS von JBoss mit dem Nachrichtenkonzept der 1-zu-1 – Übertragung verwendet [7].

2.7 Data-Mining

Data-Mining ist die Entdeckung von versteckten, bisher noch unbekannten Informationen einer großen Menge von Daten [8]. Es liefert neues Wissen für ein besseres Verständnis der Daten. Die Daten werden ohne Erwartungen bezüglich des Ergebnisses analysiert. Das *Data-Mining* lässt sich gemäß Abbildung 9 in den Prozess der Wissensentdeckung einordnen.

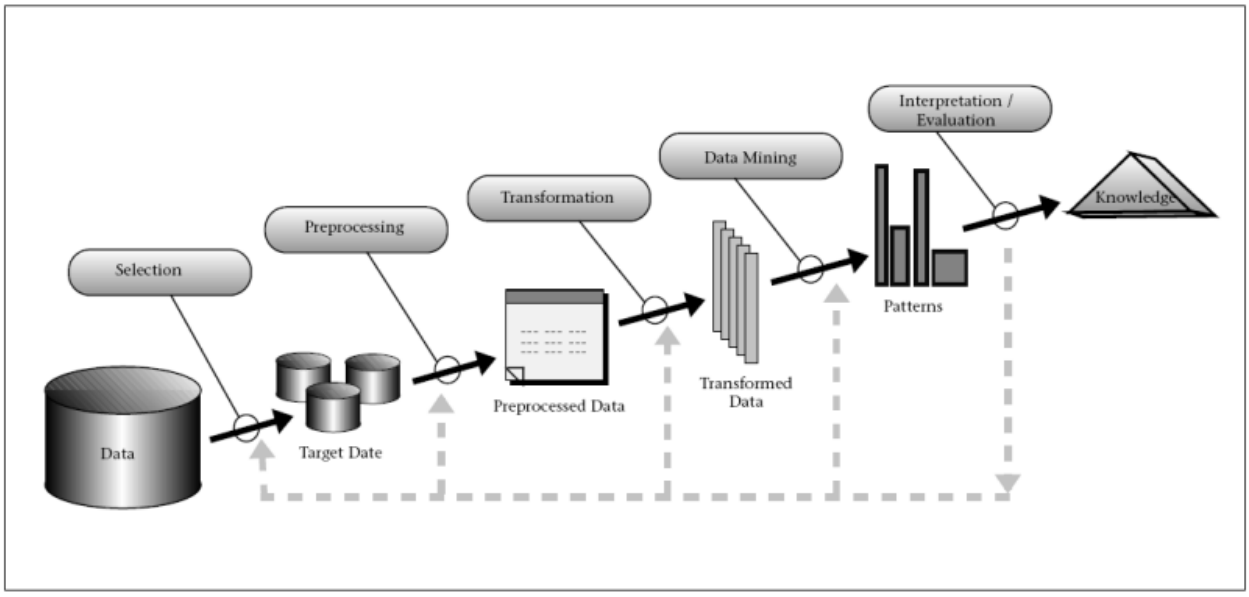

Abbildung 9: Steps that compose the KDD process - Fayyad 1996 [37]

Die *Data-Mining-Verfahren* lassen sich in zwei Kategorien einteilen: Es gibt deskriptive (beschreibende) und prediktive (vorhersagende) Verfahren. Eine der bekanntesten, deskriptiven *Data-Mining-Techniken* ist das *Association Rule Discovery-Verfahren (ARD)*, welches oft bei Warenkorbanalysen zum Einsatz kommt. Hierbei werden Regeln der Form „Wenn Nüsse und Bier gekauft werden, dann werden mit einer bestimmten Häufigkeit und Wahrscheinlichkeit auch Kartoffelchips gekauft.“ aus den Daten erkannt. Neben dem ARD-Verfahren gibt es im Bereich der deskriptiven Techniken das *Clustering-Verfahren*. Hier werden auf Grund eines gegebenen Daten-Sets Cluster

gebildet. Die Eigenschaften der Cluster müssen den folgenden zwei Regeln entsprechen:

1. Objekte innerhalb eines Clusters besitzen eine starke Ähnlichkeit / Homogenität
2. Objekte verschiedener Cluster besitzen eine starke Heterogenität

Ein Vertreter der prediktiven Techniken ist die Klassifikation [8].

2.7.1 Klassifikation

Im Gegensatz zum *Clustering*, bei dem die Cluster zu Beginn nicht vorgegeben sind, sondern durch den Algorithmus entstehen, sind bei der Klassifikation die Klassen (vergleichbar mit Cluster) vorgegeben. Die Datensätze werden bei der Klassifikation anhand ihrer Attributwerte in die verschiedenen Klassen eingeordnet. Generell gliedert sich die Klassifikation in drei Phasen (Abb. 10): Die *Training-Phase*, *Test-Phase* und *Application-Phase*.

Bei der *Test-Phase* wird anhand gegebener Daten, inklusive Klassenzugehörigkeit der einzelnen Datensätze, durch einen Klassifikationsalgorithmus ein Klassifikationsmodell entwickelt. In der anschließenden *Test-Phase* wird die Korrektheit des Modells mit Hilfe von Testdaten überprüft.

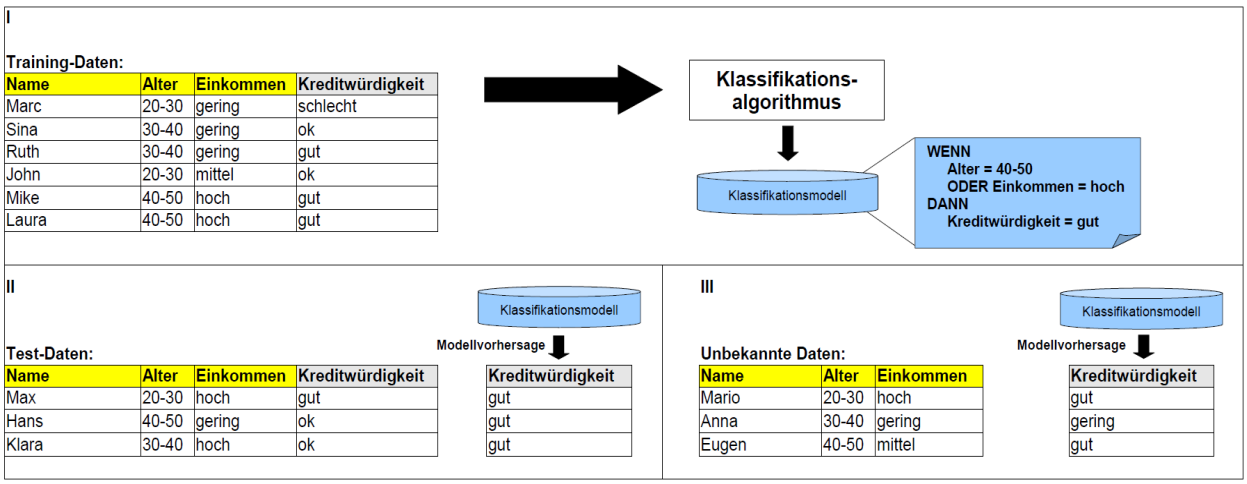

Abbildung 10: Klassifikationsalgorithmus. I: Training-Phase, II: Test-Phase, III: Application-Phase [8]

Im dritten Schritt, der *Application-Phase*, werden die unbekannten Datensätze mit Hilfe des Klassifikationsmodells einer Klasse zugeordnet [8].
In dieser Arbeit wurde das Klassifikationsverfahren leicht abgeändert. Es gibt keinen Algorithmus, welcher ein Klassifikationsmodell erzeugt. Das Modell ist bereits vor Beginn der Klassifikation gegeben. Alle Datensätze werden anhand des vorgegebenen Modells einer Klasse zugeordnet. Somit ist die in dieser Diplomarbeit verwendete Klassifikation ein deskriptives Verfahren, da keine Vorhersagen getroffen werden.

2.8 Semantische Daten

Die Verwendung und Bedeutung von semantischen Daten lässt sich am besten anhand des *Semantic Web* erläutern. Die Idee ist, dass die bereits vorhandenen Daten des *World Wide Web* mit semantischen Daten angereichert werden.
Angenommen man möchte sich über den Begriff SOAP im Sinne der Technologiespezifikation informieren, so gibt man in einer Suchmaschine seiner Wahl den Begriff SOAP ein. Man wird feststellen, dass es schwierig ist die passenden Informationen zu finden. Es werden Informationen zu Seifen, Geschirrspülmittel oder gar Seifenopern angezeigt. Nach langem Durchklicken wird man irgendwann auch die gesuchte SOAP-Spezifikation des W3C[3] finden. Die zahlreichen, irrelevanten Informationen erscheinen, da Computer die Informationen lediglich bereitstellen, jedoch nicht gut genug verstehen können. Lässt man jedoch einen *Semantic Web Agenten* nach dem Begriff SOAP (mit Augenmerk auf SOAP als Technologiespezifikation) suchen, so werden ausschließlich relevante Suchergebnisse aufgelistet. Zudem würde man noch Informationen über WSDL und XML bekommen, da diese in direktem Zusammenhang mit SOAP als Technologiespezifikation stehen. Ziel des *Semantic Web* ist es, dass die Maschinen Daten lesen, verarbeiten, transformieren und sogar auf den Daten operieren können. Um solch ein *Semantic Web* zu implementieren, werden Daten benötigt, welche die Daten beschreiben. Das sind sogenannte Metadaten. Somit kann eine Maschine die Daten anhand der semantischen Metadaten sinnvoll verarbeiten. Aufgabe ist es, möglichst viele semantische Informationen mit den Daten zu verknüpfen, damit ein Computer Rückschlüsse ziehen kann. Das heißt, er kann die Daten verstehen und in Zusammenhang mit anderen Daten bringen. Der Voreiter zur

3 http://www.w3.org/

Erstellung von Metadaten ist die *eXtensible Markup Language (XML)*, bei der durch beschreibende Tags die Daten eine Semantik bekommen [9].

2.8.1 Resource Description Definition (RDF)

RDF, eine Spezifikation von W3C, dient zur semantischen Beschreibung der Daten im Internet. RDF basiert auf der Technologie von XML und URI (*Unified Resource Identifier*). Eine Ressource, welche durch eine URI zu identifizieren ist, wird durch ein sogenanntes RDF-Tripel (Abb. 11) beschrieben. Das Tripel besteht aus einem Subjekt (Ressource), einem Prädikat (Eigenschaft) und einem Objekt (Eigenschaftswert) [9]. Zum Beispiel würde man folgenden Satz derartig beschreiben: „Mitarbeiter 'Max Mustermann' arbeitet an der Handarbeitsstation ML1.“

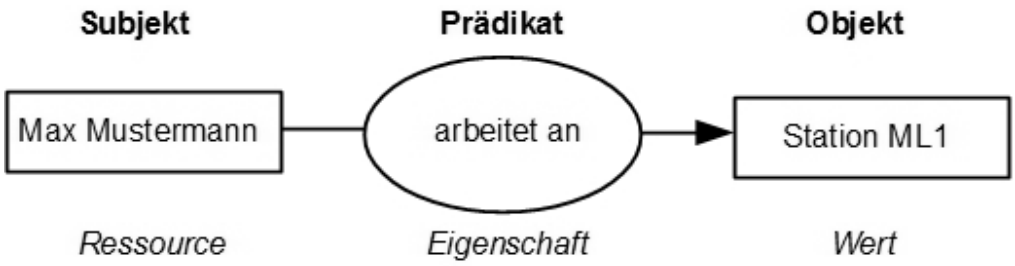

Abbildung 11: RDF-Tripel

Als Serialisierung in RDF/XML würde dies folgendermaßen aussehen (Abb. 12):

```
<?xml version="1.0"?>

<RDF>
  <Description about="http://www.MSB-DA.com/MaxMustermann">
    <arbeitet>Station ML1</arbeitet>
  </Description>
</RDF>
```

Abbildung 12: RDF-Tripel als RDF/XML serialisiert

2.8.2 Resource Description Definition Schema (RDFS)

Um die in RDF formulierten Aussagen zu verstehen, bedarf es einem Vokabular. RDFS ist solch ein Vokabular, mit dem Beziehungen zwischen RDF-Aussagen bzw. Gruppen ähnlicher RDF-Ressourcen beschrieben werden können. Das Vokabular ist jeweils auf

ein bestimmtes Anwendungsgebiet (Domäne) bezogen. Wie auch bei den RDF-Tripels bestehen RDFS-Tripels aus Subjekt, Prädikat und Objekt. Das Subjekt ist beim RDFS eine Klasse. Dem Prädikat und Objekt entsprechen eine Klasseneigenschaft und ein Wert, der die Klassen bzw. die Beziehungen zwischen den Ressourcen beschreibt. Eine Ressource ist beim RDFS eine Instanz der Klasse. Mit RDFS lässt sich eine Hierarchie aufbauen, da jede Klasse eine Unterklasse einer weiteren Klasse sein kann. Deshalb ist es für die Maschinen möglich, die Semantik der Klassen von Ressourcen zu ermitteln. Existieren zudem Regeln für die Verwendung der Ressourcen, innerhalb des Vokabulars, so spricht man von einer Ontologie [9].

2.8.3 Web Ontology Language (OWL)

Aufbauend auf RDFS bietet OWL ein noch viel umfangreicheres Vokabular zur Definition von *Semantic Web Ontologien* an. Eine Ontologie im Zusammenhang mit dem *Semantic Web* ist wie folgt definiert:

„Ein Schema, das die Hierarchien und Beziehungen zwischen verschiedenen Ressourcen ausdrücklich definiert. Semantic Web-Ontologien bestehen aus einer Taxonomie (Klassifikationssystem der Ressourcen) und einer Reihe von Inferenzregeln, anhand derer Maschinen logische Schlüsse ziehen können [9].“

Die wichtigsten Erweiterungen von OWL gegenüber RDFS sind [10]:

- Relationale Charakterisierung von Eigenschaften: transitiv, invers, funktional, symmetrisch, usw.
- Bildung neuer Klassen durch mengenlogische Operationen, wie zum Beispiel: Schnitt -, Vereinigungsmengen oder auch Kardinalitäten.
- Erweitertes *Reasoning*, um Schlussfolgerungen aus den Daten zu ziehen.

OWL lässt sich nach dem Komplexitätsgrad in drei Klassen einteilen [11].

	Anwendung	Komplexität	Komplexitätsklasse
OWL Lite ⊂ OWL DL	– Klassifikationshierarchien mit einfachen Nebenbedingungen – Kardinalitäten (von 0 bis 1)	niedrig	EXPTIME[4]
OWL DL ⊂ OWL Full	– maximale Ausdrucksstärke – Alle Schlüsse in endlicher Zeit berechenbar – beschränkte Benutzung aller OWL Sprachkonstrukte	mittlere	NEXPTIME[5]
OWL Full	– maximale Ausdrucksstärke mit unbeschränkter Nutzung aller OWL Sprachkonstrukte – keine Garantie für die Berechenbarkeit – unwahrscheinlich alle Schlüsse der Ontologie zu erkennen	hohe	unentscheidbar

Tabelle 2: Untersprachen von OWL.

2.9 JENA Semantic Web Framework

Das JENA-Framework [23] ist eine Ansammlung von JAVA-Klassen, die es ermöglichen Semantic Web-Anwendungen zu implementieren. RDF-API, OWL-API, ARQ und die Inference-API sind die Features von JENA. In der RDF-API dient die Klasse `Model` als Grundlage für einen RDF-Graph. Mit der Anwendungsschnittstelle RDF sind Basisoperationen, wie Hinzufügen von Ressourcen, Eigenschaften bzw. RDF-Tripels möglich. Zudem besteht die Möglichkeit RDF-Daten als RDF/XML oder N-Tripels zu lesen und zu schreiben. N-Tripels ist eine weitere Darstellungsform für RDF-Daten. Es gibt noch weitere Operationen, zum Beispiel Mengenoperationen auf das Modell, sowie eine Navigation durch den RDF-Graph. Als Erweiterung der RDF-API geht die OWL-API hervor. Mit dieser ist es möglich auf Basis eines `OntModel` neue Ontologien aufzubauen oder bereits erstellte und geladene Ontologien zu modifizieren. Jedes Modell besitzt ein Profil, welches die Ontologiesprache (siehe Tabelle 2) festlegt. Durch die Anwendungsschnittstelle OWL werden alle Elemente (Klassen, Individuen, usw.) zur Verfügung gestellt. Mit ARQ ist es möglich Anfragen mittels der Anfragesprache

4 Entscheidungsprobleme der Klasse EXPTIME können von einer deterministischen Turingmaschine in beschränkter Zeit akzeptiert werden.
5 Entscheidungsprobleme der Klasse NEXPTIME können von einer nichtdeterministischen Turingmaschine in beschränkter Zeit akzeptiert werden.

SPARQL an das Modell (RDF-Daten) zu stellen. Darauf wird hier nicht detailliert eingegangen, da diese Anwendungsschnittstelle nicht in der Arbeit verwendet wurde. Die Inference-API ist eine Schnittstelle für einen *Reasoner*, welcher aus logischen Aussagen Schlüsse ziehen kann. Ein *Reasoner* findet mit Hilfe von logischen Ableitungsregeln Informationen, welche in dieser Form nicht in der Ontologie stehen [12].

2.9.1 JENA-Rules

Der *Jena-Rules-Reasoner* implementiert einen *Reasoner* von RDFS und OWL. Somit ist es möglich Regeln zu schreiben, welche sowohl auf RDFS-Daten als auch auf OWL-Daten arbeiten. Die Regeln werden üblicherweise in einer Textdatei formuliert. Der Aufbau dieser Datei besteht aus einem Kopf, der durch die Direktive „@prefix“ den *Namespace* der verwendeten Regelelemente festlegt. Im Anschluss daran stehen die eigentlichen Regeln. Der Aufbau der Regeln ist sehr stark an die Beschreibung von RDF-Daten angelehnt und setzt sich auch aus Tripeln (Subjekt – Prädikat – Objekt) zusammen. Neben den Tripeln gibt es noch sogenannte *Built-Ins*[6]. Dies sind prozedurale Primitive, wie zum Beispiel boolesche Prädikate (*equal (?a,?b)*) oder mathematische Operatoren (*add (?a,?b,?c) entspricht ?a + ?b = ?c*). Variablen werden in den Jena-Rules immer durch ein vorangestelltes „?“ gekennzeichnet. Es gibt drei verschiedene Arten von Regeln [13]:

1. *Forward-Rule* (Abb. 13): Diese Regeln erkennt man durch den Pfeil nach rechts „->“. Der Teil links vom Pfeil entspricht der Voraussetzung (Wenn) und der Teil rechts vom Pfeil entspricht der Folgerung (Dann). Beide Teile der Regel, Voraussetzung und Folgerung, dürfen eine Konjunktion von Atomen, sprich Tripels und *Built-Ins,* enthalten.

```
@prefix pre: <http://jena.hpl.hp.com/prefix#>.
[rule1:(?a pre:father ?b)(?c pre:brother ?a)->(?c pre:uncle ?b)]
```

Abbildung 13: Forward-Rule: Wenn "a" der Vater von "b" und "c" der Bruder von "a" ist, dann ist "c" der Onkel von "b".

6 http://jena.sourceforge.net/inference/#RULEbuiltins

2. *Backward-Rule* (Abb. 14): Die *Backward-Rule* lässt sich am Linkspfeil „<-“ erkennen. Diese Art von Regel kann man auch als Anfrage interpretieren. Der Voraussetzungsteil und Folgerungsteil sind umgekehrt als bei der *Forward-Rule*. Bei der *Backward-Rule* darf der Folgerungsteil genau ein Tripel besitzen. Es sind keine *Built-Ins* im Folgerungsteil erlaubt.

```
@prefix ex: http://example.com/school#
[rule2: (?s rdf:type ex:fleißigerStudent) <-
             (?s rdf:type ex:Student) (?s ex:erreichtePkt ?p)
              greaterThan(?p,60)
]
```

Abbildung 14: Backward-Rule unter Verwendung einer Built-In Primitive: Ein Student ist fleißig, wenn er mehr als 60 Punkte erreicht hat.

3. *Hybrid-Rule* (Abb. 15): Bei dieser Art von Regel werden sowohl *Forward-Rules* als auch *Backward-Rules* verwendet. Zum Beispiel lässt sich die Regel aus Abbildung 14 auch als *Hybrid-Rule* ausdrücken.

```
@prefix ex: http://example.com/school#
[rule3: (?s rdf:type ex:Student)
             ->
            [rule4: (?s rdf:type ex:fleißigerStudent)
               <-
               (?s ex:erreichtePkt ?p)
               greaterThan(?p,60)
            ]
 ]
```

Abbildung 15: Hybrid-Rule: Ein Student ist fleißig, wenn er mehr als 60 Punkte erreicht hat.

Aus der Struktur der *JENA-Rule-Inference-Engine* (Abb. 16) lässt sich der grundsätzliche Ablauf zur Ausführung eines Regel-Sets erkennen.

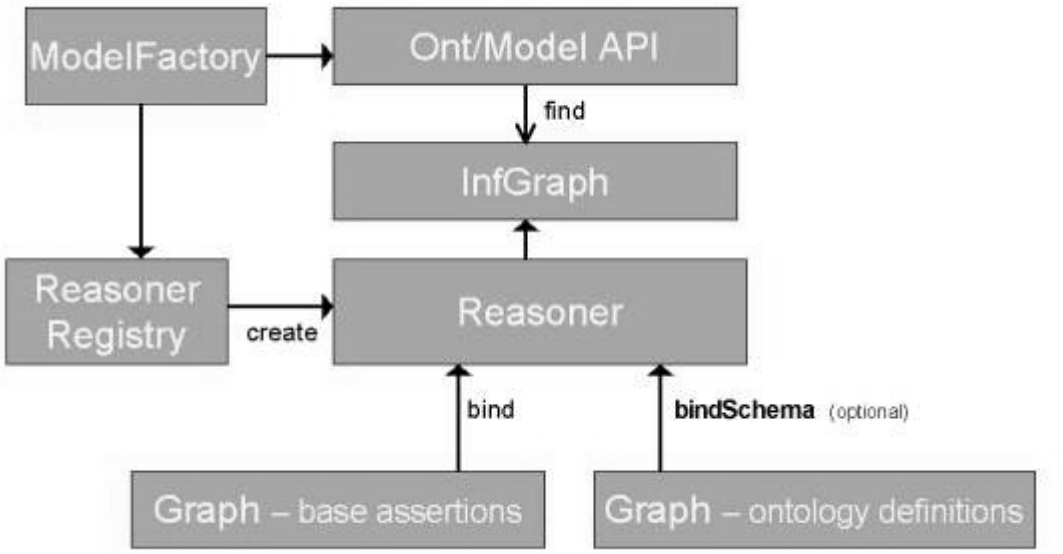

Abbildung 16: Struktur der JENA-Rule-Inference-Engine [14]

Über die *ModelFactory* wird zunächst ein leeres Standardmodell `model` erzeugt. An dieses Modell wird eine Ressource gebunden, die zur Konfiguration dient. Der Konfigurationsressource werden Eigenschaften angefügt, zum einen der Modus der Regeln (*forward*, *backward* oder *hybrid*) und zum anderen das Rule-Set. Über die *ReasonerRegistry* wird eine Instanz des *Reasoners* erzeugt. Nun werden noch die Testdaten für die Ausführung des Regel-Sets benötigt. Die Testdaten werden in das zu Beginn erstellte Standardmodell geladen. Schließlich wird mit der Instanz des *Reasoners* und dem `model` ein Inferenzmodell erzeugt, auf dem das Regel-Set ausgeführt wird [14].

3. Konzeption

In diesem Kapitel wird die Konzeption für das in der Arbeit entwickelte Fehleranalysesystem beschrieben. Im Kapitel 4 wird erläutert, wie das Konzept konkret implementiert wurde.

Um ein automatisiertes Fehleranalysesystem in der Produktion zu entwickeln, muss der MAPE-Zyklus geschlossen werden. Die Aufgabe dieser Arbeit ist es, den Monitoring-Teil des Zyklus zu implementieren.

Die Hauptanalyse der Produktionsdaten findet im *Provenance-aware Service Repository (PASR)* statt. Im PASR werden gebietsspezifische Informationen (*Domain Knowledge* – Abb. 17) als RDF-Tripels abgespeichert. Durch eine speziell entwickelte Sprache *Service Provenance Query Language (SPQL)* können Anfragen auf die Daten im PASR verarbeitet werden [24].

Lediglich die Vorfehleranalyse der Produktionsdaten ist in den Monitoring-Teil integriert.

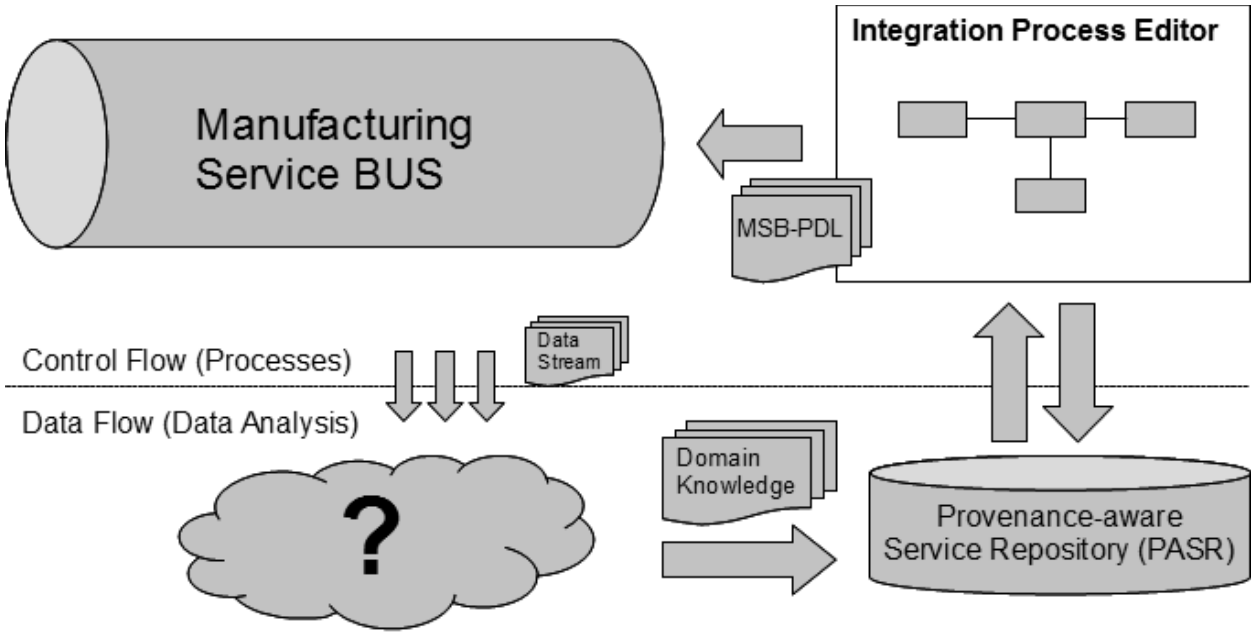

Abbildung 17: Manufacturing Service Bus, Service-Repository und Integration Process Editor

Die in Abbildung 17 dargestellte Wolke steht symbolisch für den Monitoring-Teil des MAPE-Zyklus. Das Monitoring wird mit Hilfe eines SQL-Triggers und dem *Java*

Messaging Service realisiert. Die nachfolgende Vorfehleranalyse findet in der Datenstromverarbeitung NEXUS DS statt. Die Daten vom Leitrechner der Produktion, welcher an den *Manufacturing Service Bus* angeschlossen ist, können durch drei verschiedene Ansätze abgerufen werden (Abbildung 18).

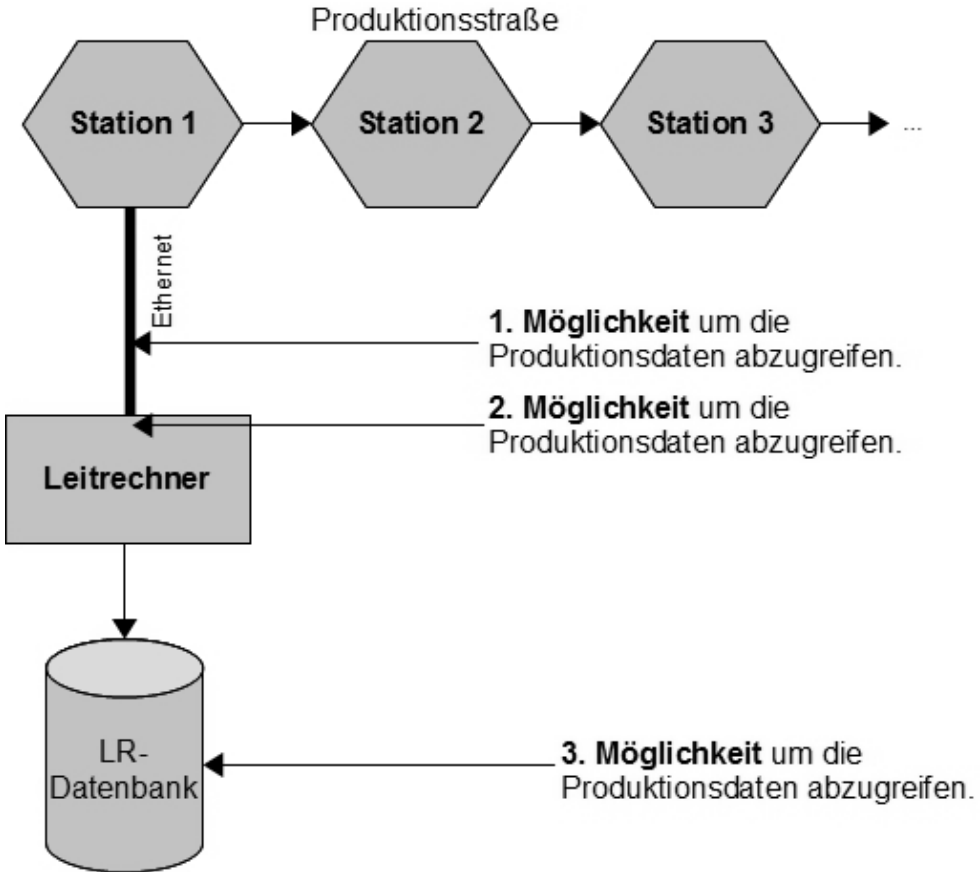

Abbildung 18: Möglichkeiten zum Abgriff der Produktionsdaten

Bei der ersten Möglichkeit werden die Daten direkt aus dem Ethernet-Protokoll gelesen. Eine zeitnahe Weiterleitung der Daten ist bei dieser Methode möglich. Neben der aufwendigen Umsetzung dieser Variante, liegen zudem die Produktionsdaten unformatiert vor.

Greift man die Daten nach der Serialisierung im Leitrechner ab (2. Möglichkeit), erhält man bereits vorformatierte, etwas brauchbarere Daten für die Fehleranalyse. Da die Zeit zwischen der Ankunft der Produktionsdaten im Leitrechner und der Speicherung der Daten in der Leitrechnerdatenbank vernachlässigbar klein ist, wurde in dieser Arbeit

die dritte Möglichkeit zum Abgriff der Produktionsdaten gewählt. Da die Zeit für die Fehleranalyse deutlich größer ist als die Zeit, welche benötigt wird um die Produktionsdaten in der Leitrechnerdatenbank abzuspeichern, kann man immer noch von einer zeitnahen Weiterleitung der Daten reden. Ein weiterer Vorteil der dritten Variante ist, dass mittels eines *SQL-Triggers* die formatierten Produktionsdaten direkt (über einen *JMS-Server*) an die Datenstromverarbeitung weitergeleitet werden können. Im Folgenden werden vier Ansätze erläutert, welche zur Weiterleitung der Produktionsdaten an die Datenstromverarbeitung NEXUS DS verwendet werden können. Alle Ansätze erfüllen die Prinzipien einer serviceorientierten Architektur (Kapitel 2.1).

1. *Push-Ansatz:*

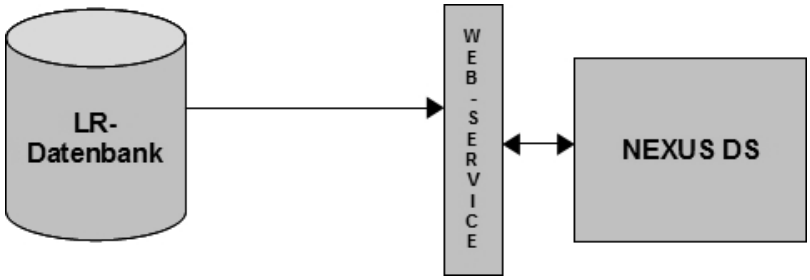

Abbildung 19: Push-Ansatz

Dieser Ansatz kann hier nicht verwendet werden, da bei einer Nichtverfügbarkeit des Web-Services die komplette Leitrechnerdatenbank stehen würde.

2. *Pull-Ansatz:*

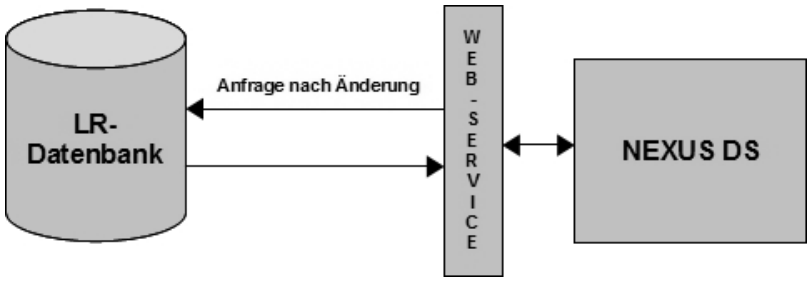

Abbildung 20: Pull-Ansatz

Bei diesem Ansatz wäre die zeitnahe Weiterleitung der Log-Daten nicht mehr erfüllt.

3. *JMS-Ansatz:*

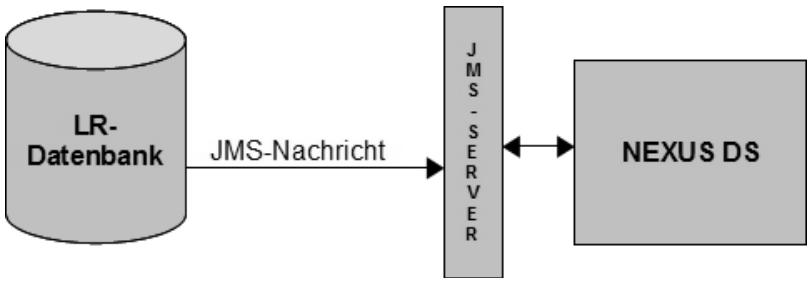

Abbildung 21: JMS-Ansatz

Dieser Ansatz ist zwar etwas langsamer als der Push-Ansatz, arbeitet jedoch immer noch zeitnah. Aufgrund der asynchronen Kommunikation kommt es bei einer Nichterreichbarkeit des *JMS-Servers* nicht zum Stillstand der Leitrechnerdatenbank. Dieser Ansatz wird verwendet um die Lücke zwischen dem Leitrechner und NEXUS DS zu schließen.

4. *SOAP/HTTP-Ansatz*

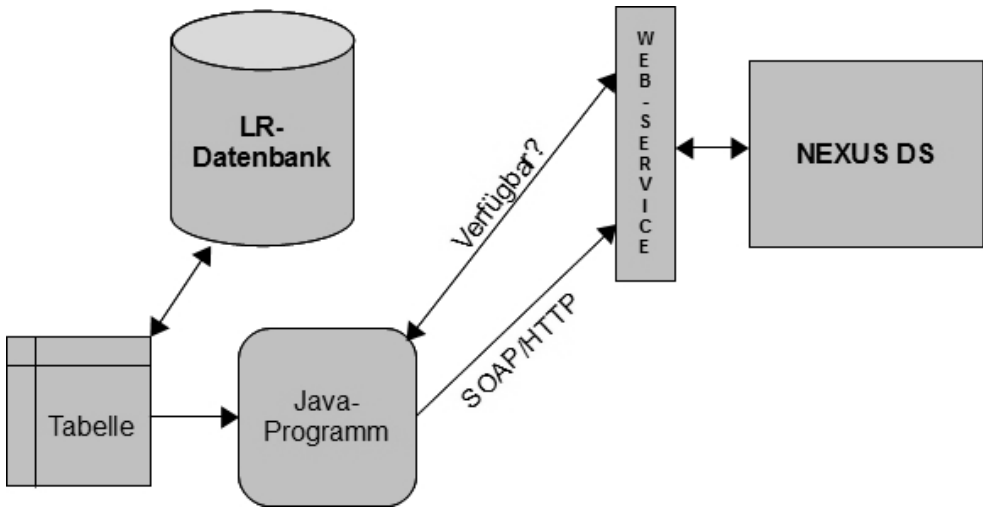

Abbildung 22: SOAP/HTTP-Ansatz

Wegen der hohen Komplexität kommt dieser Ansatz für die hiesige Anwendung ebenfalls nicht in Frage.

Sobald die Produktionsdaten die *Source* der Datenstromverarbeitung NEXUS DS erreicht haben, wird mit Hilfe der Operatoren (Abb. 23) die Fehleranalyse durchgeführt. Mit dem Ergebnis der Fehleranalyse steht der Lösungsvorschlag für das Problem fest.

Dieser muss schließlich an das PASR gesendet werden.

Die Schnittstelle zum PASR wird durch einen Web-Service-Aufruf geschaffen. Das Fragezeichen der Wolke in Abbildung 17 wird durch einen Ausführungsgraph von NEXUS DS ersetzt. Abbildung 23 zeigt den Ausführungsgraph und die Komposition der verwendeten Operatoren für das Monitoring und die Voranalyse der Produktionsdaten.

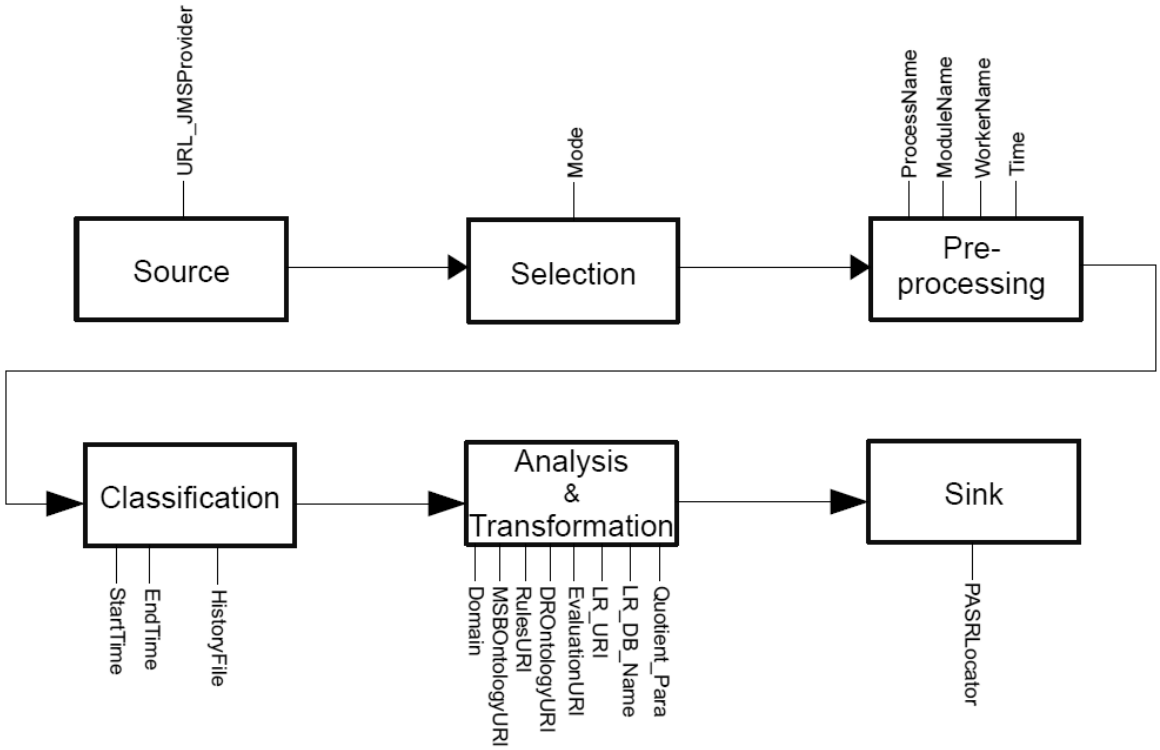

Abbildung 23: NEXUS DS - Ausführungsgraph für den Fehleranalyseprozess

Der mit der GUI erstellte Ausführungsgraph wird in einer AWML-Datei gespeichert. Der Grundaufbau solch einer AWML-Datei beschreibt die Abbildung 24.
Für den Fehleranalyseprozess werden sechs Operatoren benötigt: eine *Source*, vier *Operators* und eine *Sink*. In NEXUS DS ist es möglich bei jedem Operator ein oder mehrere Parameter zu setzen. Beispielsweise ist bei der *Source* im Parameter *URL_JMSProvider* die URL des *JMS-Servers* eingetragen. Die Erklärung der weiteren Parameter sind der Tabelle 3 zu entnehmen.

Parameter	**Erklärung**
URL_JMSProvider	URL des JMS-Provider
Mode	*Mode 0:* Datensätze ausschließlich fehlerhafter Produkte werden weitergeleitet. *Mode 1:* Datensätze ausschließlich korrekter Produkte werden weitergeleitet. *Mode 2:* Datensätze sowohl fehlerhafter als auch korrekter Produkte werden weitergeleitet.
ProcessName	Produktvarianten
ModuleName	Stationen der Produktionsstraße
WorkerName	Mitarbeiter an den Stationen der Produktionsstraße
Time	Zeitstempel – Schicht
HistoryFile	beinhaltet Log-Datensätze der Vergangenheit
MSBOntologyURI	OWL-Datei der MSB-Ontologie
RulesURI	Regeln, welche auf die Ontologie zur Analyse angewandt werden
DROntologyURI	OWL-Datei der Ontologie „Domain Recommendation“
EvaluationURI	CSV-Datei der zusätzlichen Stationsbewertungsdaten
LR_URI	SQL-Server URI des Leitrechners
LR_DB_Name	Name der Leitrechnerdatenbank
Quotient_Para	Faktor, welcher als Schwellwert für den Fehlerquotienten zweier Stationen dient
StartTime	Startzeitpunkt der Analyse
EndTime	Endzeitpunkt der Analyse
Domain	Name des Analyseprozesses
PASRLocator	URL des Provenance-aware Service Repository

Tabelle 3: Erläuterungen der Parameter aller im Ausführungsgraph zur Fehleranalyse verwendeten Operatoren

```
<awml:awml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:gml="http://www.opengis.net/gml" xmlns:nsat="http://www.nexus.uni-
stuttgart.de/1.0/NSAT" xmlns:nsas="http://www.nexus.uni-
stuttgart.de/1.0/NSAS" xmlns:awml="http://www.nexus.uni-
stuttgart.de/2.0/AWML" xmlns:nscs="http://www.nexus.uni-
stuttgart.de/1.0/NSCS">

<awml:nexusobject>
   <eas:serviceFormatInfo xmlns:eas="http://...">
      <nsas:value>
         <eas:namespace xmlns:eas="http://...">http://www.nexus.uni
            -stuttgart.de/1.0/SNSetupDescriptor/ECS
         </eas:namespace>
         <eas:format xmlns:eas="http://...">negm</eas:format>
      </nsas:value>
   </eas:serviceFormatInfo>
   <nsas:type>
      <nsas:value
         xmlns:ecs="http://...">ecs:NexusStreamNodeSetup</nsas:value>
   </nsas:type>
   <eas:block xmlns:eas="http://...">
      <nsas:value>
         <eas:blockType xmlns:eas="http://...">source</eas:blockType>
         <eas:blockID
            xmlns:eas="http://...">XYOperatorisland05B5772C5D2685CBA2FD1E
            52AAE8581D8</eas:blockID>
         <eas:classURI
         xmlns:eas="http://...">urn:java:de.uni_stuttgart.nexus...
         </eas:classURI>
      </nsas:value>
   </eas:block>
   <nsas:kind>
      <nsas:value>real</nsas:value>
   </nsas:kind>
   <nsas:nol>
      <nsas:value>nexus:||/</nsas:value>
   </nsas:nol>
   <eas:commandID xmlns:eas="http://...">
      <nsas:value>island0</nsas:value>
   </eas:commandID>
   <eas:inputManager xmlns:eas="http://...">
      <nsas:value>
         <eas:blockID
            xmlns:eas="http://...">XYOperatorisland05B5772C5D2685CBA2FD1E
            52AAE8581D8</eas:blockID>
         <eas:inputManagerURI
            xmlns:eas="http://...">urn:java:de.uni_stuttgart.nexus...
         </eas:inputManagerURI>
         <eas:inputManagerID xmlns:eas="http://...">
            CountBasedParallelAsyncInputManager36island05B5772C5D2685C
            BA2FD1E52AAE8581D8</eas:inputManagerID>
      </nsas:value>
   </eas:inputManager>
```

```
   <eas:queue xmlns:eas="http://...">
      <nsas:value>
         <eas:queueID xmlns:eas="http://...">
            queue_0_XYOperatorisland05B5772C5D2685CBA2FD1E52AAE8581D8
         </eas:queueID>
         <eas:queueURI xmlns:eas="http://...">
            urn:java:de.uni_stuttgart.nexus...</eas:queueURI>
      </nsas:value>
   </eas:queue>
   <eas:parameter xmlns:eas="http://...">
      <nsas:value>
         <eas:parameterKeyID xmlns:eas="http://...">0</eas:parameterKeyID>
         <eas:blockID xmlns:eas="http://...">
            XYOperatorisland05B5772C5D2685CBA2FD1E52AAE8581D8</eas:blockID>
         <eas:parameterValue xmlns:eas="http://...">
            Parameter-Value</eas:parameterValue>
         <eas:parameterKey xmlns:eas="http://...">
            Parameter-Key</eas:parameterKey>
      </nsas:value>
   </eas:parameter>
   <eas:link xmlns:eas="http://...">
      <nsas:value>
         <eas:inputQueueID xmlns:eas="http://...">
            queue_0_XYOperatorisland05B5772C5D2685CBA2FD1E52AAE8581D8
         </eas:inputQueueID>
         <eas:inputBlockID xmlns:eas="http://...">
            XYOperatorisland05B5772C5D2685CBA2FD1E52AAE8581D8
         </eas:inputBlockID>
         <eas:inputBlockOutputSlotID xmlns:eas="http://...">
            0</eas:inputBlockOutputSlotID>
         <eas:outputBlockInputSlotIDType
            xmlns:eas="http://...">input</eas:outputBlockInputSlotIDType>
         <eas:outputBlockInputSlotID xmlns:eas="http://...">
            0</eas:outputBlockInputSlotID>
         <eas:outputBlockID xmlns:eas="http://...">
            XYOperatorisland05B5772C5D2685CBA2FD1E52AAE8581D8
         </eas:outputBlockID>
      </nsas:value>
   </eas:link>
</awml:nexusobject>

</awml:awml>
```

Abbildung 24: AWML-Datei eines Ausführungsgraphs

Im Folgenden wird der grobe Ablauf des Fehleranalyseprozesses beschrieben. Im Kapitel 4 - Implementierung - wird detaillierter auf die einzelnen Operatoren eingegangen.

Die Log-Daten der Leitrechnerdatenbank erreichen als erstes den *Sourceoperator* der Datenstromverarbeitung. Die Daten werden umgehend an den Operator *Selection* weitergeleitet. Hier werden, in Abhängigkeit des Parameterwertes *Mode,* die Log-Daten selektiert. Für die Fehleranalyse der Produktionsdaten werden ausschließlich

Datensätze fehlerhafter Produkte an den *Pre-Processing-Operator* weitergegeben. Durch ihn werden die Fehlerdaten für den Klassifikationsalgorithmus aufbereitet. Der Operator *Classification* übernimmt die Klassifizierung der Fehlerdaten. Der Benutzer hat die Möglichkeit über ein *History-File* und die Angabe eines Zeitraums auch ältere Log-Datensätze mit in die Klassifizierung einzubeziehen. Das Ergebnis der Klassifizierung wird durch den *Analysis & Transformation-Operator* verarbeitet und in semantische Daten transformiert, die in einer Ontologie (MSB-Ontologie*)* gespeichert werden. Um die Fehlerquelle zu lokalisieren, die Ursache zu ermitteln und eine Fehlerbeschreibung mit Lösungsvorschlag anzugeben, werden Regeln auf die semantischen Daten angewandt. Der Lösungsvorschlag wird an den *Sink-Operator* gesendet, welcher einen Web-Service aufruft und den Lösungsvorschlag in einem Repository speichert.

4. Implementierung

Der erste Schritt der Fehleranalyse besteht darin, durch ein Monitoring die Log-Daten aus der Datenbank des Leitrechners zeitnah zu exportieren. Dies wird mittels eines *Datenbanktriggers* (*Log-Daten-Trigger*) realisiert. Sobald neue Log-Daten in der Datenbank abgelegt werden, reagiert der *Trigger* und sendet diese mittels eines *JMS-Senders* an den *JMS-Server*. Auf dem *JMS-Server* werden die Log-Daten in einer Warteschlange, bis zur Abholung durch den *JMS-Receiver,* gespeichert (siehe Abb. 25).

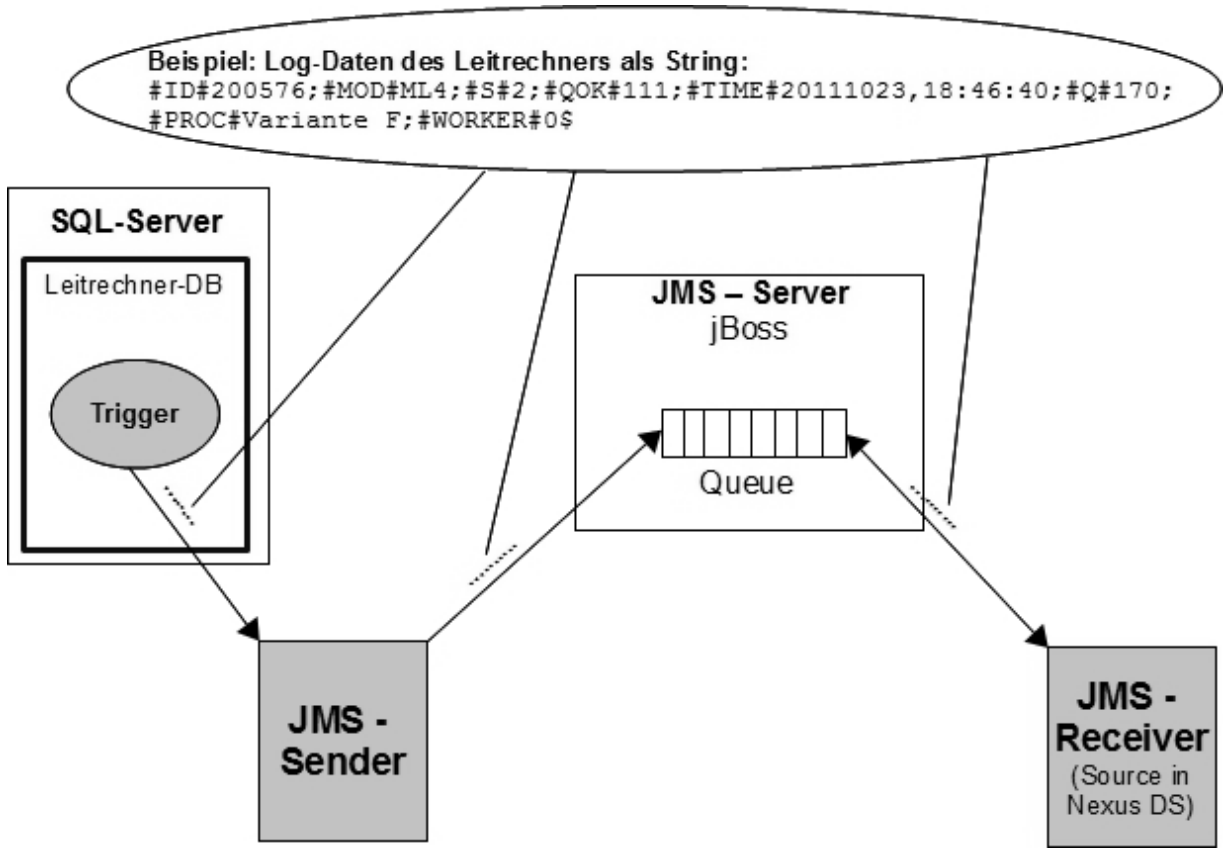

Abbildung 25: Export der Log-Daten der Leitrechnerdatenbank mittels Java Messaging Service

Zur Vervollständigung des MAPE-Zyklus werden die Log-Daten zur Analyse in die Datenstromverarbeitung NEXUS DS importiert. Im Quelloperator (*Source*) des Ausführungsgraphs ist ein *JMS-Receiver* implementiert. Somit wird eine Brücke von der Datenbank des Leitrechners, über den *JMS-Server*, hin zur Datenstromverarbeitung NEXUS DS geschlagen.

4.1 Log-Daten-Trigger

Der *Log-Daten-Trigger* ist innerhalb der Datenbank des Leitrechners implementiert. Dieser triggert bei einem Update auf die Tabelle *dbo.log*. Damit bei einer Fehlfunktion des *Triggers* keine Log-Daten verloren gehen, wird der *Trigger* nach dem Einfügen eines neuen Datensatzes (AFTER INSERT) ausgeführt. Der prinzipielle Aufbau des *Triggers* ist in Abbildung 26 beschrieben.
Der *Trigger* wird zunächst erstellt, alle nötigen Variablen werden deklariert und anschließend mit den entsprechenden Daten (siehe Kapitel 2.4.1 Leitrechner) gefüllt. Zum Schluss werden die Daten zu einem String zusammengesetzt und durch den *JMS-Sender*, welcher lokal auf dem Leitrechner gespeichert ist, an den *JMS-Server* gesendet.

```
 -- Erstellen des Triggers
CREATE TRIGGER [dbo].[MSBTrigger]
   ON  [dbo].[Log]
   AFTER INSERT
AS
 -- Variablendeklaration
declare @ID varchar(max)
declare @task varchar(max)
… usw.

begin
    -- Füllen der Variablen durch die entsprechenden Daten
    set @ID = (select ID from inserted);
    set @task = (select TaskNo COLLATE Latin1_General_CI_AS from inserted);
    … usw.
    -- Aufruf eines JMSSender (das Programm liegt lokal auf dem Leitrechner)
    set @befehl = 'java -jar JMSQueueSender.jar '
      +'"'+'#ID#'+@ID+';'+@task+';'
      +@OpNo+';#MOD#'+@modName+';#S#'+@status+';#QOK#'+@qOK+';#TIME#'
+@date+','+@time+';#Q#'+@qty+';#PROC#'+@process+';#WORKER#'+@worker+';'+@star
t+'$'+'"';
    EXEC master.dbo.xp_cmdshell @befehl;
end
```

Abbildung 26: Implementierung des Log-Daten-Triggers

Einige Bestandteile des zusendenden Strings werden mit String-Prefixe versehen, um ein späteres Parsen im *Selection-*, und *Analysis & Transformation-Operator* zu vereinfachen. Beispielsweise zeigt das Prefix *#MOD#*, dass es sich bei dem nachfolgenden Teilstring um einen Stationsnamen (*ModuleName*) der Produktionsstraße handelt.

4.2 Source-Operator

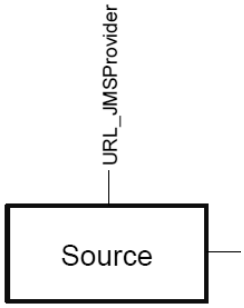

Abbildung 27: Source-Operator

Der *Source-Operator* (Abb. 27) ist der erste Operator in der Datenstromverarbeitung und dient als Datenquelle für den Fehleranalyseprozess. Der *Source-Operator* besitzt einen Parameter *URL_JMSProvider*, dessen Wert die URL des J*MS-Servers* ist. Im Quelloperator ist ein *JMS-Receiver* implementiert, der sich mit der Methode *init()* am *JMS-Server* registriert. Nach der Registrierung wird am Ende von *init()* ein *MessageListener* instantiiert, welcher die Warteschlange des *JMS-Servers* auf neue bzw. bereits anliegende Nachrichten überprüft. Sobald eine Nachricht die Warteschlange erreicht, wird diese durch den *MessageListener* abgeholt, welcher unmittelbar danach die *run()-Methode* aufruft. In der *run()-Methode* wird die Nachricht in einen *InputStream* transformiert und an den *Selection-Operator* gesendet. Die Abbildung 28 beschreibt den Aufbau des Nachrichtenstrings, der vom *Log-Daten-Trigger* bzw. *JMS-Sender* gesendet wird.

```
#ID#200576;#MOD#ML4;#S#2;#QOK#111;#TIME#20111023,18:46:40;
#Q#170;#PROC#Variante F;#WORKER#0$
```

#ID#200576: ID des Datensatzes in der Leitrechnerdatenbank

#MOD#ML4: Station, an welcher der Fehler aufgetreten ist.

#S#2: Statuswert des Datensatzes

#QOK#111: Anzahl der Produkte, die fehlerfrei produziert wurden.

#TIME#20111023,18:46:40: Zeitpunkt, an dem der Fehler aufgetreten ist.

#Q#170: Gesamtanzahl der Produkte bezogen auf den aktuellen Auftrag

#PROC#Variante F: Name der Produktvariante

#WORKER#0: ID des Mitarbeiters, bei dem der Fehler aufgetreten ist.

Abbildung 28: Aufbau des Nachrichtenstrings bzw. der Log-Daten der Leitrechnerdatenbank

4.3 Selection-Operator

Wie auch schon beim *Source-Operator* besitzt der *Selection-Operator* (Abb. 29) genau einen Parameter *Mode*. In diesem Schritt des Fehleranalyseprozesses wird eine Selektion ausgeführt. In der Methode *process()* des Operators werden zwei unterschiedliche Selektionen vorgenommen. Im ersten Schritt werden anhand des Statuswertes (im String mit dem Prefix *#S#* markiert) nur die relevanten Datensätze herausgefiltert. Lediglich Datensätze mit Statuswert „2“ enthalten brauchbare Informationen, welche für die weitere Analyse notwendig sind. Der zweite Schritt der Selektion ist vom Wert des Parameters *Mode* abhängig. Hier wird entschieden, ob ausschließlich Datensätze fehlerhafter Produkte (*Mode=0*) oder ausschließlich Datensätze fehlerfreier Produkte (*Mode=1*) oder Datensätze sowohl fehlerhafter als auch fehlerfreier Produkte (*Mode=2*) an den *Pre-Processing-Operator* weitergegeben werden. In dieser Arbeit wurde der *Selection-Operator* ausschließlich mit dem Parameterwert *Mode=0* verwendet, da bei einer Fehleranalyse nur Log-Daten dieser Art relevant sind.

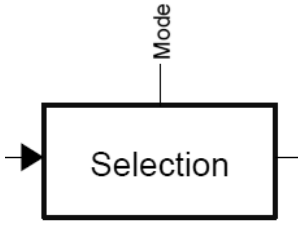

Abbildung 29: Selection-Operator

4.4 Pre-Processing-Operator

Im Pre-Processing-Operator werden die Fehlerdaten des Leitrechners für den Klassifikationsalgorithmus aufbereitet. Durch das Setzen der Parameterwerte (Abb. 30) entscheidet der Benutzer welche Klassentypen (Produktvariante, Station, Arbeiter, Zeitpunkt / Schicht) für die Klassifikation verwendet werden sollen. Die einzelnen Parameter können den Wert *true* (Information wird für die Klassifikation herangezogen) oder *false* (Information wird nicht für die Klassifikation herangezogen) annehmen. In der Methode *processingAttribute()* werden nacheinander diejenigen Informationen der Klassentypen aus dem gesendeten String (Datensatz des Leitrechners) extrahiert, welche den Parameterwert *true* besitzen. Ist beispielsweise der Parameterwert von *ProcessName=true*, so wird für die Klassifikation der Klassentyp *ProcessName*, welcher die Produktvariante enthält, herangezogen. Die konkrete Produktvariante, bei welcher in diesem Fall ein Fehler aufgetreten ist, wird an einen

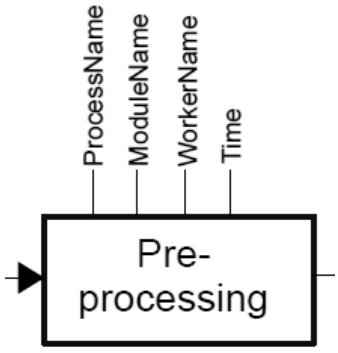

Abbildung 30: Pre-Processing-Operator

Vektor vom Datentyp String angehängt. Um später feststellen zu können, welcher Eintrag des Vektors welche Art von Information enthält (z.B. erster Eintrag ist eine Produktvariante, zweiter Eintrag ist eine ID eines Mitarbeiters), wird die Semantik durch die Methode *addClassSemantics(String)* in der Klasse *ClassSemantic* festgehalten. Die Klasse *DataRecord* dient als benutzerdefinierte Datenstruktur. Von dieser Klasse wird eine Instanz gebildet, welche zum einen den Vektor *information_vec* und zum anderen eine Datensatz-ID besitzt. Die Datensatz-ID wird von der Datenbank des Leitrechners vergeben. Der *Pre-Processing-Operator* gibt die Instanz der Klasse *DataRecord* mit den entsprechenden Informationen an den *Classification-Operator* weiter.

4.5 Classification-Operator

Beim *Classification-Operator* (Abb. 31) legt der Benutzer über die Parameter *StartTime* und *EndTime* eine Zeitspanne fest, innerhalb welcher eine Fehleranalyse durchgeführt werden soll. Mit dem dritten Parameter *HistoryFile* wird auf eine Datei (*history.txt*) verwiesen, in welcher sich vergangene Klassifikationsdaten befinden. Die Abbildung 32 beschreibt die Formatierung des *HistoryFiles*.

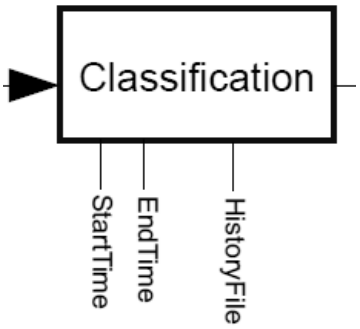

Abbildung 31: Classification-Operator

```
Datum YYYYMMTT,Zeit HHMMSS;ID;ProcessName,ModuleName,Worker-
ID,Datum und Schicht YYYYMMDD$Schicht_x (Zeilenumbruch)

Beispiel:
20120123,112658;200514;Variante_F,RS1,0,20111023$Schicht_2
```

Abbildung 32: Formatierung des HistoryFiles für die Klassifikation

In der Methode *loadHistoryData* werden diejenigen historischen Daten extrahiert, welche zum einen in der vom Benutzer angegebenen Zeitspanne liegen. Zum anderen werden nur solche Daten aus der Historie gelesen, welche dieselbe Anzahl an Klassentypen besitzt wie der aktuelle Fehlerdatensatz des *Pre-Processing-Operators*. Das heißt, wenn zum Beispiel vom *Pre-Processing-Operator* ein Datensatz mit Informationen über *ProcessName* und *ModuleName* (Anzahl der Klassentypen = 2)

gesendet wird, können folglich aus der Historie auch nur die Datensätze mit genau den selben (nicht mehr und nicht weniger) Informationen gelesen werden. Jeder extrahierte Datensatz der Historie wird, wie bereits beschrieben, in eine Instanz der Klasse *DataRecord* gepackt und an die Methode *classification* gesendet. Im Anschluss wird jeder Datensatz, welcher vom *Pre-Processing-Operator* gesendet wurde, ebenfalls mit Hilfe der Methode *classification* klassifiziert und in die Historie geschrieben (*writeToHistoryFile*).
In der Methode *classification(DataRecord p)* findet die eigentliche Klassifizierung statt. Es werden abhängig von der Art und Anzahl der Klassentypen folgende Klassen und Instanzen gebildet (Abb. 33).

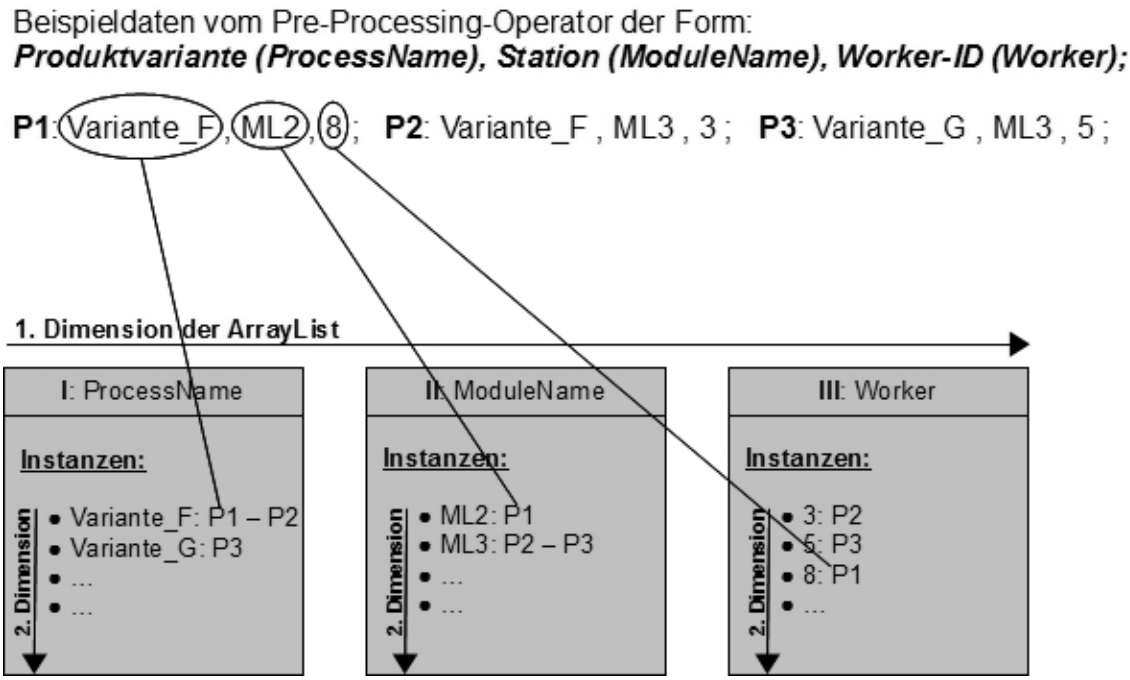

Abbildung 33: Klassifizierung. In diesem Fall sollen folgende Klassentypen für die Klassifizierung berücksichtigt werden: ProcessName, ModuleName und Worker (Anzahl der Klassentypen = 3)

Der Klassifikationsalgorithmus klassifiziert die Datensätze (P1, P2, …) abhängig von ihren Attributwerten (*ProcessName*, *ModuleName*, *Worker*). Somit wird jeder Datensatz genau einmal einer Instanz jeder Hauptklasse (I – III) zugeordnet. Die Abbildung 34 zeigt die Implementierung des Klassifikationsalgorithmus.
Die Hauptklassen I – III werden in einer zweidimensionalen ArrayList *classes* vom Datentyp *ClassType* gespeichert. In der ersten Dimension der ArrayList werden die

Hauptklassen durchnummeriert. Jede Hauptklasse besitzt in der zweiten Dimension Instanzen, an die Log-Datensätze angehängt werden. *ClassType* dient als benutzerdefinierte Datenstruktur und wird durch die gleichnamige Klasse definiert. Die Klasse *ClassType* besitzt ein Attribut, das den Namen der Instanz enthält. Des weiteren besitzt *ClassType* einen Vektor *information_vec* vom Datentyp *DataRecord*, an den alle Log-Daten angehängt werden, die zur entsprechenden Instanz gehören.

Beispielsweise ist aus der Abbildung 33 anhand des Log-Datensatzes *P1* zu erkennen, dass bei der Produktvariante F an Station ML2 durch den Mitarbeiter mit ID=8 ein Fehler entstanden ist.

```
// Klassifikationsalgorithmus
public void classification(DataRecord p){
   nbClassType = p.getPointValueVec().size();
   boolean exist = false;

   // Hauptklassen abhängig von der Anzahl der Klassentypen erzeugen
   if (classes.size()==0){
      for (int i = 0; i<nbClassType ;i++){
         classes.add(new ArrayList<Class_Type>());
      }
   }
   // Schleife über alle Hauptklassen
   for (int i=0; i<nbClassType;i++){
      exist=false;
      // Schleife durch alle Instanzen der aktuellen Hauptklasse
      for (int j=0;j<classes.get(i).size();j++){
         // Ist die gesuchte Instanz bereits vorhanden → Datensatz an Instanz
            anhängen
         if (classes.get(i).get(j).getName().equals(p.getPointValueVec().get(i))){
            classes.get(i).get(j).addPoint(p);
            exist = true;
        }
      }
      // Ist die gesuchte Instanz nicht vorhanden → Instanz anlegen und Datensatz
         anhängen
      if (exist==false){
         classes.get(i).add(new Class_Type(p.getPointValueVec().get(i),p));
      }
   }
}
```

Abbildung 34: Klassifikationsalgorithmus

Die ArrayList *classes*, welche alle Hauptklassen inklusive Instanzen und Datensätze enthält, wird an den *Analysis & Transformation-Operator* weitergegeben.

4.6 Analysis & Transformation-Operator

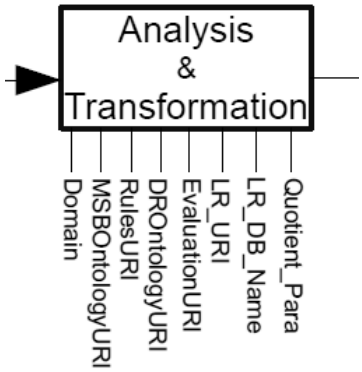

Abbildung 35: Analysis & Transformation-Operator

Beim *Analysis & Transformation-Operator* wird im ersten Schritt aus der Hauptmethode *process()* die Methode *initialization()* aufgerufen. Hierbei werden zwei Ontologien, die MSB-Ontologie und die Domain-Recommendation-Ontologie, initialisiert. In den Parametern *MSBOntologyURI* und *DROntologyURI* werden die URIs der jeweiligen Ontologien eingetragen (Abb. 35). Mit Hilfe der MSB-Ontologie (*msbOntModel*) und den entsprechenden Regeln wird die Fehleranalyse durchgeführt. Das Ergebnis der Fehleranalyse, der Lösungsvorschlag des Problems, wird in semantische Daten transformiert und in die Domain-Recommendation-Ontologie (*DROntModel*) geschrieben. Zudem wird das *Evaluation-File*, welches die für die Fehleranalyse benötigten Zusatzinformationen (Bewertung der Arbeiterstationen) enthält, initialisiert.

Die Abbildung 36 veranschaulicht den Aufgabenprozess des *Analysis & Transformation-Operators*.

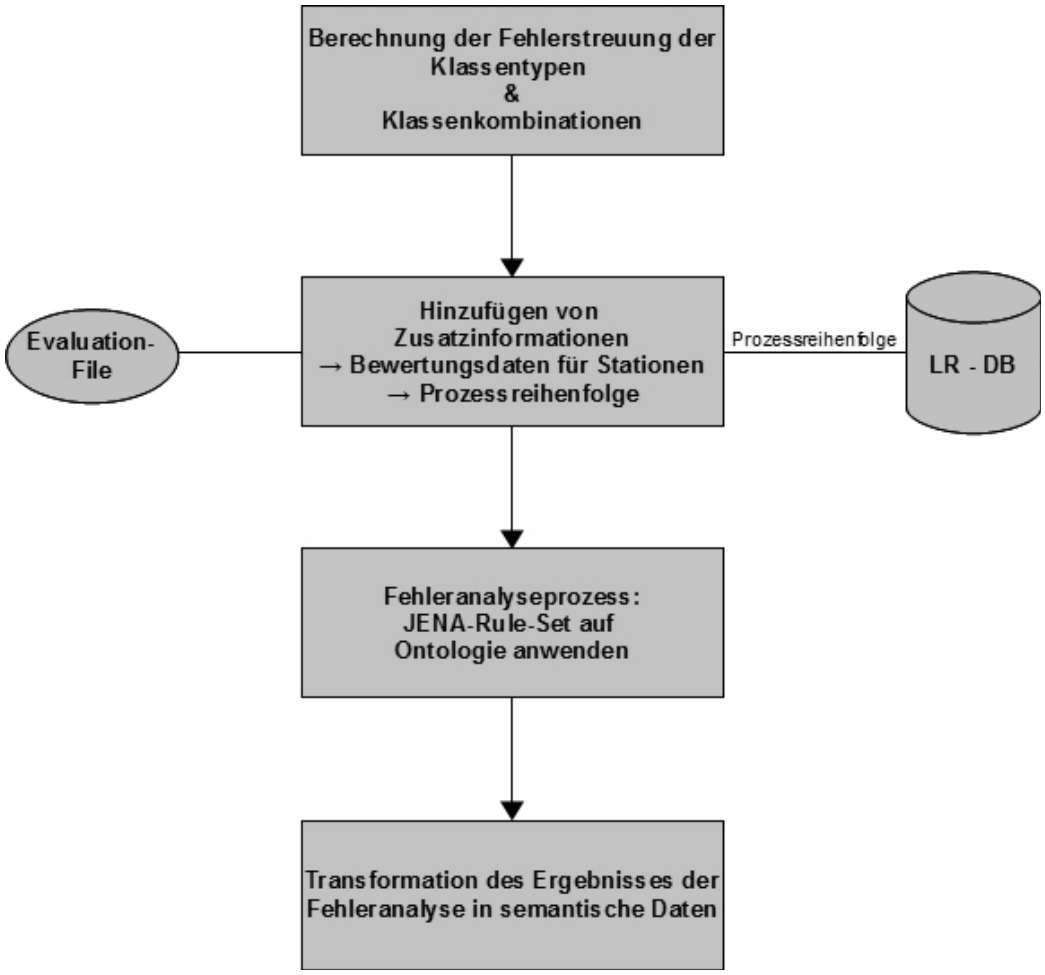

Abbildung 36: Ablauf der Fehleranalyse, deren Vorbereitung und die anschließende Transformation

4.6.1 Fehleranalyse

Um eine gewisse Sicherheit für die Korrektheit der Fehleranalyse zu erreichen, wird die Analyse erst gestartet, sobald mindestens zehn Fehlerdatensätze vom Klassifikationsalgorithmus klassifiziert wurden. Als erstes wird bei der Fehleranalyse mit Hilfe der Klassifikation, welche vom *Classification-Operator* übergeben wurde, ein Mining durchgeführt. Hierbei werden durch verschiedene Kombinationen der Hauptklassen neue Klassen gebildet und eine Fehlerstreuung berechnet. Die Abbildung 37 zeigt, welche neue Klassenkombinationen entstehen, wenn drei Klassentypen (*ProcessName*, *ModuleName*, *Worker*) für die Fehlerstreuung berücksichtigt werden sollen.

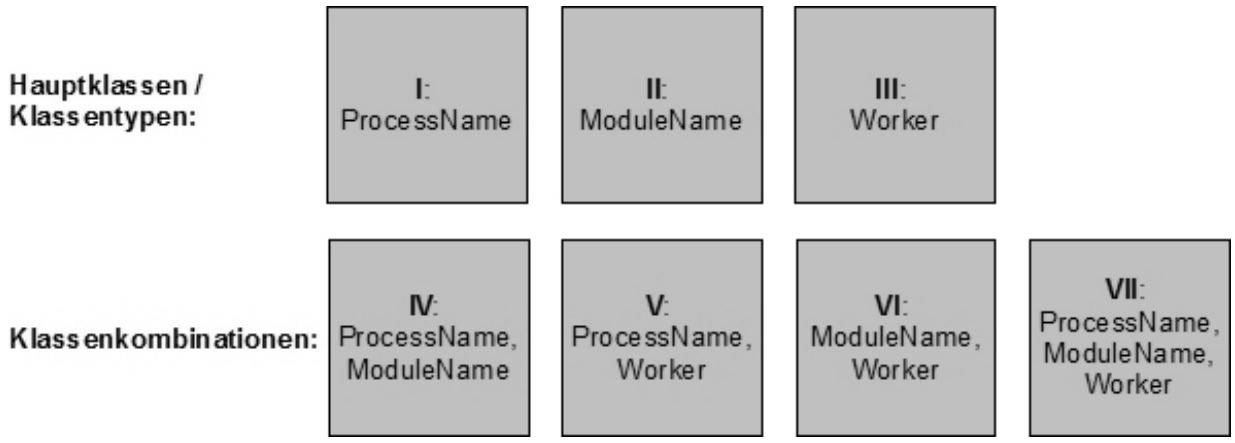

Abbildung 37: Klassenkombinationen, basierend auf den drei Klassentypen

In der Klasse II wurden die Datensätze lediglich anhand des Stationsnamens (ModuleName) den Instanzen zugeordnet. Somit befinden sich zum Beispiel in der Instanz *RS1* alle Datensätze, die auf einen Fehler an der Roboterstation *RS1* hinweisen. In Klasse VI findet unter Berücksichtigung der Klassentypen *ModuleName* (Stationsname) und *Worker* (Arbeiter) eine Zuordnung der Datensätze zu den entsprechenden Instanzen statt. Hier wären beispielsweise der Instanz *„ML1, 5“* alle Datensätze zugewiesen, welche auf einen Fehler an Station *ML1* durch den Mitarbeiter mit *ID=5* hinweisen.
Die Methode *combinations* gibt, basierend auf der Anzahl der zu berücksichtigenden Klassentypen, ein *StringArray* zurück, welches alle Klassenkombinationen und Klassentypen (Abb. 37) enthält. Angenommen es sind drei Klassentypen zu berücksichtigen, so enthält das *StringArray* sieben Felder, wobei drei Felder für die Klassentypen selbst stehen. Die restlichen vier Felder stehen für die Klassenkombinationen. In diesem Fall wird jede Kombination durch eine dreistellige, binäre Zahl ausgedrückt. Jede „1“ in der Binärzahl bedeutet, dass der entsprechende Klassentyp für die aktuelle Klassenkombination verwendet wird.
Nun wird für die Instanzen der Klassen I-VII (Abb. 37) eine Fehlerstreuung berechnet. Die Abbildung 38 veranschaulicht diesen Prozess.

Fehlerdaten:
ML2, Variante_F, 1
ML2, Variante_F, 2
ML2, Variante_F, 3
ML2, Variante_G, 2
ML1, Variante_F, 3
ML1, Variante_F, 3
ML1, Variante_G, 1
ML1, Variante_G, 2

Klassen:

I: ProcessName	II: ModuleName	III: Worker	IV: ProcessName & ModuleName
• Variante_F: 5 Fehler • Variante_G: 3 Fehler	• ML1: 4 Fehler • ML2: 4 Fehler	• ID=3: 3 Fehler • ID=1: 2 Fehler • ID=2: 3 Fehler	• ML1, Variante_F: 2 Fehler • ML1, Variante_G: 2 Fehler • ML2, Variante_F: 3 Fehler • ML2, Variante_G: 1 Fehler

V: ProcessName & Worker	VI: ModuleName & Worker	VII: ProcessName & ModuleName & Worker
• Variante_F, 3: 3 Fehler • Variante_F, 2: 1 Fehler • Variante_F, 1: 1 Fehler • Variante_G, 1: 1 Fehler • Variante_G, 2: 2 Fehler	• ML1, 3: 2 Fehler • ML1, 1: 1 Fehler • ML1, 2: 1 Fehler • ML2, 1: 1 Fehler • ML2, 2: 2 Fehler • ML2, 3: 1 Fehler	• ML1, Variante_F, 3: 2 Fehler • ML2, Variante_F, 2: 1 Fehler • ML2, Variante_G, 2: 1 Fehler • ML1, Variante_G, 1: 1 Fehler • ML1, Variante_G, 2: 1 Fehler • ML2, Variante_F, 1: 1 Fehler • ML2, Variante_F, 3: 1 Fehler

Abbildung 38: Berechnung der Fehlerstreuung im Zuge des Mining-Prozesses

Der Fehlerdatensatz *„ML2, Variante_F,1“* aus Abbildung 38 bedeutet:

Bei der Herstellung von Produktvariante F an Station *ML2* ist durch den Mitarbeiter mit *ID=1* ein Fehler aufgetreten.

Durch die Berechnung der Fehlerstreuung für die Instanzen der Klassen I-VII erhält man einen Mehrwert an Wissen. Dieses Wissen wird in der MSB-Ontologie (Abb. 39) gespeichert.

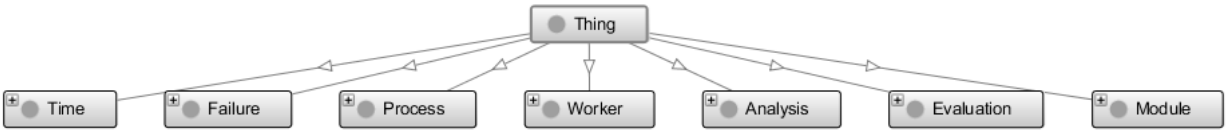

Abbildung 39: Klassen der MSB-Ontologie

Die Fehlerstreuung für die Klassentypen (Klasse I-III) werden in der entsprechenden Ontologieklasse (*Process*, *Module*, *Worker*, *Time*) gespeichert.

Beispielsweise könnte für ein Individuum (*Variante_F)* der Klasse *Process* folgende Fehlerstreuung vermerkt werden:

Bei der Herstellung von Produktvariante F sind 10 Fehler gemeldet worden.

Für die Speicherung der Fehlerstreuung der Klassenkombinationen (Klasse IV-VII) gibt es eine gemeinsame Ontologieklasse *Failure*. In dieser Klasse werden über *ObjectProperties* und *DataProperties* sowohl Informationen zur Fehlerstreuung, als auch die beteiligten Klassentypen, zum Beispiel *ModuleName* & *Worker*, abgespeichert. Beispielsweise könnte für ein Individuum (*FailureNb1)* der Klasse *Failure* folgendes gespeichert werden:

FailureNb1 bezieht sich auf Station ML1 (ObjectProperty „hasModule“) und Mitarbeiter ID=5 (ObjectProperty „hasWorker“) & an ML1 sind durch den Mitarbeiter (ID=5) 7 Fehler entstanden (DataProperty „hasFailure“).

Eine *ObjectProperty* beschreibt eine Eigenschaft bzw. Beziehung zwischen zwei Individuen einer Ontologieklasse.
Eine *DataProperty* beschreibt eine Beziehung zwischen einem Individuum und einem *Built-In datatype*, zum Beispiel einem String, Integer, Boolean, usw.

Bei der Implementierung des Mining-Prozesses sind drei Methoden beteiligt. In der Methode *mining()* wird nach der Berechnung der Klassenkombinationen die Fehlerstreuung für die Klassentypen berechnet. Zur Berechnung der Fehlerstreuung bei den Klassenkombinationen wird zunächst die Methode *dimensions_2()* aufgerufen. In dieser Methode wird die Fehlerstreuung für alle Kombinationen berechnet, bei denen genau zwei Klassentypen beteiligt sind. Sind mehr als zwei Klassentypen beteiligt, so wird die Methode *dimensions_n()* aufgerufen und die Fehlerstreuung für die entsprechenden Klassenkombinationen berechnet.
Zusätzlich zum neu erlangten Wissen durch das Mining sind dennoch weitere Informationen nötig, um eine Fehleranalyse durchzuführen. Durch die Methode *addEvaluationData* werden Bewertungsdaten für die Arbeiterstationen (ML1, ML2, …) an die Klasse Evaluation (Abb. 39) der MSB-Ontologie angefügt. Hierbei werden aus einer Text-Datei, dessen Speicherort durch den Parameter *EvaluationURI* (Abb. 35) festgelegt ist, Arbeiterstationen hinsichtlich der Montageanweisungen, der Ergonomie

und des Zeitdrucks benotet. Im *Evaluation-File* werden die Durchschnittsnoten, bezüglich einer Station und der entsprechenden Produktvariante, als Float-Werte eingetragen. Die Notenskala reicht von der Note 1.0 (sehr gute Montageanweisungen – sehr gute Ergonomie – sehr geringer Zeitdruck), bis hin zur Note 6.0 (ungenügende Montageanweisungen – ungenügende Ergonomie – sehr hoher Zeitdruck). Zudem ist in der Evaluationsdatei noch eine Information über den Reifegrad der verschiedenen Produktvarianten gespeichert. Diese Skala reicht von 1-3 (1 = neu eingeführte Variante, 3 = lang vorhandene (reife) Produktvariante).
Um später die Fehlerquelle korrekt identifizieren zu können, muss aus der Datenbank des Leitrechners (*LR_URI* – Abb. 35) die Prozessreihenfolge[7] der Produktherstellung abgefragt werden. In der Methode *processSequence* wird dafür mittels JDBC eine Verbindung zur Datenbank aufgebaut. Die Prozessreihenfolge wird aus der Tabelle *dbo.tmpProcess* abgerufen und in einer *ArrayList* gespeichert.
Zunächst sind genug Informationen vorhanden um die Fehleranalyse mit Hilfe der MSB-Ontologie und den entsprechenden Regeln (JENA-Rule-Set) zu starten. Damit man mit den Regeln auf beide Ontologien, die MSB-Ontologie und die DR-Ontologie, Zugriff hat, müssen diese mittels einer UNION-Methode zu einem *OntModel* vereinigt werden. Durch die Methode *runEngine* wird die Regelmaschine für ein *OntModel* zusammen mit einem Regel-Set (siehe Anhang A1) gestartet. Da die Ausführungsreihenfolge bei JENA-Rules nicht voraussagbar ist, wurde für diese Arbeit ein Parser entwickelt, der alle Regeln bis zum $-Zeichen einliest und daraufhin ausführt (Abb. 40).

7 Prozessreihenfolge: Zeitliche und physische Reihenfolge der Stationen, an welchen die Produkte bearbeitet werden.

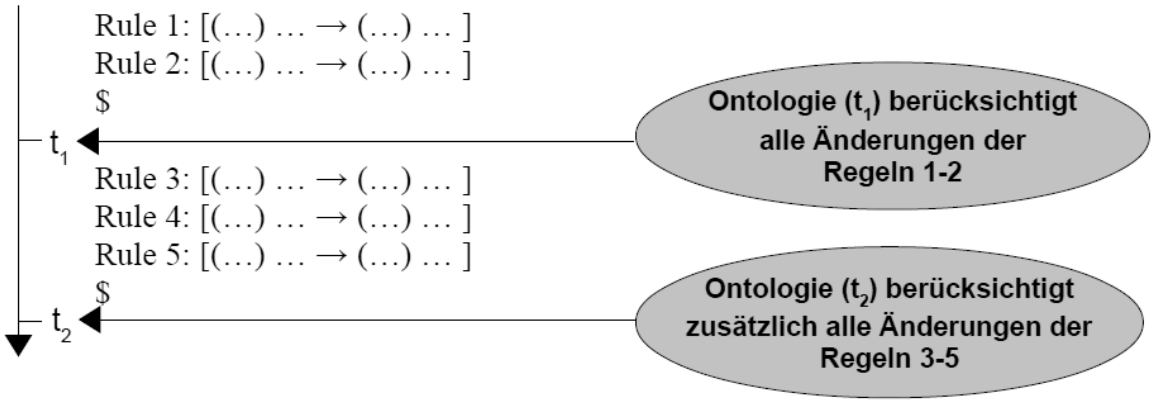

Abbildung 40: Parser für JENA-Rules

So wird sicher gestellt, dass die Regeln vor einem $-Zeichen zeitlich vor den Regeln nach diesem $-Zeichen ausgeführt werden. Dadurch ist die Ausführung von abhängigen Regeln möglich, da alle Ontologieänderungen nach dem lesen eines $-Zeichens sichtbar werden.
Der Fehleranalyseprozess gliedert sich in fünf Schritte:

1. Auffälligkeiten finden
2. Einen Verdacht für die Fehlerquelle schöpfen und überprüfen
3. Auswertung der Zusatzinformationen (Bewertungsdaten)
4. Lösung finden
5. Lösung in semantische Daten transformieren

In den Regeln (Anhang A1) zur Fehleranalyse, deren Speicherort durch den Parameter *RulesURI* (Abb. 35) festgelegt ist, lassen sich diese fünf Schritte wieder finden. Im folgenden wird die Bedeutung jeder dieser Regeln beschrieben:

- Rule 1: Hier wird die maximale Fehleranzahl gesucht, welche an einer Station entstanden ist. In der MSB-Ontologie gibt es eine *DataProperty* „hasAFirstMax“, dessen Wert zu Beginn auf 0 gesetzt wurde. Nun werden alle Individuen der Klasse *Module* durchsucht. Wenn dessen Fehleranzahl größer ist als der Wert von „hasAFirstMax“, so wird der Wert von „hasAFirstMax“ neu gesetzt.

- Rule 2: Bei dieser Regel wird die zweithöchste Fehleranzahl einer Station gesucht. Dafür wurde der Wert der DataProperty „hasASecondMax“ zu Beginn auf 0 gesetzt. Nun werden wieder alle Individuen der Klasse Module hinsichtlich ihrer Fehleranzahl untersucht. Ist die Fehleranzahl größer als der Wert von „hasASecondMax“ und kleiner als der Wert von „hasAFirstMax“, so wird der Wert von „hasASecondMax“ aktualisiert.

- Rule 3: In der Regel 3 wird der Quotient der Werte von „hasAFirstMax“ und „hasASecondMax“ berechnet. Ist dieser Quotient größer als der Wert, welcher vom Benutzer mittels des Parameters *Quotient_Para* (Abb. 35) festgelegt wurde, so besteht ein Verdacht, dass ein Problem innerhalb der Produktionsstraße vorliegt. In diesem Fall wird der Wert der *DataProperty* „hasAAsserQuotient“ auf *true* gesetzt. Beispiel (Abb. 41):

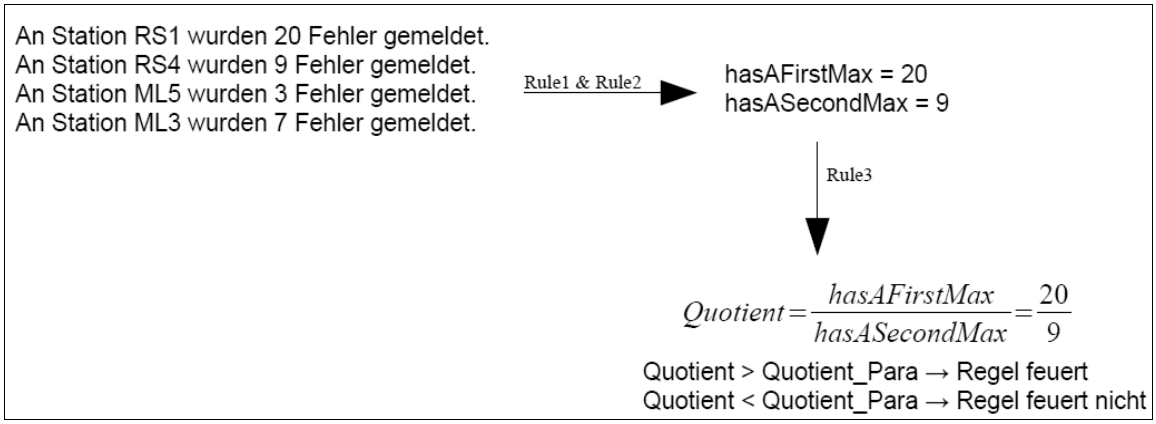

Abbildung 41: Regel 3: Berechnung des Quotienten

- Rule 4: Diese Regel feuert nur dann, wenn „hasAAsserQuotient“ den Wert *true* besitzt, ansonsten würde kein Verdacht für ein Problem bestehen. Die Regel 4 sucht anhand der Fehleranzahl bzw. des Wertes von „hasAFirstMax“ diejenige Station, welche den Produktionstop verursacht hat und speichert den Stationsnamen als Wert der *ObjectProperty* „hasAStopModule“.
- Rule 5: Vor der Ausführung der Regel 5 wird aus dem Vektor der Methode *processSequence()* diejenige Station herausgesucht, welche zeitlich und physisch direkt vor der Station liegt, die den Produktionstop verursacht hat. Es

wird vermutet, dass der eigentliche Fehler nicht an der Stop-Station liegt, sondern an der Vorgänger-Station. Der Name der Vorgänger-Station ersetzt die Variable *%FaultyModule* in der Regel. Somit erhält die *DataProperty* „hasAFaultyModule“ als Wert den Namen der Vorgänger-Station.
Beispiel (Abb. 42):

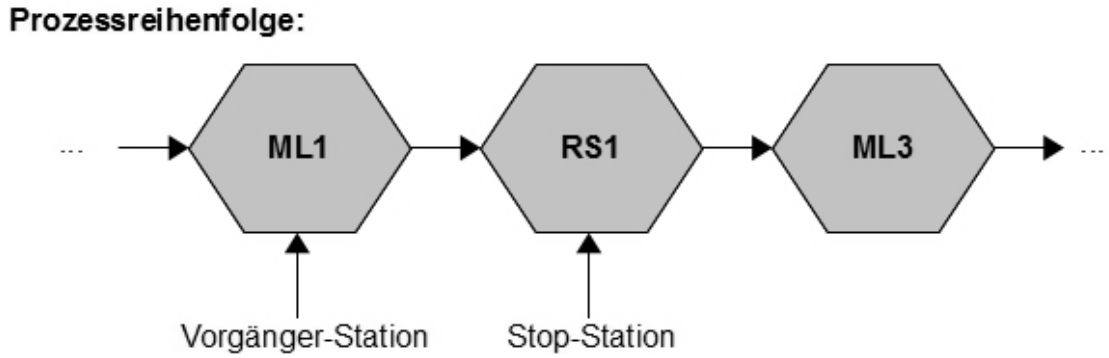

Abbildung 42: Prozessreihenfolge

- Rule 6: Die meisten Fehler sind an der Stop-Station entstanden. Die Regel 6 untersucht bei welcher Produktvariante die meisten Fehler, bezogen auf die Stop-Station, entstanden sind. Hierfür werden diejenigen Individuen der Klasse *Failure* herangezogen, welche zur Klassenkombination *„ModuleName & ProcessName“* (Abb. 37 – Klasse IV) gehören. Wie schon zuvor, wird auch hier die maximale Fehleranzahl mit Hilfe der *DataProperty* „hasACountProcFailure“, welche anfangs den Wert 0 besitzt, ermittelt.
- Rule 7: Diese Regel sucht anhand der maximalen Fehleranzahl „hasACountProcFailure“ den Namen derjenigen Produktvariante, bei welcher, bezogen auf die Stop-Station, die meisten Fehler entstanden sind. Der Name der Produktvariante wird als String in der *ObjectProperty* „hasAFaultyProcess“ gespeichert.

Die DataProperty „hasACountProcFailure“ speichert die Anzahl der Fehler (Integer-Wert – Rule 6), welche bei der Produktvariante (Name als String – Rule 7), bezogen auf die Stop-Station, gemeldet wurden.

Die Regeln 8-11 lesen und überprüfen die Bewertungsdaten der Arbeiterstationen, welche durch eine Mitarbeiterbefragung entstanden sind. Aus der Ontologieklasse

Evaluation werden diejenigen Daten herangezogen, welche Bewertungen für die entsprechende Arbeiterstation (Wert von „hasAFaultyModule“), unter Bearbeitung der entsprechenden Produktvariante (Wert von „hasAFaultyProcess“), enthalten.

- Rule 8: Die Regel 8 überprüft die Bewertungen hinsichtlich der Montageanweisungen. Ist die Durchschnittsnote (Wert von „hasEAssemblyInstructionsAVGGrade“) schlechter als 4.0, so wird in diesem Fall der Verdacht, dass der Fehler an der Station *FaultyModule* liegt, bestätigt. Die Bestätigung des Verdachts wird durch das Setzen der *DataProperty* „hasAaigConfirmed=true“ gekennzeichnet.
- Rule 9: Hier wird der Reifegrad (Wert von „hasEMaturityOfVariante“) der fehlerhaften Produktvariante überprüft. Ist dieser Wert kleiner 2.0, so handelt es sich um eine neuere Produktvariante. Dies spricht dafür, dass auf Grund des Reifegrades der Fehler an der Station *FaultyModule* entstanden ist. Auch in diesem Fall würde sich der Fehlerverdacht bestätigen („hasAmofConfirmed=true“).
- Rule 10: Diese Regel wertet den Zeitdruck (Wert von „hasEPressureOfTimeAVGGrade“) aus. Ist diese Durchschnittsnote schlechter als 4.0, wird der Verdacht bezüglich des Fehlers an der Station *FaultyModule* bestätigt („hasApotgConfirmed=true“).
- Rule 11: Bei dieser Regel wird die Benotung hinsichtlich der Ergonomie (Wert von „hasEErgonomicsAVGGrade“) an der Station *FaultyModule* ausgewertet. Ist hierbei die Durchschnittsnote schlechter als 4.0, wird wiederum der anfängliche Fehlerverdacht bestätigt („hasAegConfirmed=true“).

In den Regeln 12-16 wird nachgeschaut, in wie vielen Fällen der Verdacht, dass der Fehler an der Station *FaultyModule* liegt, bestätigt wurde.

- Rule 12: Es wird überprüft, ob die Regel 9, die Regel 10 und die Regel 11 gefeuert haben.
- Rule 13: Diese Regel überprüft, ob die Regel 8, die Regel 9 und die Regel 10 gefeuert haben.

- Rule 14: Die Regel 14 überprüft, ob die Regel 8, die Regel 9 und die Regel 11 gefeuert haben.
- Rule 15: In dieser Regel wird überprüft, ob die Regel 8, die Regel 10 und die Regel 11 gefeuert haben.
- Rule 16: Mit der Regel 16 wird überprüft, ob alle vier Regeln (Regel 8-11) gefeuert haben.

Der Verdacht gilt als endgültig bestätigt, wenn mindestens eine der fünf Regeln 12-16 gefeuert hat. Ist dies der Fall, wird der Wert der *DataProperty* „hasAErrorConfirmed“ auf *true* gesetzt.

4.6.2 Transformation

Mit den Regeln 17-22 werden die Ergebnisse der Fehleranalyse in der DR-Ontologie festgehalten. Voraussetzung für das Feuern dieser Regeln ist, dass die *DataProperty* „hasAErrorConfirmed“ den Wert *true* besitzt.

- Rule 17: Hier wird dem Individuum der Klasse *Condition* als Wert für die Eigenschaft „hasConditionValue“ der Name der fehlerhaften Produktvariante (Wert von „hasAFaultyProcess“) zugewiesen.
- Rule 18: Diese Regel setzt als Wert für das Individuum der Klasse Domain, mit der Eigenschaft „hasDomainValue“, den Namen der Analyse ein. Da es sich bei dieser Arbeit um eine Fehleranalyse handelt, wird „hasDomainValue“ als Wert den String „FailureManagement“ bekommen. Den Namen der Analyse kann der Benutzer über den Operator-Parameter *Domain* (Abb. 35) angeben.
- Rule 19 & Rule 20: Bei diesen Regeln wird überprüft, ob der entstandene Fehler *online*, das heißt ohne Produktionstop, oder *offline*, das heißt mit Produktionstop, repariert werden kann. Handelt es sich beim *FaultyModule* um keine Roboterstation („hasAisFModRobotStation=false“) und ist es möglich an der Station *FaultyModule* Reparaturanweisungen am Display anzeigen zulassen, so kann der Fehler *online* behoben werden. Um herauszufinden, ob es möglich ist Reparaturanweisungen anzuzeigen, wird vor der Ausführung der Regel eine Datenbankverbindung zum Leitrechner aufgebaut. In der Tabelle

dbo.ModuleProperty in der Spalte *repairInstruct* ist vermerkt, ob Reparaturanweisungen an der Fehlerstation anzuzeigen sind. Ein Integerwert von 1 bedeutet, dass es möglich ist Reparaturanweisungen anzeigen zu lassen. Dieser Integerwert ersetzt vor der Ausführung der Regel 19 die Variable %RepairInstruct. Dementsprechend wird der Wert der Eigenschaft „hasMannerValue“ auf *online* oder *offline* (Rule 20) gesetzt.

- Rule 21: Hier wird dem Individuum der Klasse *Location* über die Eigenschaft „hasLocationValue“ als Wert der Name des *FaultyModule* zugewiesen.

Rule 22 beschreibt gemäß dem Grundaufbau semantischer Daten, Subjekt – Prädikat – Objekt, das Prädikat und das Objekt. Für das Prädikat erhält die Eigenschaft „hasPredicateValue“ der DR-Ontologie den Wert „Repair“. Für das Objekt erhält die Eigenschaft „hasObjectValue“ den Wert „Failure“.
Nach der Ausführung aller Regeln ist die Fehleranalyse abgeschlossen und das Ergebnis in der DR-Ontologie vermerkt. Mit der Methode *setDRValues* werden die vor dem Starten der Regel-Maschine vereinten Ontologien wieder aufgetrennt. Lediglich die DR-Ontologie wird als RDF/XML-Serialisierung an den Sink-Operator gesendet.

4.7 Sink-Operator

Der *Sink-Operator* bildet als Web-Service Client die Schnittstelle zum *Provenance-aware Service Repository (PASR)*. Das Ergebnis des *Analysis & Transformation-Operators* wird in eine Warteschlange des Sink-Operators geschrieben. Diese Warteschlange wird in der init-Methode des Sink-Operators instantiiert und kann maximal 20 Ergebnisse (InputStreams) speichern. Mit dem Parameter *PASRLocator* (Abb. 43) kann der Benutzer die URL des Web-Services / Repository angeben. Nach dem Abruf des aktuellen Ergebnisses aus der Warteschlange wird der *InputStream* in einen *String* umgewandelt und als SOAP-Nachricht an das Repository gesendet.

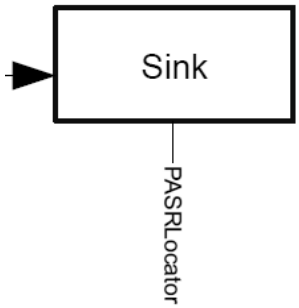

Abbildung 43: Sink-Operator

5. Ergebnisse

Als Testumgebung für die Produktionsfehleranalyse dient die Lernfabrik des Instituts für industrielle Fertigung und Fabrikbetrieb (IFF) der Universität Stuttgart. Als Beispielprodukt wird ein Arbeitsplatzaccessoire (Abb. 44) in verschiedenen Ausprägungsvarianten gewählt.

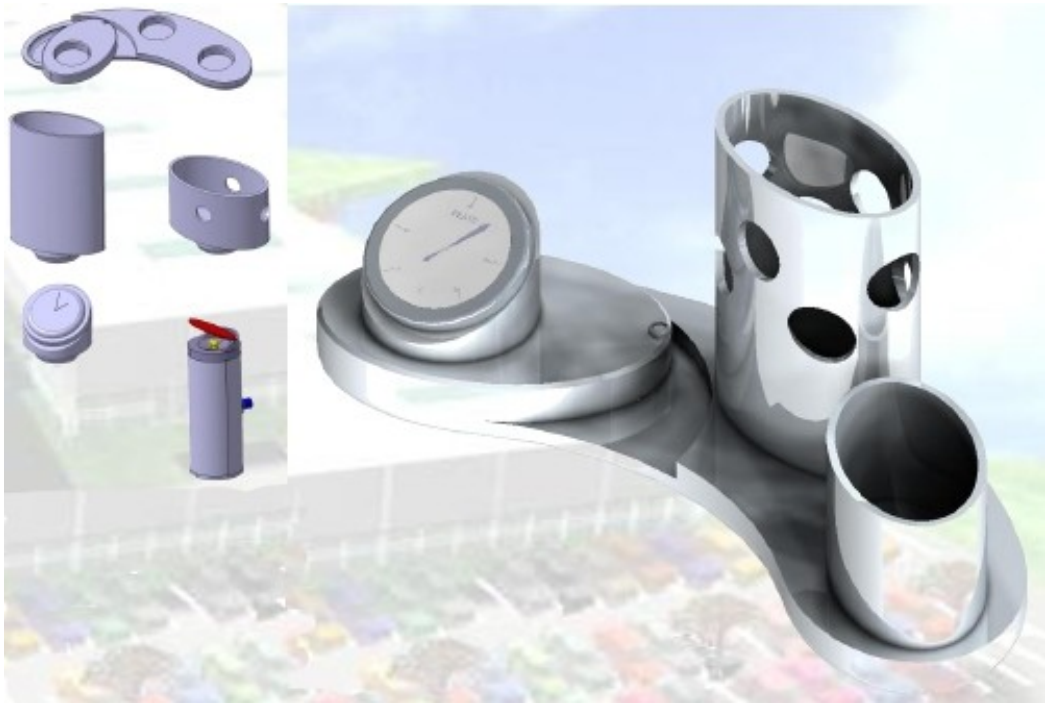

Abbildung 44: Arbeitsplatzaccessoire mit den verschiedenen Baugruppen

Das Beispielprodukt besteht aus einer Grundplatte mit drei standardisierten Schnittstellen für Baugruppen. Zu den Baugruppen zählen unterschiedlich hohe Behälter für Stifte, Büroklammern oder Radiergummis. Des weiteren existieren verschiedene Anzeigeinstrumente für Zeit, Temperatur und Luftdruck. Da alle Baugruppen dieselbe Schnittstelle besitzen, welche zu den Schnittstellen der Grundplatte passen, ergibt sich eine hohe Variantenvielfalt [25].
Um das Arbeitsplatzaccessoire herzustellen sind in diesem Beispiel drei Stationen der Produktionsstraße beteiligt. Die Prozessreihenfolge zur Herstellung des Produkts zeigt die Abbildung 45.

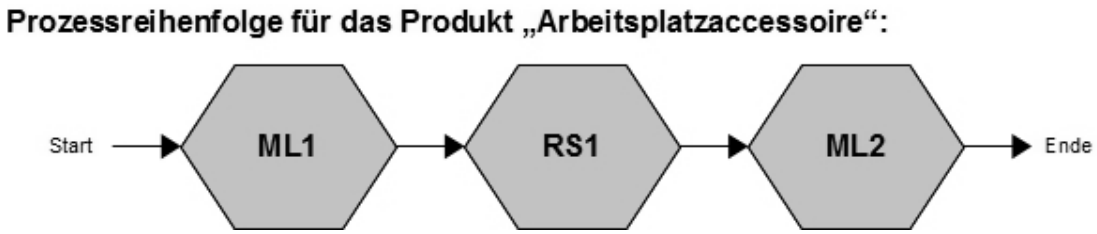

Abbildung 45: Prozessreihenfolge für das Produkt „Arbeitsplatzaccessoire“

Zum Test der implementierten Produktionsfehleranalyse werden künstlich innerhalb der Produktion, an der Roboterstation RS1, eine deutlich höhere Anzahl an Fehler erzeugt, als an den anderen beiden Stationen. Das Ergebnis der Fehleranalyse wird zeigen, dass die Fehlerquelle mit großer Wahrscheinlichkeit an Station ML1 liegt.

Für den Monitoring-, und Voranalyseprozess wird in NEXUS DS ein Ausführungsgraph geladen. Die vier Parameter des *Pre-Processing-Operators* und der Schwellwert für die Fehleranalyse (*Quotient_Para*) werden folgendermaßen gesetzt:

ProcessName = true
ModuleName = true
Worker = true
Time = false
Quotient_Para = 2

Das heißt, für diese Beispielfehleranalyse werden drei Klassentypen (ProcessName, ModuleName, Worker) verwendet. Der Klassentyp Time, welcher Auskunft gibt in welcher Schicht der Fehler gemeldet wurde, wird hier nicht berücksichtigt. Das Setzen des Parameters *Quotient_Para* auf den Wert 2 bedeutet, dass die Fehleranalyse startet, sobald an einer Station mehr als doppelt so viele Fehler gemeldet wurden, als an den anderen Stationen.
Die restlichen Parameter werden ihrer Beschreibung (Tabelle 3) entsprechend mit Werte gefüllt. Nach dem Laden des Ausführungsgraphs in NEXUS DS ist die Datenstromverarbeitung zur Fehleranalyse bereit und wartet auf Log-Daten von der Leitrechnerdatenbank der Lernfabrik. Es werden neun Beispielaufträge für die Produktion des Arbeitsplatzaccessoires in der Leitrechnerdatenbank angelegt. Die

Aufträge *test1 - test6* entsprechen der Produktion von jeweils 156 Arbeitsplatzaccessoires der Variante F.
Die Aufträge *test7 - test9* entsprechen der Produktion von jeweils 170 Accessoires der Produktvariante G. In die Tabelle *dbo.log* der Leitrechnerdatenbank werden manuell nacheinander folgende Fehlerdatensätze eingefügt (Abb. 46).

```
[TaskNo],[InstanceNo] ,[OperationNo],[OperationName],[ModuleName],
[PaletteID],[TimeStamp],[Status],[QuantityOK],[Stamped],[worker])

DS1: 'test1',1,50,'AsmDeskset','RS1',0,'20120223 06:46:40.547',2,111,0,0
DS2: 'test1',1,60,'Remove','ML2',0,'20120223 06:48:40.547',2,125,0,0
DS3: 'test2',1,50,'AsmDeskset','RS1',0,'20120223 07:06:40.547',2,120,0,0
DS4: 'test3',1,50,'AsmDeskset','RS1',0,'20120223 07:36:40.547',2,101,0,0
DS5: 'test4',1,50,'AsmDeskset','RS1',0,'20120223 08:00:40.547',2,135,0,0
DS6: 'test4',1,60,'Remove','ML2',0,'20120223 08:02:40.547',2,125,0,4
DS7: 'test5',1,50,'AsmDeskset','RS1',0,'20120223 08:46:40.547',2,128,0,0
DS8: 'test6',1,50,'AsmDeskset','RS1',0,'20120223 11:16:40.547',2,98,0,0
DS9: 'test7',1,50,'AsmDeskset','RS1',0,'20120223 12:16:40.547',2,90,0,0
DS10: 'test8',1,50,'AsmDeskset','RS1',0,'20120223 12:46:40.547',2,144,0,0
DS11: 'test8',1,60,'Remove','ML2',0,'20120223 12:52:40.547',2,125,0,3
DS12: 'test9',1,50,'AsmDeskset','RS1',0,'20120223 13:06:40.547',2,110,0,0
```

Abbildung 46: Beispieldatensätze zur Fehleranalyse

Nach dem Einfügen von Datensatz 12 (DS12) gibt der *Classification-Operator* folgende Klassifikation (Abbildung 47) an den *Analysis & Transformation-Operator* weiter. Auf der Grundlage dieser Klassifikation wird im Analyseteil des *Analysis & Transformation-Operators* die Fehlerstreuung berechnet (Abbildung 48). Anschließend wird zusammen mit den Bewertungsdaten (Abbildung 49) und dem JENA-Rule-Set (Anhang A1) die Fehleranalyse durchgeführt.

```
   Klassentyp I: Worker
Instanz 0: DS1-DS2-DS3-DS4-DS5-DS7-DS8-DS9-DS10-DS12
Instanz 4: DS6
Instanz 3: DS11
_____________________________________________________________________
   Klassentyp II: Module
Instanz RS1: DS1-DS3-DS4-DS5-DS7-DS8-DS9-DS10-DS12
Instanz ML2: DS2-DS6-DS11
_____________________________________________________________________
   Klassentyp III: Process
Instanz Variante_F: DS1-DS2-DS3-DS4-DS5-DS6-DS7-DS8
Instanz Variante_G: DS9-DS10-DS11-DS12
```

Abbildung 47: Klassifikation nach dem Einfügen der Datensätze aus Abb. 46

```
   Beteiligte Klassentypen: Worker
Instanz 0 besitzt Fehler: 10
Instanz 4 besitzt Fehler: 1
Instanz 3 besitzt Fehler: 1

_____________________________________________________________________
   Beteiligte Klassentypen: Module
Instanz RS1 besitzt Fehler: 9
Instanz ML2 besitzt Fehler: 3

_____________________________________________________________________
   Beteiligte Klassentypen: Process
Instanz Variante_F besitzt Fehler: 8
Instanz Variante_G besitzt Fehler: 4

_____________________________________________________________________
   Beteiligte Klassentypen: Module
   Beteiligte Klassentypen: Worker
Instanz RS1 & Instanz 0 besitzen 9 Fehler.
Instanz ML2 & Instanz 0 besitzen 1 Fehler.
Instanz ML2 & Instanz 4 besitzen 1 Fehler.
Instanz ML2 & Instanz 3 besitzen 1 Fehler.

_____________________________________________________________________
   Beteiligte Klassentypen: Process
   Beteiligte Klassentypen: Worker
Instanz Variante_F & Instanz 0 besitzen 7 Fehler.
Instanz Variante_F & Instanz 4 besitzen 1 Fehler.
Instanz Variante_G & Instanz 0 besitzen 3 Fehler.
Instanz Variante_G & Instanz 3 besitzen 1 Fehler.

_____________________________________________________________________
   Beteiligte Klassentypen: Process
   Beteiligte Klassentypen: Module
Instanz Variante_F & Instanz RS1 besitzen 6 Fehler.
Instanz Variante_F & Instanz ML2 besitzen 2 Fehler.
Instanz Variante_G & Instanz RS1 besitzen 3 Fehler.
Instanz Variante_G & Instanz ML2 besitzen 1 Fehler.

_____________________________________________________________________
   Beteiligte Klassentypen: Process
   Beteiligte Klassentypen: Module
   Beteiligte Klassentypen: Worker
Instanz Variante_F & Instanz RS1 & Instanz 0 besitzen 6 Fehler.
Instanz Variante_F & Instanz ML2 & Instanz 0 besitzen 1 Fehler.
Instanz Variante_F & Instanz ML2 & Instanz 4 besitzen 1 Fehler.
Instanz Variante_G & Instanz RS1 & Instanz 0 besitzen 3 Fehler.
Instanz Variante_G & Instanz ML2 & Instanz 3 besitzen 1 Fehler.
```

Abbildung 48: Berechnung der Fehlerstreuung

```
'Module','Process','AssemblyInstructionsAVGGrade','ErgonomicsAVG
Grade','PressureOfTimeAVGGrade','MaturityOfVariante'
'ML1','Variante_F',5.0,4.0,4.5,3
'ML1','Variante_G',1.0,2.0,2.0,3
```

Abbildung 49: Bewertungsdaten für die Station ML1.

Liegt das Ergebnis der Fehleranalyse vor, sprich es existiert ein Lösungsvorschlag für das aktuelle Problem, so muss die Lösung in semantische Daten transformiert werden. Dies geschieht durch das Ausführen der Regeln 17-22 (Anhang A1 – JENA-Rule-Set). Nach der Transformation wird der Lösungsvorschlag in Form von RDF-Tripels an den *Sink-Operator* gesendet. Wie man in Abbildung 50 erkennen kann, ergab die Fehleranalyse, dass der Fehler bei der *Produktvariante F* an der Station *ML1* repariert werden muss. Da die Station *ML1* in der Lage ist Reparaturanweisungen für den Arbeiter anzuzeigen, kann der Fehler ohne Produktionstop – *online* – repariert werden.

```
<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/10/
   Ontology1321208040015.owl#domainRLocation">
   <Ontology1321208040015:hasLocationValue>
      ML1</Ontology1321208040015:hasLocationValue>
   <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/10/
      Ontology1321208040015.owl#Location"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/10/
   Ontology1321208040015.owl#domainRManner">
   <Ontology1321208040015:hasMannerValue>
      online</Ontology1321208040015:hasMannerValue>
   <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/10/
      Ontology1321208040015.owl#Manner"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/10/
   Ontology1321208040015.owl#domainRCondition">
   <Ontology1321208040015:hasConditionValue>
      http://www.semanticweb.org/ontologies/2011/10/
      MSBOntology.owl#Variante_F
   </Ontology1321208040015:hasConditionValue>
   <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/10/
      Ontology1321208040015.owl#Condition"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
</rdf:Description>

<rdf:Description  rdf:about="http://www.semanticweb.org/ontologies/2011/10/
   Ontology1321208040015.owl#domainRObject">
   <Ontology1321208040015:hasObjectValue>
      Failure</Ontology1321208040015:hasObjectValue>
   <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/10/
      Ontology1321208040015.owl#Object"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/10/
   Ontology1321208040015.owl#domainRPredicate">
   <Ontology1321208040015:hasPredicateValue>
      Repair</Ontology1321208040015:hasPredicateValue>
   <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/10/
      Ontology1321208040015.owl#Predicate"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/10/
   Ontology1321208040015.owl#domainFailure">
   <Ontology1321208040015:hasDomainValue rdf:datatype="
      http://www.w3.org/2001/XMLSchema#string">DomainFailureManagement
   </Ontology1321208040015:hasDomainValue>
   <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/10/
      Ontology1321208040015.owl#Domain"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
</rdf:Description>
```

Abbildung 50: Ergebnis der Fehleranalyse in Form von semantischen Daten (RDF/XML-Serialisierung)

Der Tabelle 4 sind verschiedene Zeitmessungen für die Klassifikation, Fehlerstreuung und die Anwendung des JENA-Rule-Sets zu entnehmen.

	Ø Zeit für 10 – 100 Fehlerdatensätze	**Ø Zeit / Datensatz**
Klassifikation	–	ca. 8 µs
Fehlerstreuung + Schreiben der Ergebnisse in die Ontologie	ca. 62ms	–
Anwendung des JENA-Rule-Sets	ca. 320ms	–

Tabelle 4: Zeitmessung für die Klassifikation, Fehlerstreuung und das JENA-Rule-Set

Die Zeitmessungen (Tabelle 4) entsprechen einem Mittelwert aus jeweils zehn Messungen und wurden mit einem Intel Core i5 processor 520M (2,4 GHz, 3MB L3 Cache) und 4GB Memory durchgeführt.

6. Diskussion

In diesem Kapitel wird der Lösungsansatz zur Optimierung der IST-Analyse einer realen Produktionsumgebung einer bereits abgeschlossenen Arbeit, mit dem hier entwickelten Lösungsansatz verglichen.

Im Rahmen einer Bachelorthesis [33] wurde ein System für die Firma epro GmbH entwickelt, das eine kontinuierliche Analyse des Produktionszyklus und die grafische Aufbereitung der erzeugten Ergebnisse innerhalb der Sensorfertigung durchführt. Mit dem System sollen die Produktionskosten durch die Erkennung fehlerhafter Sensoren gesenkt werden. Als Grundlage für den Analyseprozess dienen verschiedene Messungen (Umgebungstemperatur, Induktivität, Spannungen, usw.) an den Sensoren innerhalb der Produktionsschritte.

In der Bachelorthesis werden zwei verschiedene Ansätze zur Analyse von fehlerhaften Sensoren vorgestellt:

- Neuronale Netze: Künstliche neuronale Netze (Abbildung 51) können zur Analyse von Produktionsdaten eingesetzt werden.

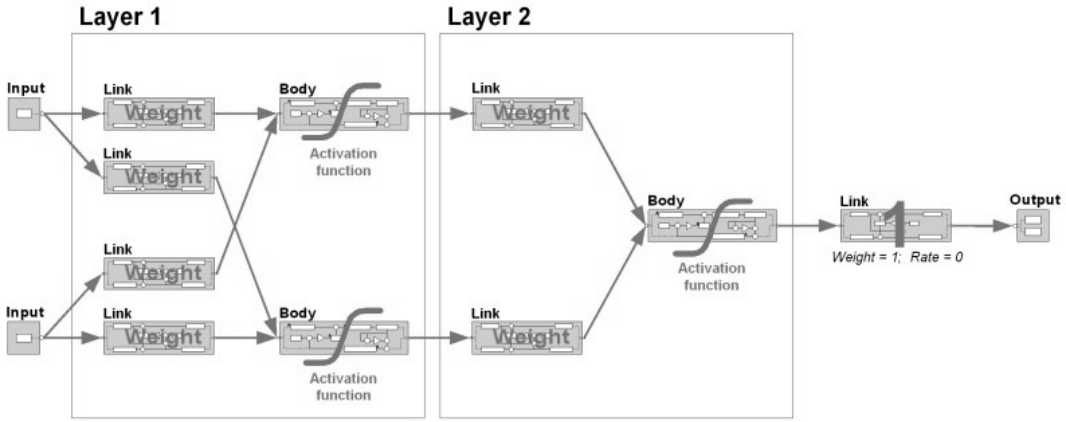

Abbildung 51: Neuronales Netz [35]

Die Eingangsneuronen werden mit Messwerten (Umgebungstemperatur, Induktivität, Spannungen, usw.) bzw. numerischen Daten belegt. Durch Gewichtungen der numerischen Werte werden bei Erreichen eines Schwellwertes Aktivierungsfunktionen ausgelöst, welche das Neuron feuern lässt. Der Output des neuronalen Netzes beschreibt die Eigenschaften eines

Sensors, anhand deren fehlerhafte Produkte erkannt werden können.

- Inferenzmaschine: Mit einer Inferenzmaschine ist der Einsatz regelbasierter Systeme in Produktionsumgebungen möglich. Vorteil einer Inferenzmaschine ist die gezielte Anpassung an die Gegebenheit und die flexible Ergebnisgestaltung. Es können beispielsweise neue Erkenntnisse durch eine Überarbeitung der Regeln direkt in den Analyseprozess eingebracht werden.

Die Erkennung fehlerhafter Produkte mittels einer Inferenzmaschine wurde in der Thesis von Christoph Zuleger in zwei Phasen unterteilt:

1. Analysephase: In dieser Phase werden Regeln zur Erkennung von Fehlmessungen und zur Bestimmung erwarteter Produktcharakteristika eingesetzt.
2. Reflektionsphase: In dieser Phase wird das Ergebnis der Analysephase repräsentiert. Da eine automatische Korrektur von Regeln nicht Gegenstand der Thesis von Christoph Zuleger ist, wird das Ergebnis der Analyse lediglich reflektiert. Ziel der Phase ist somit eine automatisierte Überwachung der Analyseergebnisse hinsichtlich charakteristischer Unstimmigkeiten.

Der Nachteil eines Regelsystems gegenüber eines neuronalen Netzes ist das Fehlen von adaptiven Eigenschaften. Es besteht nur die Möglichkeit durch das Manipulieren der Regeln neue Erkenntnisse in den Analyseprozess einzubringen [33].

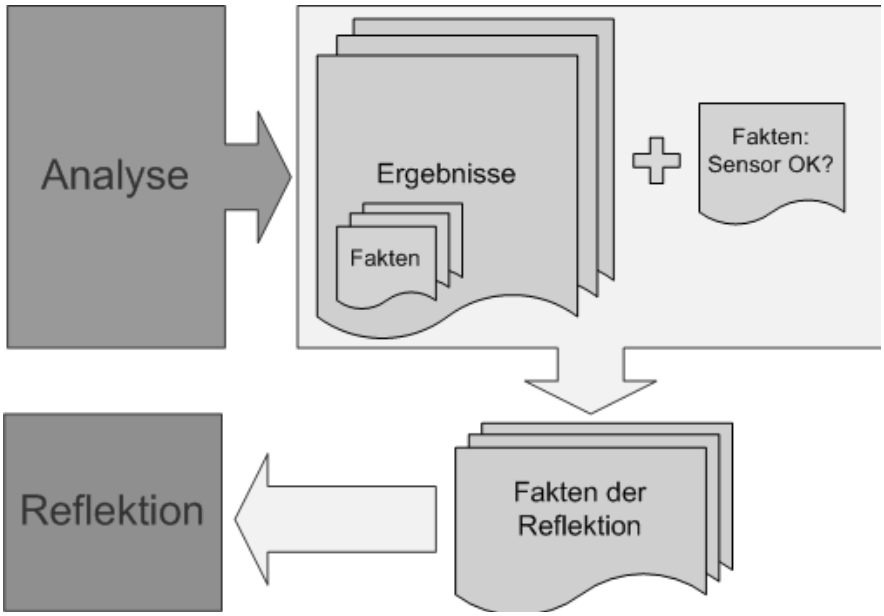

Abbildung 52: Analyse-, und Reflektionsphase [33]

Im Vergleich zu dieser Diplomarbeit wurde bei der Arbeit von Christoph Zuleger keine Inferenzmaschine verwendet, die mit Ontologien arbeitet, sondern es wurde die Inferenzmaschine *Drools* [34] von *JBoss* verwendet. Mit *Drools* lässt sich die Logik der Geschäftsprozesse in Regeln, Workflows und Events integrieren bzw. organisieren. Ein weiterer Unterschied ist, dass sich dem in dieser Diplomarbeit entwickelten Analyseprozess keine Reflektionsphase anschließt, der lediglich das Ergebnis der Analyse darstellt. Sondern das Ergebnis der Fehleranalyse wird in einer Transformationsphase als semantische Daten mit einer gleichzeitigen Erstellung einer Reparaturempfehlung dargestellt. Somit wird das manuelle Interpretieren des Analyseergebnisses von einem Mitarbeiter, durch einen automatisierten Vorgang ersetzt.

7. Zusammenfassung und Ausblick

Nach einer Einführung in die Bereiche der serviceorientierten Architektur, JAVA Messaging Service, semantische Daten und NEXUS DS wurde die Konzeption und Implementierung des Monitoring-Teils des MAPE-Zyklus, der für eine automatisierte Fehleranalyse nötig ist, vorgestellt. Als Grundlage für das Fehleranalysesystem diente die Lernfabrik des Instituts für industrielle Fertigung und Fabrikbetrieb (IFF) an der Universität Stuttgart. Mit Hilfe eines SQL-Triggers und dem *Java Messaging Service* werden die Log-Daten aus der Leitrechnerdatenbank der Lernfabrik exportiert und an die Source der Datenstromverarbeitung (NEXUS DS) gesendet. In vier weiteren Operatoren der Datenstromverarbeitung findet unter anderem durch die Ausführung einiger Regeln die Vorfehleranalyse der Log-Daten statt. Anschließend wird durch die Anwendung weiterer Regeln ein Lösungsvorschlag, wiederum als RDF/XML Serialisierung, an den letzten Operator der Datenstromverarbeitung, den Sink-Operator, gesendet. Dieser Operator ruft einen Web-Service auf, welcher das Provenance-aware Service Repository ist und den Lösungsvorschlag für die Durchführung der eigentlichen Fehleranalyse annimmt.
Zusammen mit den bereits implementierten Komponenten, dem Manufacturing Service Bus, dem Provenance-aware Service Repository und dem Integration Process Editor sind alle Voraussetzungen für eine automatisierte Fehleranalyse in der Produktion erfüllt. Zukünftig können solche Systeme die manuelle Fehleranalyse ersetzen und somit auf Veränderungen in der Produktion reagieren.
Eine Erweiterung für den hier implementierten Monitoring-Teil wäre das Hinzufügen einer Fehlerfrüherkennung. Man könnte durch einen prediktiven Algorithmus, welcher auf weitere Messdaten der Produktionsstraße und frühere Fehlerverläufe zugreift, voraussagen, dass es bald zu einem Fehler in der Produktionsstraße kommen könnte. Beispielsweise könnte man anhand der Bearbeitungszeit eines Produktes erkennen:

Wenn die Bearbeitungszeit für das selbe Produkt deutlich höher ist als in der Vergangenheit, könnte man Vorhersagen, dass aufgrund dessen in naher Zukunft ein Fehler innerhalb der Produktionsstraße gemeldet werden könnte.

Zudem bietet sich durch die sehr hohe Flexibilität der Datenstromverarbeitung NEXUS DS an, einzelne Operatoren des Ausführungsgraphs zu ersetzen, um somit eine andere

Art von Analyse durchzuführen. Beispielsweise könnte man durch das Ändern des Parameters *Mode* beim Selection-Operator und das Ersetzen des *Analysis & Transformation-Operators* durch einen Operator, welcher die Produktivität einer Fabrik analysiert, eine automatisierte Produktivitätsanalyse durchführen.

8. Quellenverzeichnis

[1] VNI Österreich. Verein Netzwerk Logistik. Hub and Spoke Systeme. Stand: Januar 2012. http://www.vnl.at/Hub-and-Spoke-Systeme.275.0.html.

[2] Minguez, J; Ruthardt, F; Riffelmacher, P; Scheibler, T; Mitschang, B: Service-based Integration in Event-driven Manufacturing Environments, Proceedings of the 1st Symposium on Web Intelligent Systems and Services (WISS), 11th International conference on Web Information System Engineering (WISE), Hong Kong, (2010)

[3] Ursula Kotzur. Betreuerin: Jutta Mülle. Sicherheit in Service-orientierten Architekturen. Seminar: Aktuelle Herausforderungen von Datenschutz und Datensicherheit in modernen Informationssystemen. IPD – Lehrstuhl für Systeme der Informationsverwaltung Universität Karlsruhe (TH), Sommersemester 2007.

[4] Dr. Klaus Manhart. Serviceorientierte Architekturen – Grundlegende Konzepte. TEC Channel IT Experts Inside, 28.11.2006. http://www.tecchannel.de/webtechnik/soa/456248/serviceorientierte_architekturen_grundlegende_konzepte/index4.html

[5] Cipriani, Nazario; Eissele, Mike; Brodt, Andreas; Gromann, Matthias; Mitschang, Bernhard: NexusDS: A Flexible and Extensible Middleware for Distributed Stream Processing, ACM (Hrsg): IDEAS ’09: Proceedings of the 2008 International Symposium on Database Engineering and Applications, 2009.

[6] Westkämper, E.: Deutschland – Standort mit Zukunft? Forschung für die Zukunft: Germany – Location with Future? Research for the Future.
In: Technische Univ. Chemnitz / Institut für Betriebswissenschaften und Fabriksysteme: Von der integrierten Fertigung zur vernetzten Produktion: Festschrift zum Ehrenkolloquium anlässlich des 70. Geburtstages von Prof. Siegfried Wirth, 13. Juli 2006, Chemnitz. Chemnitz, 2006, S. 27-50

[7] Dipl. Math. H. Max Straub. Advanced Services. Java – Java Messaging Service (JMS) – JMS Grundlagen. Stand: Januar 2012. http://www.straub.as/java/jms/basic.html

[8] Schwarz, Holger. Data-Warehouse-, Data-Mining- und OLAP-Technologien. Chapter 6: Data Mining. Universität Stuttgart, Wintersemester 2010/2011.

[9] Altova. Bibliothek – Semantic Web. Stand: Januar 2012. http://www.altova.com/de/semantic_web.html

[10] Boteram, Felix: Präsentation: Auszeichnungs- und Modellierungssprachen im Kontext semantischer Technologien. Vorlesung von Winfried Gödert, Prof. Dipl.-Math.: Datenformate, Datenmodellierung, Datenaustausch [BIB 7] - WS 2009/10. Fachhochschule Köln, 22.11.2007. http://www.fbi.fh-koeln.de/institut/personen/goedert/material/Boteram_Datenformate_Solo_02.ppt

[11] Stern, Matthias. Diplomarbeit: Verwendung von Ontologien zur Verbesserung von Informationsgewinn in E-Learningsystemen. Institut für Informationsverarbeitung und Computergestützte Neue Medien (IICM). Technische Universität Graz, Juni 2004.

[12] Augustin, A.; Kranz, M.; Schäfermeier, R. Seminar Moderne Webtechnologien: Semantic Web: Reasoners und Frameworks. AG Netzbasierte Informationssysteme, 2007. http://www.ag-nbi.de/lehre/07/S_MWT/Material/05_ReasonersFrameworks.pdf

[13] Wiki der Informatik Technische Universität Cottbus. Thema: JENA-Rules. Stand: Januar 2012. http://hydrogen.informatik.tu-cottbus.de/wiki/index.php/JenaRules

[14] Reynolds, Dave. The general purpose rule egine. Stand: 28.03.2010. http://jena.sourceforge.net/inference/#rules

[15] J. Kephart and D. Chess, “The Vision of Autonomic Computing,” *IEEE Computer*, Vol. 36, No. 1 (2003), pp. 41-50, http://www.research.ibm.com/autonomic/research/papers/AC_Vision_Computer_Jan_2003.pdf.

[16] Lernfabrik für advanced Industrial Engineering. Institut für Industrielle Fertigung und Fabrikbetrieb (IFF), Universität Stuttgart, http://www.lernfabrik-aie.de

[17] David E. Chappell. Enterprise Service BUS – Theory in Practice. O´Reilly, June 2004, 288 pages.

[18] Red Hat®, Jboss Community, Jboss Enterprise. Jboss Enterprise Service Bus (ESB), http://www.jboss.org/jbossesb

[19] IBM Software. WebSphere Enterprise Service Bus – ESB for quick integration of applications and processes, http://www-01.ibm.com/software/integration/wsesb

[20] IBM. Types of JDBC drivers, 2012. http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=%2Frzaha%2Fjdbctydr.htm

[21] GlassFish – Open Source Application Server, Januar 2012. http://www.sun.com/software/products/message_queue/index.xml

[22] Jboss. HonetQ – Message Oriented Middleware, Januar 2012. http://www.sun.com/software/products/message_queue/index.xml

[23] The Apache Software Foundation. Apache Incubators. JENA – JAVA-Framework. http://incubator.apache.org/jena/

[24] Minguez, J.; Niedermann, F.; Mitschang, B.; , ”A provenance-aware service repository for EAI process modeling tools,” 2011 IEEE International Conference on Information Reuse and Integration (IRI), pp.42-47 (2011)

[25] S. Kluge, P. Riffelmacher, V. Hummel, E. Westkämper. Montagesystemplanung – ein Handlungsfeld der Lernfabrik für aIE. wt Werkstatttechnik online Jahrgang 97 (2007) H.3, Seite 150 – 156.

[26] François Bry, Tim Furche und Dan Olteanu. Datenströme - Aktuelles Schlagwort / Datenströme. Informatik-Spektrum. Volume 27, Number 2, 168-171.

[27] David Luckham. The power of events. 2002.

[28] Roy Schulte David Luckham. Event processing glossary, version 1.1. 2008.

[29] F. Burger, P. Debicki, F. Kötter. Vergleich von Complex Event Processing-Ansätzen für Business Activity Monitoring. Fachstudie Nr. 112. Institut für Architektur und Anwendungssystemen, Universität Stuttgart, 2010.

[30] Michael Eckert, Francois Bry. Complex Event Processing (CEP). Institut für Informatik, Ludwig-Maximilians-Universität München. Gesellschaft für Informatik e.V., 2009. http://www.gi.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/complex-event-processing-cep-221.html

[31] Guido Schmutz (Trivadis), Torsten Winterberg (OPITZ CONSULTING GmbH). Event Driven Architecture (EDA) – Nutzen für moderne Anwendungslandschaften, Februar 2011. http://www.slideshare.net/opitzconsulting/event-driven-architecture-opitz-consulting-schmutz-winterberg

[32] Jakob Voß. Datenströme. Seminararbeit WS 2004/2005 bei Prof. Felix Naumann, Humboldt-Universität zu Berlin, 15.2.2005.

[33] Christoph Zuleger. Konzeption und Implementation eines Expertensystems zur Überwachung der Sensorfertigung bei der Firma epro GmbH. Bachelorthesis im Studiengang Angewandte Informatik, Fachhochschule Münster, September 2009.

[34] Red Hat®, Jboss Community, Drools – The Business Logic integration Platform, http://www.jboss.org/drools

[35] Bildquelle: Neuronales Netz, Februar 2012. http://www.colinfahey.com/neural_network_with_back_propagation_learning/neuron_network_forward_propagation.jpg

[36] Bildquelle: Dostal et al: Serviceorientierte Architekturen mit Web Services, Elsevier/Spektrum

[37] Bildquelle: Quelle: http://liris.cnrs.fr/abstract/fayyad1996.png

[38] Stjepan Grzan. Enabling technology for an indoor location aware information system. Diplomarbeit, Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR), Universität Stuttgart, Februar 2002.

Anhang

A1 Jena-Rule-Set zur Fehleranalyse

```
[rule1: (?m rdf:type msb:Module) (?m msb:hasFailure ?f) (?a rdf:type
msb:Analysis)(?a msb:hasAFirstMax ?curr) greaterThan(?f,?curr) -> (?a
msb:hasAFirstMax ?f)]

[rule2: (?m rdf:type msb:Module) (?m msb:hasFailure ?f) (?a rdf:type
msb:Analysis) (?a msb:hasAFirstMax ?curr) (?a msb:hasASecondMax ?
second) lessThan(?f,?curr) greaterThan(?f,?second) ->  (?a
msb:hasASecondMax ?f)]

[rule3: (?a rdf:type msb:Analysis) (?a msb:hasAFirstMax ?curr) (?a
msb:hasASecondMax ?second)  notEqual(?second,"0"^^xsd:int) quotient(?
curr ?second ?result) ge(?result,%Quotient_Para) -> (?a
msb:hasAAsserQuotient "true"^^xsd:boolean)]

[rule4: (?a rdf:type msb:Analysis) (?a msb:hasAAsserQuotient ?asser)
equal(?asser, "true"^^xsd:boolean) (?m rdf:type msb:Module) (?a
msb:hasAFirstMax ?max) (?m msb:hasFailure ?max) -> (?a
msb:hasAStopModule ?m)]

[rule5: (?a rdf:type msb:Analysis) -> (?a msb:hasAFaultyModule
%FaultyModule)]

[rule6: (?a rdf:type msb:Analysis) (?a msb:hasAStopModule ?stop) (?f
rdf:type msb:Failure) (?f msb:hasModule ?fmod) equal(?fmod,?stop) (?f
msb:hasProcess ?proc) (?a msb:hasACountProcFailure ?cpf) (?f
msb:hasFailure ?fcount) greaterThan(?fcount,?cpf) -> (?a
msb:hasACountProcFailure ?fcount)]

[rule7: (?a rdf:type msb:Analysis) (?a msb:hasAStopModule ?stop) (?f
rdf:type msb:Failure) (?f msb:hasModule ?fmod) equal(?fmod,?stop) (?f
msb:hasProcess ?proc) (?a msb:hasACountProcFailure ?cpf) (?f
msb:hasFailure ?cpf) -> (?a msb:hasAFaultyProcess ?proc)]

[rule8: (?a rdf:type msb:Analysis) (?a msb:hasAFaultyModule ?mod) (?a
msb:hasAFaultyProcess ?proc) (?e rdf:type msb:Evaluation) (?e
msb:hasEModule ?mod) (?e msb:hasEProcess ?proc) (?e
msb:hasEAssemblyInstructionsAVGGrade ?aig) ge(?aig,"4.0"^^xsd:float)
-> (?a msb:hasAaigConfirmed "true"^^xsd:boolean)]

[rule9: (?a rdf:type msb:Analysis) (?a msb:hasAFaultyModule ?mod) (?a
msb:hasAFaultyProcess ?proc) (?e rdf:type msb:Evaluation) (?e
msb:hasEModule ?mod) (?e msb:hasEProcess ?proc) (?e
msb:hasEMaturityOfVariante ?mof) le(?mof,"2.0"^^xsd:float) -> (?a
msb:hasAmofConfirmed "true"^^xsd:boolean)]
```

```
[rule10: (?a rdf:type msb:Analysis) (?a msb:hasAFaultyModule ?mod) (?a
msb:hasAFaultyProcess ?proc) (?e rdf:type msb:Evaluation) (?e
msb:hasEModule ?mod) (?e msb:hasEProcess ?proc) (?e
msb:hasEPressureOfTimeAVGGrade ?potg) ge(?potg,"4.0"^^xsd:float) -> (?
a msb:hasApotgConfirmed "true"^^xsd:boolean)]

[rule11: (?a rdf:type msb:Analysis) (?a msb:hasAFaultyModule ?mod) (?a
msb:hasAFaultyProcess ?proc) (?e rdf:type msb:Evaluation) (?e
msb:hasEModule ?mod) (?e msb:hasEProcess ?proc) (?e
msb:hasEErgonomicsAVGGrade ?eg) ge(?eg,"4.0"^^xsd:float) -> (?a
msb:hasAegConfirmed "true"^^xsd:boolean)]

[rule12: noValue(?a,msb:hasAErrorConfirmed) (?a rdf:type msb:Analysis)
(?a msb:hasAegConfirmed "true"^^xsd:boolean) (?a msb:hasApotgConfirmed
"true"^^xsd:boolean) (?a msb:hasAmofConfirmed "true"^^xsd:boolean) ->
(?a msb:hasAErrorConfirmed "true"^^xsd:boolean)]

[rule13: noValue(?a,msb:hasAErrorConfirmed) (?a rdf:type msb:Analysis)
(?a msb:hasAaigConfirmed "true"^^xsd:boolean) (?a
msb:hasApotgConfirmed "true"^^xsd:boolean) (?a msb:hasAmofConfirmed
"true"^^xsd:boolean) -> (?a msb:hasAErrorConfirmed
"true"^^xsd:boolean)]

[rule14: noValue(?a,msb:hasAErrorConfirmed) (?a rdf:type msb:Analysis)
(?a msb:hasAaigConfirmed "true"^^xsd:boolean) (?a msb:hasAegConfirmed
"true"^^xsd:boolean) (?a msb:hasAmofConfirmed "true"^^xsd:boolean) ->
(?a msb:hasAErrorConfirmed "true"^^xsd:boolean)]

[rule15: noValue(?a,msb:hasAErrorConfirmed) (?a rdf:type msb:Analysis)
(?a msb:hasAaigConfirmed "true"^^xsd:boolean) (?a msb:hasAegConfirmed
"true"^^xsd:boolean) (?a msb:hasApotgConfirmed "true"^^xsd:boolean) ->
(?a msb:hasAErrorConfirmed "true"^^xsd:boolean)]

[rule16: noValue(?a,msb:hasAErrorConfirmed) (?a rdf:type msb:Analysis)
(?a msb:hasAaigConfirmed "true"^^xsd:boolean) (?a msb:hasAegConfirmed
"true"^^xsd:boolean) (?a msb:hasApotgConfirmed "true"^^xsd:boolean) (?
a msb:hasAmofConfirmed "true"^^xsd:boolean) -> (?a
msb:hasAErrorConfirmed "true"^^xsd:boolean)]

[rule17: (?a rdf:type msb:Analysis) (?a msb:hasAErrorConfirmed
"true"^^xsd:boolean) (?Cind rdf:type dr:Condition) (?a
msb:hasAFaultyProcess ?proc) strConcat(?proc,?procStr) -> (?Cind
dr:hasConditionValue ?procStr)]

[rule18: (?a rdf:type msb:Analysis) (?a msb:hasAErrorConfirmed
"true"^^xsd:boolean) (?Dom rdf:type dr:Domain) -> (?Dom
dr:hasDomainValue "DomainFailureManagement")]

[rule19: (?a rdf:type msb:Analysis) (?a msb:hasAErrorConfirmed
"true"^^xsd:boolean) (?Manner rdf:type dr:Manner) equal("1"^^xsd:int,
%RepairInstruct) (?a msb:hasAisFModRobotStation ?robot) equal(?
robot,"false"^^xsd:boolean) -> (?Manner dr:hasMannerValue "online")]
```

```
[rule20: (?Manner rdf:type dr:Manner) noValue(?
Manner,dr:hasMannerValue) -> (?Manner dr:hasMannerValue "offline")]

[rule21: (?a rdf:type msb:Analysis) (?a msb:hasAErrorConfirmed
"true"^^xsd:boolean) (?Loc rdf:type dr:Location) (?a
msb:hasAFaultyModule ?fm) strConcat(?fm,?fmStr)  -> (?Loc
dr:hasLocationValue ?fmStr)]

[rule22: (?a rdf:type msb:Analysis) (?a msb:hasAErrorConfirmed
"true"^^xsd:boolean) (?Pred rdf:type dr:Predicate) (?Obj rdf:type
dr:Object) -> (?Pred dr:hasPredicateValue "Repair") (?Obj
dr:hasObjectValue "Failure")]
```

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Ludwigsburg, 15.03.2012